Doctrine 2 ORM Documentation

Release 2.1

Doctrine Project Team :: Traducido por Nacho Pacheco

November 03, 2011

Índice general

1. Guía de referencia **1**
- 1.1. Introducción . . . 1
- 1.2. Arquitectura . . . 6
- 1.3. Configurando . . . 8
- 1.4. Preguntas más frecuentes . . . 15
- 1.5. Asignación básica . . . 18
- 1.6. Asignando asociaciones . . . 28
- 1.7. Asignando herencia . . . 46
- 1.8. Trabajando con objetos . . . 49
- 1.9. Trabajando con asociaciones . . . 61
- 1.10. Transacciones y concurrencia . . . 70
- 1.11. Eventos . . . 75
- 1.12. Procesamiento masivo . . . 83
- 1.13. Lenguaje de consulta *Doctrine* . . . 86
- 1.14. El generador de consultas . . . 109
- 1.15. `SQL` nativo . . . 116
- 1.16. Change Tracking Policies . . . 122
- 1.17. Partial Objects . . . 124
- 1.18. Asignación XML . . . 125
- 1.19. Asignación YAML . . . 134
- 1.20. Referencia de anotaciones . . . 136
- 1.21. PHP Mapping . . . 147
- 1.22. Memoria caché . . . 152
- 1.23. Mejorando el rendimiento . . . 158
- 1.24. Herramientas . . . 159
- 1.25. Metadata Drivers . . . 165
- 1.26. Buenas prácticas . . . 167
- 1.27. Limitations and Known Issues . . . 169

2. Guías iniciales **173**
- 2.1. Primeros pasos . . . 173
- 2.2. Trabajando con asociaciones indexadas . . . 192
- 2.3. Extra Lazy Associations . . . 197

2.4. Composite and Foreign Keys as Primary Key 198

3. Recetario **205**
3.1. Campos agregados 205
3.2. Extendiendo DQL en *Doctrine 2*: Paseantes AST personalizados 210
3.3. Funciones *DQL* definidas por el usuario 213
3.4. Implementing ArrayAccess for Domain Objects 217
3.5. Implementing the Notify ChangeTracking Policy 218
3.6. Implementing Wakeup or Clone 219
3.7. Integrating with CodeIgniter 220
3.8. SQL-Table Prefixes 223
3.9. Strategy-Pattern 224
3.10. Validation of Entities 228
3.11. Working with DateTime Instances 230
3.12. Mysql Enums 232

CAPÍTULO 1

Guía de referencia

1.1 Introducción

1.1.1 Bienvenido

Doctrine 2 es un asignador objetorelacional (ORM) para *PHP* 5.3.0+ que proporciona persistencia transparente de objetos PHP. Se sitúa en la parte superior de una poderosa capa de abstracción de base de datos (DBAL por DataBase Abstraction Layer). La principal tarea de los asignadores objetorelacionales es la traducción transparente entre objetos (PHP) y las filas relacionales de la base de datos.

Una de las características clave de *Doctrine* es la opción de escribir las consultas de base de datos en un dialecto `SQL` propio orientado a objetos llamado *Lenguaje de Consulta Doctrine* (`DQL` por *Doctrine Query Language*), inspirado en Hibernates HQL. Además `DQL` difiere ligeramente de `SQL` en que abstrae considerablemente la asignación entre las filas de la base de datos y objetos, permitiendo a los desarrolladores escribir poderosas consultas de una manera sencilla y flexible.

1.1.2 Descargo de responsabilidad

Esta es la documentación de referencia de *Doctrine* 2. Las guías introductorias y tutoriales que puedes seguir de principio a fin, como el libro "Guía para *Doctrine*" más conocido de la serie *Doctrine* 1.x, estará disponible en una fecha posterior.

1.1.3 Usando un asignador objetorelacional

Cómo insinúa el término ORM, *Doctrine* 2 tiene como objetivo simplificar la traducción entre las filas de la base de datos y el modelo de objetos PHP. El caso de uso principal para *Doctrine*, por lo tanto son las aplicaciones que utilizan el paradigma de programación orientado a objetos. Para aplicaciones que no trabajan principalmente con objetos, *Doctrine* 2 no se adapta muy bien.

1.1.4 Requisitos

Doctrine 2 requiere un mínimo de *PHP* 5.3.0. Para obtener un rendimiento mejorado en gran medida, también se recomienda que utilices APC con PHP.

1.1.5 Paquetes *Doctrine* 2

Doctrine 2 se divide en tres paquetes principales.

- Común
- DBAL (incluye Común)
- ORM (incluye DBAL+Común)

Este manual cubre principalmente el paquete ORM, a veces toca partes de los paquetes subyacentes DBAL y Común. El código base de *Doctrine* está dividido en estos paquetes por varias razones y se van a ...

- ... Hace las cosas más fáciles de mantener y desacopladas
- ... te permite usar código de *Doctrine* común sin el ORM o DBAL
- ... te permite usar DBAL sin el ORM

El paquete Común

El paquete Común contiene componentes altamente reutilizables que no tienen dependencias más allá del propio paquete (y PHP, por supuesto). El espacio de nombres raíz del paquete Común es `Doctrine\Common`.

El paquete DBAL

El paquete DBAL contiene una capa de abstracción de base de datos mejorada en lo alto de PDO, pero no está vinculada fuertemente a PDO. El propósito de esta capa es proporcionar una sola *API* que fusione la mayor parte de las diferencias entre los diferentes proveedores RDBMS. El espacio de nombres raíz del paquete DBAL es `Doctrine\DBAL`.

El paquete ORM

El paquete ORM contiene las herramientas de asignación objetorelacional que proporcionan persistencia relacional transparente de objetos PHP sencillos. El espacio de nombres raíz del paquete ORM es `Doctrine\ORM`.

1.1.6 Instalando

Puedes instalar *Doctrine* de diferentes maneras. Vamos a describir todas las diferentes maneras y tú puedes elegir la que mejor se adapte a tus necesidades.

PEAR

Puedes instalar cualquiera de los tres paquetes de *Doctrine* desde la utilidad de instalación de la línea de ordenes PEAR.

Para instalar sólo el paquete `Común` puedes ejecutar la siguiente orden:

```
$ sudo pear install pear.doctrine-project.org/DoctrineCommon-<versión>
```

Si deseas utilizar la capa de abstracción de base de datos de *Doctrine* la puedes instalar con la siguiente orden.

```
$ sudo pear install pear.doctrine-project.org/DoctrineDBAL-<versión>
```

O, si quieres conseguir las tareas e ir por el ORM lo puedes instalar con la siguiente orden.

```
$ sudo pear install pear.doctrine-project.org/DoctrineORM-<versión>
```

Nota: La etiqueta `<versión>` anterior representa la versión que deseas instalar. Por ejemplo, si la versión actual al momento de escribir esto es `2.0.7` para el ORM, por lo tanto la podrías instalar con lo siguiente:

```
$ sudo pear install pear.doctrine-project.org/DoctrineORM-2.0.7
```

Cuando instalas un paquete mediante PEAR puedes requerir y cargar el cargador de clases `ClassLoader` con el siguiente código.

```
<?php
require 'Doctrine/Common/ClassLoader.php';
$classLoader = new \Doctrine\Common\ClassLoader('Doctrine');
$classLoader->register();
```

Los paquetes se instalan en tu directorio de código compartido PEAR PHP en un directorio llamado `doctrine`. También consigues instalar una agradable utilidad de línea de ordenes y a tu disposición en el sistema. Ahora, cuando ejecutes la orden `doctrine` verás lo que puedes hacer con ella.

```
$ doctrine
Interfaz de la línea de ordenes de Doctrine versión 2.0.0BETA3-DEV

Uso:
  [opciones] orden [argumentos]

Opciones:
  --help           -h Muestra este mensaje de ayuda.
  --quiet          -q No muestra ningún mensaje.
  --verbose        -v Incrementa el detalle de los mensajes.
  --version        -V Muestra la versión del programa.
  --color          -c Fuerza la salida ANSI en color.
  --no-interaction -n No hace ninguna pregunta interactivamente.

Ordenes disponibles:
  help                          Muestra ayuda para una orden (?)
  list                          Lista las ordenes
dbal
  :import                       Importa archivo(s) ``SQL`` directamente a la base de datos.
  :run-sql                      Ejecuta ``SQL`` arbitrario directamente desde la línea de ordenes.
orm
  :convert-d1-schema            Convierte el esquema *Doctrine* 1.X al esquema *Doctrine* 2.X.
  :convert-mapping              Convierte información de asignación entre los formatos compatibles.
  :ensure-production-settings   Verifica que *Doctrine* está configurado apropiadamente para un entorn
  :generate-entities            Genera clases entidad y métodos cooperantes a partir de la informació
  :generate-proxies             Genera clases delegadas para clases entidad.
  :generate-repositories        Genera clases repositorio desde tu información de asignación.
  :run-dql                      Ejecuta ``DQL`` arbitrario directamente desde la línea de ordenes.
  :validate-schema              Valida la asignación de archivos.
orm:clear-cache
  :metadata                     Borra todos los metadatos en caché de varios controladores de caché.
  :query                        Borra todas las consultas en caché de varios controladores de caché.
  :result                       Borra el resultado en caché de varios controladores de caché.
orm:schema-tool
  :create                       Procesa el esquema y, o bien lo crea directamente en la conexión de al
  :drop                         Procesa el esquema y, o bien borra el esquema de base de datos de la c
  :update                       Procesa el esquema y, o bien actualiza el esquema de la base de datos
```

Descargando el paquete

También puedes utilizar *Doctrine* 2 descargando la última versión del paquete de la página de descarga.

Ve la sección de configuración sobre cómo configurar y arrancar una versión de *Doctrine* descargada.

GitHub

Alternativamente, puedes clonar la última versión de *Doctrine* 2 a través de GitHub.com:

```
$ git clone git://github.com/doctrine/doctrine2.git doctrine
```

Esto descarga todas las fuentes del paquete ORM. Necesitas iniciar los submódulos Github para las dependencias del paquete `Common` y `DBAL`:

```
$ git submodule init
$ git submodule update
```

Esto actualiza a Git para utilizar *Doctrine* y las versiones de los paquetes recomendados de *Doctrine* para la versión Maestra clonada de *Doctrine* 2.

Consulta el capítulo de configuración sobre cómo configurar una instalación Github de *Doctrine* con respecto a la carga automática.

> **NOTA**
>
> No debes combinar consignaciones *Doctrine-Common, Doctrine-DBAL* y *Doctrine-ORM* maestras con las demás en combinación. El ORM posiblemente no funcione con las versiones maestras de `Common` o `DBAL` actuales. En su lugar el ORM viene con los submódulos Git que se requieren.

Subversion

> **NOTA**
>
> Usar el espejo SVN no es recomendable. Este sólo te permite acceso a la última confirmación maestra y no busca los submódulos automáticamente.

Si prefieres subversión también puedes descargar el código desde GitHub.com a través del protocolo de subversión:

```
$ svn co http://svn.github.com/doctrine/doctrine2.git doctrine2
```

Sin embargo, esto sólo te permite ver el maestro de *Doctrine* 2 actual, sin las dependencias `Common` y `DBAL`. Las tienes que tomar tú mismo, pero te podrías encontrar con incompatibilidad entre las versiones de las diferentes ramas maestras `Common`, `DBAL` y `ORM`.

Inicio rápido con el recinto de seguridad

> **NOTA** El recinto de seguridad sólo está disponible a través del repositorio Github de Doctrine2 o tan pronto como lo descargues por separado en la página de descargas. Lo encontrarás en el directorio $root/tools/sandbox.

El recinto de seguridad es un entorno preconfigurado para evaluar y jugar con *Doctrine* 2.

1.1.7 Descripción

Después de navegar por el directorio `sandbox`, deberías ver la siguiente estructura:

```
sandbox/
    Entities/
        Address.php
        User.php
    xml/
        Entities.Address.dcm.xml
        Entities.User.dcm.xml
    yaml/
        Entities.Address.dcm.yml
        Entities.User.dcm.yml
    cli-config.php
    doctrine
    doctrine.php
    index.php
```

Aquí está un breve resumen del propósito de estos directorios y archivos:

- El directorio `Entities` es donde se han creado las clases del modelo. Allí están dos entidades de ejemplo.
- El directorio `xml` es donde se crean todos los archivos de asignación XML (si deseas utilizar la asignación XML). Ya están allí dos ejemplos de asignación de documentos para 2 entidades de ejemplo.
- El directorio `yaml` es donde se crean los archivos de asignación YAML (si deseas utilizar la asignación YAML). Ya están allí dos ejemplos de asignación de documentos para 2 entidades de ejemplo.
- `cli-config.php` contiene código de arranque para una configuración que utiliza la herramienta de consola `doctrine` cada vez que se ejecuta una tarea.
- `doctrine`/`doctrine.php` es una herramienta de línea de ordenes.
- `index.php` es un archivo de arranque clásico de una aplicación php que utiliza *Doctrine* 2.

Minitutorial

1. Desde el directorio `tools/sandbox`, ejecuta la siguiente orden y deberías ver el mismo resultado.

   $ php doctrine orm:schema-tool:create

   Creando el esquema de base de datos...

   ¡Esquema de base de datos creado satisfactoriamente!

2. Dale otro vistazo al directorio `tools/sandbox`. Se ha creado un esquema de base de datos SQLite con el nombre `database.sqlite`.
3. Abre `index.php` y en la parte inferior haz las correcciones necesarias para que sea de la siguiente manera:

```php
<?php
//... cosas del arranque

## COLOCA TU CÓDIGO ABAJO

$user = new \Entities\User;
$user->setName('Garfield');
$em->persist($user);
$em->flush();

echo "¡Usuario guardado!";
```

Abre index.php en el navegador o ejecútalo en la línea de ordenes. Deberías ver "¡guardado por el usuario!".

4. Inspecciona la base de datos SQLite. Una vez más dentro del `directorio/sandbox`, ejecuta la siguiente orden:

   $ php doctrine dbal:run-sql "select * from users"

Debes obtener la siguiente salida:

```
array(1) {
  [0]=>
  array(2) {
    ["id"]=>
    string(1) "1"
    ["name"]=>
    string(8) "Garfield"
  }
}
```

Acabas de guardar tu primera entidad con un identificador generado en una base de datos SQLite.

5. Reemplaza el contenido de `index.php` con lo siguiente:

```
<?php
//... cosas del arranque

## COLOCA TU CÓDIGO ABAJO

$q = $em->createQuery('select u from Entities\User u where u.name = ?1');
$q->setParameter(1, 'Garfield');
$garfield = $q->getSingleResult();

echo "Hola " . $garfield->getName() . "!";
```

Acabas de crear tu primer consulta `DQL` para recuperar al usuario con el nombre 'Garfield' de una base de datos SQLite (Sí, hay una manera más fácil de hacerlo, pero hemos querido presentarte a `DQL` en este momento. ¿Puedes **encontrar** el camino más fácil?).

> **SUGERENCIA** Al crear nuevas clases del modelo o modificar los ya existentes puedes recrear el esquema de base de datos con la orden `'doctrine orm:schema-tool --drop` seguida de `doctrine orm:schema-tool --create`.

6. ¡Explora *Doctrine* 2!

En lugar de leer el manual de referencia también te recomendamos ver los tutoriales:

Tutorial primeros pasos

1.2 Arquitectura

Este capítulo ofrece una descripción de la arquitectura en general, la terminología y las limitaciones de *Doctrine* 2. Te recomiendo leer detenidamente este capítulo.

1.2.1 Entidades

Una entidad es un ligero, objeto persistente del dominio. Una entidad puede ser cualquier clase PHP regular que observa las siguientes restricciones:

- Una clase entidad no debe ser final o contener métodos final.
- Todas las propiedades persistentes/campo de cualquier clase de entidad siempre deben ser privadas o protegidas, de lo contrario la carga diferida podría no funcionar como se espera.
- Una clase entidad no debe implementar `__clone` o debe *hacerlo de manera segura*.
- Una clase entidad no debe implementar `__wakeup` o debe *hacerlo de manera segura*. También considera implementar serializable en su lugar.
- Cualquiera de las dos clases entidad en una jerarquía de clases que heredan directa o indirectamente la una de la otra, no deben tener asignada una propiedad con el mismo nombre. Es decir, si B hereda de A entonce B no debe tener un campo asignado con el mismo nombre que un campo ya asignado heredado de A.
- **Una entidad no puede usar `func_get_args()` para implementar parámetros variables.** Los delegados generados no son compatibles con este por razones de rendimiento y tu código en realidad podría dejar de funcionar cuando viola esta restricción.

Las entidades admiten la herencia, asociaciones polimórficas, y consultas polimórficas. Ambas clases abstractas y específicas pueden ser entidades. Las entidades pueden extender clases que no son entidad, así como clases entidad, y las clases que no son entidad pueden extender a las clases entidad.

> **Truco** El constructor de una entidad sólo se invoca cuando *tú* construyes una nueva instancia con la palabra clave *new*. *Doctrine* nunca llama a los constructores de la entidad, por lo tanto eres libre de utilizarlos como desee e incluso tienes que requerir argumentos de algún tipo.

Estados de la entidad

Una instancia de entidad se puede caracterizar como *NEW*, *MANAGED*, *DETACHED* o *REMOVED*.

- Una *NEW* instancia de la entidad no tiene identidad persistente, y todavía no está asociada con un `EntityManager` y `UnitOfWork` (es decir, se acaba de crear con el operador "new").
- Una instancia de la entidad *MANAGED* es una instancia con una identidad persistente que se asocia con un *EntityManager* y cuya persistencia se maneja así.
- A instancia de entidad *DETACHED* es una instancia con una identidad persistente que no es (o no) asociada a un *EntityManager* y una *UnitOfWork*.
- A instancia de entidad *REMOVED* es una instancia con una identidad persistente, asociada a un *EntityManager*, que se eliminará de la base de datos al confirmar la transacción.

Campos persistentes

El estado persistente de una entidad lo representan las variables de la instancia. Una variable de instancia se debe acceder directamente sólo desde dentro de los métodos de la entidad por la instancia de la entidad en sí. Las variables de instancia no se deben acceder por los clientes de la entidad. El estado de la entidad está a disposición de los clientes solamente a través de los métodos de la entidad, es decir, métodos de acceso (métodos captador/definidor) u otros métodos del negocio.

Una colección valorada de campos persistentes y las propiedades se debe definir en términos de la interfaz `Doctrine\Common\Collections\Collection`. El tipo de implementación de la colección se puede utilizar por la aplicación para iniciar los campos o propiedades antes de persistir la entidad. Una vez que la entidad se manejó (o separó), el posterior acceso debe ser a través del tipo de la interfaz.

Serializando entidades

La serialización de entidades puede ser problemática y no se recomienda en realidad, al menos no mientras una instancia de la entidad aún mantenga referencias a objetos delegados o si todavía está gestionada por un `EntityManager`. Si vas a serializar (y deserializar) instancias de la entidad que todavía mantienen referencias a objetos delegados puedes tener problemas con la propiedad privada debido a limitaciones técnicas. Los objetos delegados implementan `__sleep` y `__sleep` no tiene la posibilidad para devolver nombres de las propiedades privadas de las clases padre. Por otro lado, no es una solución para los objetos delegados implementar `serializable` porque `serializable` no funciona bien con todas las referencias a objetos potencialmente cíclicas (por lo menos no hemos encontrado una manera, sin embargo, si lo consigues, por favor ponte en contacto con nosotros).

1.2.2 El `EntityManager`

La clase `EntityManager` es un punto de acceso central a la funcionalidad ORM proporcionada por *Doctrine* 2. La *API* de `EntityManager` se utiliza para gestionar la persistencia de los objetos y para consultar objetos persistentes.

Escritura transaccionales en segundo plano

Un `EntityManager` y la `UnitOfWork` subyacente emplean una estrategia denominada "escritura transaccional en segundo plano", que retrasa la ejecución de las declaraciones `SQL` con el fin de ejecutarlas de la manera más eficiente y para ejecutarlas al final de una transacción, de forma que todos los bloqueos de escritura sean liberados rápidamente. Deberías ver a *Doctrine* como una herramienta para sincronizar tus objetos en memoria con la base de datos en unidades de trabajo bien definidas. Trabajas con tus objetos y los modificas, como de costumbre y cuando termines llamas a `EntityManager#flush()` para persistir tus cambios.

La unidad de trabajo

Internamente un `EntityManager` utiliza una `UnitOfWork`, la cual es una típica implementación del patrón unidad de trabajo, para realizar un seguimiento de todas las cosas que hay que hacer la próxima vez que invoques a `flush`. Por lo general, no interactúas directamente con una `UnitOfWork` sino con el `EntityManager` en su lugar.

1.3 Configurando

1.3.1 Proceso de arranque

El arranque de *Doctrine* es un procedimiento relativamente sencillo que consta más o menos de tan sólo 2 pasos:

- Se asegura de que los archivos de clase de *Doctrine* se pueden cargar bajo demanda.
- Obtiene una instancia del `EntityManager`.

Carga de clase

Vamos a empezar con la configuración de la carga de clases. Tenemos que configurar algún cargador de clases (a menudo llamado "autocargador") para que los archivos de las clases de *Doctrine* se carguen bajo demanda. El espacio de nombres *Doctrine* contiene un cargador de clases minimalista muy rápido y se puede utilizar para *Doctrine* y cualquier otra biblioteca, donde los estándares de codificación garantizan que la ubicación de una clase en el árbol de directorios se refleja en su nombre y espacio de nombres y donde hay un espacio de nombres raíz común.

Nota: No estás obligado a utilizar el cargador de clases de *Doctrine* para cargar las clases de *Doctrine*. A *Doctrine* no le importa cómo se cargan las clases, si quieres usar un gestor de clases o cargar las clases de *Doctrine* tú mismo, sólo hazlo. En la misma línea, el cargador de clases del espacio de nombres de *Doctrine* no está destinado a utilizarse para las clases de *Doctrine*, tampoco. Se trata de un cargador de clases genérico que puedes utilizar para cualquier clase que siga algunas normas de nomenclatura básicas descritas anteriormente.

El siguiente ejemplo muestra la configuración de un *cargador de clases* para los diferentes tipos de instalaciones de *Doctrine*:

Nota: Este asume que has creado algún tipo de guión para probar el siguiente código con él. Algo así como un archivo `test.php`.

Descarga PEAR o Tarball

```
<?php
// test.php

require '/path/to/libraries/Doctrine/Common/ClassLoader.php';
$classLoader = new \Doctrine\Common\ClassLoader('Doctrine', '/path/to/libraries');
$classLoader->register(); // registra una pila de autocarga SPL
```

Git

El arranque de Git supone que ya has recibido los paquetes de actualización relacionados a través de `git submodule --init`

```
<?php
// test.php

$lib = '/ruta/a/doctrine2-orm/lib/';
require $lib . 'vendor/doctrine-common/lib/Doctrine/Common/ClassLoader.php';

$classLoader = new \Doctrine\Common\ClassLoader('Doctrine\Common', $lib . 'vendor/doctrine-common/lib
$classLoader->register();

$classLoader = new \Doctrine\Common\ClassLoader('Doctrine\DBAL', $lib . 'vendor/doctrine-dbal/lib');
$classLoader->register();

$classLoader = new \Doctrine\Common\ClassLoader('Doctrine\ORM', $lib);
$classLoader->register();
```

Componentes adicionales de Symfony

Si no utilizas *Doctrine* 2 en combinación con *Symfony2* tienes que registrar un espacio de nombres adicional para poder usar la herramienta *Doctrine-CLI* o el controlador de asignación YAML:

```
<?php
// configuración PEAR o Tarball
$classloader = new \Doctrine\Common\ClassLoader('Symfony', '/ruta/a/libraries/Doctrine');
$classloader->register();

// Git Setup
```

```
$classloader = new \Doctrine\Common\ClassLoader('Symfony', $lib . 'vendor/');
$classloader->register();
```

Para un mejor rendimiento de la carga de clases, es recomendable que mantengas corto tu *include_path*, lo ideal es que sólo debe contener la ruta a las bibliotecas de PEAR, y cualquier otra clase de biblioteca se debe registrar con su ruta base completa.

Obteniendo un `EntityManager`

Una vez que hayas preparado la carga de clases, adquiere una instancia del `EntityManager`. La clase *EntityManager* es el principal punto de acceso a la funcionalidad proporcionada por el ORM de *Doctrine*.

Una configuración simple del `EntityManager` requiere una instancia de `Doctrine\ORM\Configuration`, así como algunos parámetros de conexión para la base de datos:

```
<?php
use Doctrine\ORM\EntityManager,
    Doctrine\ORM\Configuration;

// ...

if ($applicationMode == "development") {
    $cache = new \Doctrine\Common\Cache\ArrayCache;
} else {
    $cache = new \Doctrine\Common\Cache\ApcCache;
}

$config = new Configuration;
$config->setMetadataCacheImpl($cache);
$driverImpl = $config->newDefaultAnnotationDriver('/path/to/lib/MiProyecto/Entities');
$config->setMetadataDriverImpl($driverImpl);
$config->setQueryCacheImpl($cache);
$config->setProxyDir('/path/to/myproject/lib/MiProyecto/Proxies');
$config->setProxyNamespace('MiProyecto\Proxies');

if ($applicationMode == "development") {
    $config->setAutoGenerateProxyClasses(true);
} else {
    $config->setAutoGenerateProxyClasses(false);
}

$connectionOptions = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite'
);

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

Nota: ¡No utilices *Doctrine* sin metadatos y caché de consultas! *Doctrine* está altamente optimizado para trabajar con las cachés. Las partes en *Doctrine* que principalmente se han optimizado para el almacenamiento en caché son la información de asignación de metadatos con el caché de metadatos y las conversiones de `DQL` a `SQL` con la caché de consultas. Sin embargo, estas dos caches sólo requieren un mínimo de la memoria absoluta, en gran medida mejoran el rendimiento en tiempo de la ejecución de *Doctrine*. El controlador de memoria caché recomendado para usar *Doctrine* es APC. APC te proporciona un código de operación cache (que se recomienda de cualquier modo) y un muy rápido almacenamiento en memoria caché que puedes utilizar para los metadatos y las caches de consulta como muestra el fragmento de código anterior.

1.3.2 Opciones de configuración

En las siguientes secciones se describen todas las opciones de configuración disponibles en una instancia de `Doctrine\ORM\Configuration`.

Directorio delegado (*REQUERIDO*)

```
<?php
$config->setProxyDir($dir);
$config->getProxyDir();
```

Obtiene o establece el directorio en el que *Doctrine* genera algunas clases delegadas. Para una detallada explicación sobre las clases delegadas y cómo se utilizan en *Doctrine*, consulta la sección *Objetos delegados* más adelante.

Espacio de nombres delegado (*REQUERIDO*)

```
<?php
$config->setProxyNamespace($namespace);
$config->getProxyNamespace();
```

Obtiene o establece el espacio de nombres a utilizar para las clases delegadas generadas. Para una detallada explicación sobre las clases delegadas y cómo se utilizan en *Doctrine*, consulta la sección *Objetos delegados* más adelante.

Controlador de metadatos (*REQUERIDO*)

```
<?php
$config->setMetadataDriverImpl($driver);
$config->getMetadataDriverImpl();
```

Obtiene o establece la implementación del controlador de metadatos utilizado por *Doctrine* para adquirir los metadatos objetorelacional para tus clases.

Actualmente hay 4 implementaciones disponibles:

- `Doctrine\ORM\Mapping\Driver\AnnotationDriver`
- `Doctrine\ORM\Mapping\Driver\XmlDriver`
- `Doctrine\ORM\Mapping\Driver\YamlDriver`
- `Doctrine\ORM\Mapping\Driver\DriverChain`

A lo largo de la mayor parte de este manual, utilizamos `AnnotationDriver` en los ejemplos. Para obtener información sobre el uso de `XmlDriver` o `YamlDriver` por favor consulta los capítulos dedicados a las asignaciones `XML` y `YAML`.

La anotación controlador se puede configurar con un método de fábrica en la opción `Doctrine\ORM\Configuration`:

```
<?php
$driverImpl = $config->newDefaultAnnotationDriver('/path/to/lib/MiProyecto/Entities');
$config->setMetadataDriverImpl($driverImpl);
```

La información de la ruta a las entidades es necesaria para la anotación controlador, porque de lo contrario las operaciones masivas en todas las entidades a través de la consola no funcionarían correctamente. Todos los controladores de metadatos aceptan tanto un solo directorio, como una cadena o una matriz de directorios. Con esta característica un solo controlador puede dar soporte a múltiples directorios de entidades.

Caché de metadatos (*RECOMENDADO*)

```
<?php
$config->setMetadataCacheImpl($cache);
$config->getMetadataCacheImpl();
```

Obtiene o establece la implementación de caché a utilizar para el almacenamiento en caché de la información de metadatos, es decir, toda la información suministrada a través de anotaciones, *XML* o *YAML*, por lo tanto no es necesario analizarla y cargarla a partir de cero en cada petición lo cual es una pérdida de recursos. La implementación de caché debe implementar la interfaz `Doctrine\Common\Cache\Cache`.

El uso de una caché de metadatos es muy recomendable.

Las implementaciones recomendadas para producción son:

- `Doctrine\Common\Cache\ApcCache`
- `Doctrine\Common\Cache\MemcacheCache`
- `Doctrine\Common\Cache\XcacheCache`

Para el desarrollo debes utilizar `Doctrine\Common\Cache\ArrayCache` la cual sólo almacena datos en base a cada petición.

Cache de consulta (*RECOMENDADA*)

```
<?php
$config->setQueryCacheImpl($cache);
$config->getQueryCacheImpl();
```

Obtiene o establece la implementación de caché para el almacenamiento en caché de las consultas `DQL`, es decir, el resultado de un proceso de análisis `DQL` que incluye al `SQL` final, así como la meta información acerca de cómo procesar el conjunto resultante de una consulta `SQL`. Ten en cuenta que la caché de consulta no afecta a los resultados de la consulta. No recibes datos obsoletos. Esta es una caché pura optimizada, sin efectos secundarios negativos (a excepción de algún uso mínimo de memoria en tu caché).

Usar una caché de consultas es muy recomendable.

Las implementaciones recomendadas para producción son:

- `Doctrine\Common\Cache\ApcCache`
- `Doctrine\Common\Cache\MemcacheCache`
- `Doctrine\Common\Cache\XcacheCache`

Para el desarrollo debes utilizar `Doctrine\Common\Cache\ArrayCache` la cual sólo almacena datos en base a cada petición.

Registro `SQL` (*Opcional*)

```
<?php
$config->setSQLLogger($logger);
$config->getSQLLogger();
```

Obtiene o establece el registrador a utilizar para registrar todas las declaraciones SQL ejecutadas por *Doctrine*. La clase del registrador debe implementar la interfaz Doctrine\DBAL\Logging\SQLLogger. Puedes encontrar una sencilla implementación predeterminada que registra la salida estándar utilizando echo y var_dump en Doctrine\DBAL\Logging\EchoSQLLogger.

Autogenerando clases delegadas (*OPCIONAL*)

```
<?php
$config->setAutoGenerateProxyClasses($bool);
$config->getAutoGenerateProxyClasses();
```

Obtiene o establece si las clases delegadas las debe generar *Doctrine* automáticamente en tiempo de ejecución. Si lo defines como falso, las clases delegadas se deben generar manualmente a través de la tarea generate-proxies de la línea de ordenes de *Doctrine*. El valor extremadamente recomendable para un entorno de producción es FALSO.

1.3.3 Configuración de Desarrollo frente a producción

Debes codificar tu arrancador de Doctrine2 con dos modelos de tiempo de ejecución diferentes en mente. Hay algunas serias ventajas al utilizar en producción APC o Memcache. Sin embargo en desarrollo esta frecuentemente dará errores fatales, al cambiar las entidades la caché aún conserva los metadatos obsoletos. Es por eso que recomendamos ArrayCache para el desarrollo.

Además debes tener a true la opción de autogeneración de clases delegadas en el desarrollo y a falso en producción. Si esta opción está establecida en true puedes dañar gravemente el rendimiento de tu archivo si varias clases delegadas se regeneran durante la ejecución del guión. Las llamadas de esa magnitud al sistema de archivos pueden incluso desacelerar todas las consultas problemáticas de base de datos de *Doctrine*. Además la escritura de un delegado establece un bloqueo de archivo exclusivo que puede causar serios cuellos de botella en el rendimiento en sistemas con peticiones simultáneas regulares.

1.3.4 Opciones de conexión

La $ConnectionOptions pasada como primer argumento de EntityManager::create() tiene que ser una matriz o una instancia de Doctrine\DBAL\Connection. Si pasas un arreglo es pasado directamente a la fábrica a lo largo de DBAL Doctrine\DBAL\DriverManager::getConnection(). La configuración DBAL se explica en la sección DBAL.

1.3.5 Objectos delegados

Un objeto delegado es un objeto que se coloca o se usa en lugar del objeto "real". Un objeto delegado puede añadir comportamiento al objeto sustituido sin que el objeto sea consciente de ello. En *Doctrine* 2, los objetos delegados se utilizan para realizar varias funciones, pero principalmente para la carga diferida transparente.

Los objetos delegados con sus facilidades de carga diferida ayudan a mantener conectado el subconjunto de objetos que ya está en memoria con el resto de los objetos. Esta es una propiedad esencial ya que sin ella no siempre serían frágiles objetos parciales en los bordes exteriores del gráfico de tus objetos.

Doctrine 2 implementa una variante del patrón delegado donde se generan las clases que extienden tus clases entidad y les agregan la capacidad de carga diferida. *Doctrine* entonces te puede dar una instancia de una clase delegada cada vez que solicitas un objeto de la clase que se está delegando. Esto ocurre en dos situaciones:

Refiriendo delegados

El método `EntityManager#getReference($nombreEntidad, $identificador)` te permite obtener una referencia a una entidad para la cual se conoce el identificador, sin tener que cargar esa entidad desde la base de datos. Esto es útil, por ejemplo, como una mejora de rendimiento, cuando deseas establecer una asociación a una entidad para la cual tienes el mismo identificador. Sólo tienes que hacer lo siguiente:

```
<?php
// $em instancia de EntityManager, $cart instancia de MiProyecto\Model\Cart
// $itemId comes from somewhere, probably a request parameter
$item = $em->getReference('MiProyecto\Model\Item', $itemId);
$cart->addItem($item);
```

Aquí, hemos añadido un elemento a un Carro sin tener que cargar el elemento desde la base de datos. Si invocas un método en la instancia del elemento, esta debe iniciar plenamente su estado de forma transparente desde la base de datos. Aquí `$item` realmente es una instancia de la clase delegada que se ha generado para la clase `Item`, pero el código no tiene por qué preocuparte. De hecho, **no debería preocuparte**. Los objetos delegados deben ser transparentes en tu código.

Asociando delegados

La segunda situación más importante en *Doctrine* utiliza objetos delegados al consultar objetos. Cada vez que consultas por un objeto que tiene una asociación de un solo valor a otro objeto que está configurado como diferido, sin unirlo a esa asociación en la misma consulta, *Doctrine* pone objetos delegados en el lugar donde normalmente estaría el objeto asociado. Al igual que otros delegados este se inicia transparentemente en el primer acceso.

Nota: Unir una asociación en una consulta `DQL` o nativa significa, esencialmente, cargar ansiosamente la asociación de esa consulta. Esto redefinirá la opción *'fetch'* especificada en la asignación de esa asociación, pero sólo para esa consulta.

Generando clases delegadas

Puedes generar clases delegadas manualmente a través de la consola de *Doctrine* o automáticamente por *Doctrine*. La opción de configuración que controla este comportamiento es:

```
<?php
$config->setAutoGenerateProxyClasses($bool);
$config->getAutoGenerateProxyClasses();
```

El valor predeterminado es `TRUE` para el desarrollo conveniente. Sin embargo, esta configuración no es óptima para el rendimiento y por lo tanto no se recomienda para un entorno de producción. Para eliminar la sobrecarga de la generación de clases delegadas en tiempo de ejecución, establece esta opción de configuración a `FALSE`. Al hacerlo en un entorno de desarrollo, ten en cuenta que es posible obtener errores de clase/archivo no encontrado si ciertas clases delegadas no están disponibles o no se cargan de manera diferida si has añadido nuevos métodos a la clase entidad que aún no están en la clase delegada. En tal caso, basta con utilizar la Consola de *Doctrine* para (re)generar las clases delegadas de la siguiente manera:

```
$ ./doctrine orm:generate-proxies
```

1.3.6 Múltiples fuentes de metadatos

Cuando utilizas diferentes componentes de *Doctrine* 2 te puedes encontrar con que ellos utilizan dos diferentes controladores de metadatos, por ejemplo XML y YAML. Puedes utilizar las implementaciones de metadatos *DriverChain*

agregando estos controladores basados en el espacio de nombres:

```
<?php
$chain = new DriverChain();
$chain->addDriver($xmlDriver, 'Doctrine\Tests\Models\Company');
$chain->addDriver($yamlDriver, 'Doctrine\Tests\ORM\Mapping');
```

Basándote en el espacio de nombres de la entidad la carga de las entidades se delega al controlador apropiado. La cadena semántica proviene del hecho de que el controlador recorre todos los espacios de nombres y coincide el nombre de la clase entidad contra el espacio de nombres con una llamada a `strpos() === 0`. Esto significa que necesitas ordenar los controladores correctamente si los subespacios de nombres utilizan diferentes implementaciones de metadatos.

1.4 Preguntas más frecuentes

Nota: Estas PF (Preguntas frecuentes) son un trabajo en progreso. Vamos a añadir un montón de preguntas y no las responderemos de inmediato para no olvidarnos lo que se pregunta a menudo. Si tropiezas con una pregunta irresoluta por favor escribe un correo a la lista de correo o entra al canal #doctrine en Freenode IRC.

1.4.1 Esquema de base de datos

¿Cómo puedo configurar el juego de caracteres y colación de las tablas *MySQL*?

No puedes establecer estos valores dentro de anotaciones, archivos de asignación *yml* o *XML*. Para hacer que trabaje una base de datos con el juego de caracteres y colación predeterminados debes configurar *MySQL* para usarlo como juego de caracteres predeterminado, o crear la base de datos con el conjunto de caracteres y detalles de colación. De esta manera se heredará en todas las tablas y columnas de la base de datos recién creada.

1.4.2 Clases entidad

Puedo acceder a una variable y es nula, ¿que está mal?

Si esta variable es una variable pública, entonces estás violando uno de los criterios para las entidades. Todas las propiedades tienen que ser protegidas o privadas para que trabaje el patrón de objetos delegados.

¿Cómo puedo añadir valores predeterminados a una columna?

Doctrine no cuenta con apoyo para establecer valores predeterminados a las columnas a través de la palabra clave "*DEFAULT*" en `SQL`. Sin embargo, esto no es necesario, puedes utilizar las propiedades de clase como valores predeterminados. Estas se utilizan al insertar:

```
class User
{
    const STATUS_DISABLED = 0;
    const STATUS_ENABLED = 1;

    private $algorithm = "sha1";
    private $status = self:STATUS_DISABLED;
}
```

.

1.4.3 Asignando

¿Por qué obtengo excepciones sobre restricción de fallo único en `$em->flush()`?

Doctrine no comprueba si estás volviendo a agregar entidades con una clave primaria que ya existe o añadiendo entidades a una colección en dos ocasiones. Tienes que comprobar tú mismo las condiciones en el código antes de llamar a `$em->flush()` si sabes que puede ocurrir un fallo en la restricción `UNIQUE`.

In Symfony 2 for example there is a Unique Entity Validator to achieve this task.

Para colecciones puedes comprobar si una entidad ya es parte de esa colección con `$collection->contains($entidad)`. Para una colección `FETCH=LAZY` esto inicia la colección, sin embargo, para `FETCH=EXTRA_LAZY` este método utiliza `SQL` para determinar si esta entidad ya es parte de la colección.

1.4.4 Asociaciones

¿Qué está mal cuando me sale un *InvalidArgumentException* "Se encontró una nueva entidad en la relación..."?

Esta excepción se produce durante el `EntityManager#flush()` cuando existe un objeto en el mapa de identidad que contiene una referencia a un objeto del cual *Doctrine* no sabe nada. Digamos por ejemplo que tomas una entidad "Usuario" desde la base de datos con un `id` específico y estableces un objeto completamente nuevo en una de las asociaciones del objeto Usuario. Si a continuación, llamas a `EntityManager#flush()` sin dejar que *Doctrine* sepa acerca de este nuevo objeto con `EntityManager#persist($nuevoObjeto)` verás esta excepción.

Puedes resolver esta excepción:

- Llamando a `EntityManager#persist($nuevoObjeto)` en el nuevo objeto
- Usar `cascade=persist` en la asociación que contiene el nuevo objeto

¿Cómo puedo filtrar una asociación?

Nativamente no puedes filtrar asociaciones en 2.0 y 2.1. Debes utilizar las consultas `DQL` para conseguir filtrar el conjunto de entidades.

Llamo a `clear()` en una colección Uno-A-Muchos, pero no se eliminan las entidades

Este es un comportamiento que tiene que ver con el lado inverso/propietario del controlador de *Doctrine*. Por definición, una asociación uno-a-muchos está en el lado inverso, es decir los cambios a la misma no serán reconocidos por *Doctrine*.

Si deseas realizar el equivalente de la operación de borrado tienes que recorrer la colección y fijar a `NULL` la referencia a la parte propietaria de muchos-a-uno para separar todas las entidades de la colección. Esto lanzará las declaraciones `UPDATE` adecuadas en la base de datos.

¿Cómo puedo agregar columnas a una tabla con relación Muchos-A-Muchos?

La asociación muchos-a-muchos sólo es compatible con claves externas en la definición de la tabla, para trabajar con tablas muchos-a-muchos que contienen columnas adicionales, tienes que utilizar la característica de claves externas como clave primaria de *Doctrine* introducida en la versión 2.1.

Ve la *guía de claves primarias compuestas* para más información.

¿Cómo puedo paginar colecciones recuperadas desde uniones?

Si estás emitiendo una declaración `DQL` que recupera una colección, tampoco es fácil iterar sobre la colección usando una declaración `LIMIT` (o equivalente del proveedor).

Doctrine no ofrece una solución para esto fuera de la caja, pero hay varias extensiones que lo hacen:

- Extensiones de Doctrine
- Pagerfanta

¿Por qué no funciona correctamente la paginación en uniones recuperadas?

La paginación en *Doctrine* utiliza una cláusula `LIMIT` (o su equivalente del proveedor) para restringir los resultados. Sin embargo, al recuperar uniones estas no devuelven el número de resultados correcto debido a que se unió con una asociación uno-a-muchos o muchos-a-muchos y esta multiplica el número de filas por el número de entidades asociadas.

Ve la pregunta anterior para una solución a esta tarea.

1.4.5 Herencia

¿Puedo utilizar herencia con *Doctrine* 2?

Si, en *Doctrine* 2, puedes utilizar herencia simple o úniendo tablas.

Ve la documentación en el capítulo sobre la :doc:asignación de herencia para más detalles.

¿Por qué *Doctrine* no crea objetos delegados de la jerarquía de mi herencia?

Si estableces una asociación muchos-a-uno o uno-a-uno entre entidad y destino a cualquier clase padre de una herencia jerárquica *Doctrine* en realidad no sabe qué clase PHP es la externa. Para averiguarlo tienes que ejecutar una consulta `SQL` para buscar esta información en la base de datos.

1.4.6 `EntityGenerator`

¿Por qué el `EntityGenerator` no hace X?

The EntityGenerator is not a full fledged code-generator that solves all tasks. Code-Generation is not a first-class priority in Doctrine 2 anymore (compared to Doctrine 1). The EntityGenerator is supposed to kick-start you, but not towards 100 %.

¿Por qué el `EntityGenerator` no genera la herencia correctamente?

Just from the details of the discriminator map the EntityGenerator cannot guess the inheritance hierachy. This is why the generation of inherited entities does not fully work. You have to adjust some additional code to get this one working correctly.

1.4.7 Rendimiento

¿Por qué se ejecuta una consulta `SQL` extra cada vez que recupero una entidad con una relación Uno-A-Uno?

Si *Doctrine* detecta que estás recuperando un lado inverso de una asociación uno-a-uno tiene que ejecutar una consulta adicional para cargar este objeto, porque no puede saber si no hay tal objeto (valor nulo) o si debe configurar un delegado cuyo `id` sea este delegado.

Para resolver este problema actualmente tienes que ejecutar una consulta para buscar esta información.

1.4.8 Lenguaje de consulta *Doctrine*

¿Qué es `DQL`?

DQL (por Doctrine Query Language) es el lenguaje de consultas de *Doctrine*, un lenguaje de consultas que se parece mucho a `SQL`, pero tienes algunas ventajas importantes cuando utilizas *Doctrine*:

- It uses class names and fields instead of tables and columns, separating concerns between backend and your object model.
- It utilizes the metadata defined to offer a range of shortcuts when writing. Por ejemplo, no es necesario que especifiques la cláusula `ON` de las uniones, ya que *Doctrine* ya las conoce.
- It adds some functionality that is related to object management and transforms them into SQL.

También tiene algunos inconvenientes, por supuesto:

- The syntax is slightly different to SQL so you have to learn and remember the differences.
- To be vendor independent it can only implement a subset of all the existing SQL dialects. La funcionalidad específica y optimización del proveedor no se pueden utilizar a través de `DQL` a menos que la implementes explícitamente.
- For some DQL constructs subselects are used which are known to be slow in MySQL.

¿Puedo ordenar por una función (por ejemplo *ORDEN BY RAND()*) en `DQL`?

No, `DQL` no es compatible con la ordenación en funciones. Si necesitas esta funcionalidad debes usar una consulta nativa o pensar en otra solución. Como nota al margen: La ordenación con *ORDER BY RAND()* es muy lenta a partir de 1000 filas.

1.5 Asignación básica

En este capítulo se explica la asignación básica de objetos y propiedades. La asignación de las asociaciones se tratará en el próximo capítulo "*Asignando asociaciones*".

1.5.1 Controladores de asignación

Doctrine proporciona varias maneras diferentes para especificar la asignación de metadatos objetorelacional:

- Anotaciones *Docblock*
- XML
- YAML

En este manual solemos hacer mención de las anotaciones en los bloques de documentación en todos los ejemplos dispersos a lo largo de todos los capítulos, sin embargo, también se dan ejemplos alternativos para YAML y XML. Hay capítulos de referencia dedicados para la asignación XML y YAML, respectivamente, que los explican con más detalle. También hay un capítulo de referencia para las anotaciones.

Nota: Si te estás preguntando qué controlador de asignación ofrece el mejor rendimiento, la respuesta es: todos dan exactamente el mismo rendimiento. Una vez que has leído los metadatos de una clase desde la fuente (anotaciones, XML o *YAML*) se almacenan en una instancia de la clase `Doctrine\ORM\Mapping\ClassMetadata` y sus instancias se almacenan en la memoria caché de metadatos. Por lo tanto al final del día todos los controladores lo realizan igual de bien. Si no estás utilizando una memoria caché de metadatos (¡no se recomienda!), entonces el controlador XML podría tener una ligera ventaja en el rendimiento debido al potente soporte nativo XML de *PHP*.

1.5.2 Introducción a las anotaciones *Docblock*

Probablemente ya hayas utilizado las anotaciones *Docblock* de alguna forma, muy probablemente proporcionando documentación de metadatos a una herramienta como `PHPDocumentor` (*@author, @link*, ...). Las anotaciones *docblock* son una herramienta para incrustar metadatos dentro de la sección de documentación que luego, alguna herramienta puede procesar. *Doctrine* 2 generaliza el concepto de anotaciones *docblock* para que se puedan utilizar en cualquier tipo de metadatos y así facilitar la definición de nuevas anotaciones *docblock*. A fin de implicar mas los valores de anotación y para reducir las posibilidades de enfrentamiento con las anotaciones de *docblock*, las anotaciones *docblock* de *Doctrine* 2 cuentan con una sintaxis alternativa que está fuertemente inspirada en la sintaxis de las anotaciones introducidas en Java 5.

La implementación de estas anotaciones *docblock* mejoradas se encuentran en el espacio de nombres `Doctrine\Common\Annotations` y por lo tanto, son parte del paquete Común. Las anotaciones *docblock* de *Doctrine* 2 cuentan con apoyo para espacios de nombres y anotaciones anidadas, entre otras cosas. El ORM de *Doctrine* 2 define su propio conjunto de anotaciones *docblock* para suministrar asignación de metadatos objetorelacional.

Nota: Si no te sientes cómodo con el concepto de las anotaciones *docblock*, no te preocupes, como se mencionó anteriormente *Doctrine* 2 ofrece alternativas XML y YAML y fácilmente podrías implementar tu propio mecanismo preferido para la definición de los metadatos del ORM.

1.5.3 Clases persistentes

Con el fin de marcar una clase para persistencia objetorelacional la debes designar como una entidad. Esto se puede hacer a través de la anotación `@Entity`.

- *PHP*

```
<?php
/** @Entity */
class MyPersistentClass
{
```

```
    //...
}
```

- *XML*

```
<doctrine-mapping>
  <entity name="MyPersistentClass">
      <!-- ... -->
  </entity>
</doctrine-mapping>
```

- *YAML*

```
MyPersistentClass:
  type: entity
  # ...
```

De manera predeterminada, la entidad será persistida en una tabla con el mismo nombre que el nombre de la clase. A fin de cambiar esta situación, puedes utilizar la anotación `@Table` de la siguiente manera:

- *PHP*

```
<?php
/**
 * @Entity
 * @Table(name="my_persistent_class")
 */
class MyPersistentClass
{
    //...
}
```

- *XML*

```
<doctrine-mapping>
  <entity name="MyPersistentClass" table="my_persistent_class">
      <!-- ... -->
  </entity>
</doctrine-mapping>
```

- *YAML*

```
MyPersistentClass:
  type: entity
  table: my_persistent_class
  # ...
```

Ahora las instancias de *MyPersistentClass* serán persistidas en una tabla llamada `my_persistent_class`.

1.5.4 Tipos de asignación de *Doctrine*

El tipo de asignación de *Doctrine* define la asignación entre un tipo de *PHP* y un tipo de `SQL`. Todos los tipos de asignación de *Doctrine* incluidos en *Doctrine* son completamente portátiles entre los diferentes *RDBMS*. Incluso, puedes escribir tus propios tipos de asignación personalizados que pueden o no ser portátiles, esto se explica más adelante en este capítulo.

Por ejemplo, el tipo de asignación `string` de *Doctrine* define la asignación de una cadena de *PHP* a un *VARCHAR SQL* (o *VARCHAR2*, etc. dependiendo del *RDBMS*). Aquí está una descripción rápida de los tipos de asignación integrados:

- `string`: Tipo que asigna un *VARCHAR SQL* a una cadena PHP.
- `integer`: Tipo que asigna un *INT SQL* a un entero de *PHP*.
- `smallint`: Tipo que asigna un *SMALLINT* de la base de datos a un entero de *PHP*.
- `bigint`: Tipo que asigna un *BIGINT* de la base de datos a una cadena de *PHP*.
- `boolean`: Tipo que asigna un valor *booleano SQL* a un *booleano* de *PHP*.
- `decimal`: Tipo que asigna un *DECIMAL SQL* a un doble de *PHP*.
- `date`: Tipo que asigna un *DATETIME* de `SQL` a un objeto *DateTime* de *PHP*.
- `time`: Tipo que asigna un *TIME* de *SQL* a un objeto *DateTime* de *PHP*.
- `datetime`: Tipo que asigna un *DATETIME/TIMESTAMP SQL* a un objeto *DateTime* de *PHP*.
- `text`: Tipo que asigna un *CLOB SQL* a una cadena de *PHP*.
- `object`: Tipo que asigna un *CLOB SQL* a un objeto de *PHP* usando `serialize()` y `unserialize()`
- `array`: Tipo que asigna un *CLOB SQL* a un objeto de *PHP* usando `serialize()` y `unserialize()`
- `float`: Tipo que se asigna un *Float SQL* (de doble precisión) en un *double* de *PHP*. *IMPORTANTE*: Sólo funciona con opciones regionales que utilizan punto decimal como separador.

Nota: ¡Tipos de asignación de *Doctrine* NO son tipos `SQL` y NO son tipos de *PHP*! Son tipos de asignación entre 2 tipos. Además los tipos de asignación *susceptibles a mayúsculas y minúsculas*. Por ejemplo, al utilizar una columna *DateTime* no coincidirá con el tipo *datetime* suministrado con *Doctrine* 2.

Nota: DateTime and Object types are compared by reference, not by value. Doctrine updates this values if the reference changes and therefore behaves as if these objects are immutable value objects.

Advertencia: Todos los tipos de fecha asumen que se está utilizando exclusivamente la zona horaria establecida por date_default_timezone_set() o por la configuración `date.timezone` de *php.ini*. Trabajar con diferentes zonas horarias causará problemas y un comportamiento inesperado.
Si necesitas manejar zonas horarias específicas que tienen que manejar esto en tu dominio, convierte todos los valores a UTC de ida y vuelta. También hay un *artículo en el recetario* sobre el trabajo con fechas y horas que proporciona consejos para implementar aplicaciones de múltiples zonas horarias.

1.5.5 Asignando propiedades

Después de marcar una clase como una entidad esta puede especificar las asignaciones para los campos de su instancia. Aquí sólo nos fijaremos en los campos simples que tienen valores escalares como cadenas, números, etc. Las asociaciones con otros objetos se tratan en el capítulo *Asignando asociaciones*.

Para marcar una propiedad para persistencia relacional se utiliza la anotación *docblock* `@Column`. Esta anotación generalmente requiere fijar por lo menos un atributo, el `type`. El atributo `type` especifica el tipo de asignación que *Doctrine* usará para el campo. Si no se especifica el tipo, se usa `string` como el tipo de asignación por omisión, ya que es el más flexible.

Ejemplo:

- *PHP*

```php
<?php
/** @Entity */
class MyPersistentClass
{
    /** @Column(type="integer") */
    private $id;
    /** @Column(length=50) */
    private $nombre; // type defaults to string
    //...
}
```

- *XML*

```xml
<doctrine-mapping>
  <entity name="MyPersistentClass">
    <field name="id" type="integer" />
    <field name="name" length="50" />
  </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
MyPersistentClass:
  type: entity
  fields:
    id:
      type: integer
    name:
      length: 50
```

En este ejemplo asignamos el campo `id` a la columna `id` utilizando el tipo de asignación `integer` y asignamos el campo `nombre` a la columna `nombre` con la asignación de tipo `string` predeterminada. Como puedes ver, por omisión los nombres de columna se supone que son los mismos que los nombres de campo. Para especificar un nombre diferente para la columna, puedes utilizar el atributo `name` de la anotación `@Column` de la siguiente manera:

- *PHP*

```php
<?php
/** @Column(name="db_name") */
private $nombre;
```

- *XML*

```xml
<doctrine-mapping>
  <entity name="MyPersistentClass">
    <field name="name" column="db_name" />
  </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
MyPersistentClass:
  type: entity
  fields:
    name:
      length: 50
      column: db_name
```

La anotación `@Column` tiene unos pocos atributos más. Aquí está una lista completa:

- `type`: (opcional, predeterminado a 'string') La asignación de tipo a usar para la columna.

- `name`: (opcional, predeterminado al nombre del campo) El nombre de la columna en la base de datos.
- `length`: (opcional, predeterminado a 255) La longitud de la columna en la base de datos. (Sólo se aplica si se usa un valor de cadena en la columna).
- `unique`: (opcional, predeterminado a `FALSE`) Si la columna es una clave única.
- `nullable`: (opcional, predeterminado a `FALSE`) Si la columna de la base de datos acepta valores nulos.
- `precision`: (opcional, predeterminado a 0) La precisión para una columna decimal (numérica exacta). (Sólo aplica si se usa una columna decimal.)
- `scale`: (opcional, predeterminado a 0) La escala para una columna decimal (numérica exacta). (Sólo aplica si se usa una columna decimal.)

1.5.6 Tipos de asignación personalizados

Doctrine te permite crear nuevos tipos de asignación. Esto puede ser útil cuando estás perdiendo un tipo de asignación específico o cuando deseas reemplazar la aplicación actual de un tipo de asignación.

A fin de crear un nuevo tipo de asignación necesitas una subclase de `Doctrine\DBAL\Types\Type` e implementar/redefinir los métodos a tu gusto. Aquí está un esqueleto de ejemplo de una clase de tipo personalizada:

```
<?php
namespace My\Project\Types;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

/**
 * My custom datatype.
 */
class MyType extends Type
{
    const MITIPO = 'mytype'; // modifícala para que coincida con el nombre de tu tipo

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        // devuelve la SQL usada para crear tu tipo de columna. Para crear un tipo de columna portáti
    }

    public function convertToPHPValue($valor, AbstractPlatform $platform)
    {
        // Esto se ejecuta cuando el valor es leído de la base de datos. Haz tus conversiones aquí, c
    }

    public function convertToDatabaseValue($valor, AbstractPlatform $platform)
    {
        // Esto se ejecuta cuando se escribe el valor en la base de datos. Haz tus conversiones aquí,
    }

    public function getName()
    {
        return self::MITIPO; // modifica para que coincida con el nombre de tu constante
    }
}
```

Restricciones a tener en cuenta:

- Si el valor de un campo es *NULL* no se llama al método `convertToDatabaseValue()`.

- La `UnitOfWork` nunca pasa los valores de la base de datos que no han cambiado al método convertir.

Cuando has implementado el tipo todavía tienes que notificar su existencia a *Doctrine*. Esto se puede lograr a través del método `Doctrine\DBAL\Types\Type#addType($nombre, $className)`. Ve el siguiente ejemplo:

```
<?php
// en el código de arranque

// ...

use Doctrine\DBAL\Types\Type;

// ...

// Registra mi tipo
Type::addType('mytype', 'My\Project\Types\MyType');
```

Como puedes ver arriba, al registrar el tipo personalizado en la configuración, especificas un nombre único para el tipo de asignación y lo asignas el nombre completamente calificado de la clase correspondiente. Ahora puedes utilizar el nuevo tipo en la asignación de esta manera:

```
<?php
class MyPersistentClass
{
    /** @Column(type="mytype") */
    private $field;
}
```

Para conseguir que la herramienta de esquema convierta el tipo de la base de datos subyacente a tu nuevo "mytype" directamente en una instancia de `MyType` tienes que registrar, además, esta asignación en tu plataforma de base de datos:

```
<?php
$conn = $em->getConnection();
$conn->getDatabasePlatform()->registerDoctrineTypeMapping('db_mytype', 'mytype');
```

Ahora, utilizando la herramienta de esquema, siempre que detecte una columna que tiene el `db_mytype` la convertirá en una instancia del tipo `mytype` de *Doctrine* para representar el esquema. Ten en cuenta que fácilmente puedes producir choques de esta manera, cada tipo de la base de datos sólo puede asignar exactamente a un tipo de asignación *Doctrine*.

1.5.7 Personalizando *ColumnDefinition*

Puedes especificar una definición personalizada de cada columna usando el atributo "*ColumnDefinition*" de `@Column`. Aquí, tienes que especificar todas las definiciones que siguen al nombre de una columna.

Nota: Al utilizar "*ColumnDefinition*" se romperá el cambio de detección en *SchemaTool*.

1.5.8 Identificadores / claves primarias

Cada clase entidad necesita un identificador / clave primaria. Para designar el campo que sirve como identificador lo marcas con la anotación `@Id`. Aquí está un ejemplo:

- *PHP*

```php
<?php
class MyPersistentClass
{
    /** @Id @Column(type="integer") */
    private $id;
    //...
}
```

- *XML*

```xml
<doctrine-mapping>
  <entity name="MyPersistentClass">
    <id name="id" type="integer" />
    <field name="name" length="50" />
  </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
MyPersistentClass:
  type: entity
  id:
    id:
      type: integer
  fields:
    name:
      length: 50
```

Sin hacer nada más, el identificador se supone que es asignado manualmente. Esto significa que tu código necesita configurar correctamente la propiedad identificador antes de pasar a una nueva entidad para `EntityManager#persist($entity)`.

Una estrategia alternativa común es usar como identificador un valor generado. Para ello, utilizas la anotación `@GeneratedValue` de esta manera:

- *PHP*

```php
<?php
class MyPersistentClass
{
    /**
     * @Id @Column(type="integer")
     * @GeneratedValue
     */
    private $id;
}
```

- *XML*

```xml
<doctrine-mapping>
  <entity name="MyPersistentClass">
    <id name="id" type="integer">
        <generator strategy="AUTO" />
    </id>
    <field name="name" length="50" />
  </entity>
</doctrine-mapping>
```

- *YAML*

```
MyPersistentClass:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      length: 50
```

Esto le dice a *Doctrine* que genere automáticamente un valor para el identificador. Cómo se genera este valor se especifica mediante la estrategia de `atributo`, que es opcional y por omisión es `AUTO`. Un valor de `AUTO` dice a *Doctrine* que utilice la estrategia de generación que es preferida por la plataforma de base de datos utilizada actualmente. Ve más adelante para obtener más detalles.

Estrategias para la generación de identificador

El ejemplo anterior muestra cómo utilizar la estrategia de generación de identificador predeterminada sin conocer la base de datos subyacente con la estrategia de detección automática. También es posible especificar la estrategia de generación de identificador de manera más explícita, lo cual te permite usar algunas características adicionales.

Aquí está la lista de posibles estrategias de generación:

- `AUTO` (predeterminada): Le dice a *Doctrine* que escoja la estrategia preferida por la plataforma de base de datos. Las estrategias preferidas son la IDENTIDAD de MySQL, SQLite y MsSQL y las SECUENCIA de Oracle y PostgreSQL. Esta estrategia ofrece una portabilidad completa.
- `SEQUENCE`: Le dice a *Doctrine* que use una secuencia de la base de datos para gestionar el *ID*. Esta estrategia actualmente no proporciona portabilidad total. Las secuencias son compatibles con Oracle y PostgreSQL.
- `IDENTITY`: Le dice a *Doctrine* que use una columna de identidad especial en la base de datos que genere un valor o inserte una fila. Esta estrategia actualmente no proporciona una total portabilidad y es compatible con las siguientes plataformas: MySQL/SQLite (*AUTO_INCREMENT*), MSSQL (*IDENTIDAD*) y PostgreSQL (*SERIAL*).
- `TABLE`: Le dice a *Doctrine* que use una tabla independiente para la generación del *ID*. Esta estrategia ofrece una portabilidad completa. ***Esta estrategia aún no está implementada***
- `NONE`: Indica a *Doctrine* que los identificadores son asignados (y por lo tanto generados) por tu código. La asignación tendrá lugar antes de pasar una nueva entidad a `EntityManager#persist`. `NONE` es lo mismo que dejar fuera a `@GeneratedValue` por completo.

Generador de secuencia

El generador de secuencias actualmente se puede utilizar en conjunto con Oracle o Postgres y permite algunas opciones de configuración adicionales, además de especificar el nombre de la secuencia:

- *PHP*

```php
<?php
class User
{
    /**
     * @Id
     * @GeneratedValue(strategy="SEQUENCE")
     * @SequenceGenerator(name="tablename_seq", initialValue=1, allocationSize=100)
```

```
     */
    protected $id = null;
}
```

- *XML*

```
<doctrine-mapping>
  <entity name="User">
    <id name="id" type="integer">
        <generator strategy="SEQUENCE" />
        <sequence-generator sequence-name="tablename_seq" allocation-size="100" initial-value="1
    </id>
  </entity>
</doctrine-mapping>
```

- *YAML*

```
MyPersistentClass:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: SEQUENCE
      sequenceGenerator:
        sequenceName: tablename_seq
        allocationSize: 100
        initialValue: 1
```

El valor inicial especifica en qué valor debe empezar la secuencia.

`allocationSize` es una potente característica para optimizar el rendimiento de `INSERT` de *Doctrine*. `allocationSize` especifica en qué valores se incrementa la secuencia cada vez que se recupera el valor siguiente. Si este es mayor que 1 (uno) *Doctrine* puede generar valores de identificador por la cantidad `allocationSizes` de las entidades. En el ejemplo anterior con `allocationSize=100` *Doctrine* 2 sólo tendrá que acceder a la secuencia una vez para generar los identificadores de 100 nuevas entidades.

El `allocationSize` *predeterminado para un* `@SequenceGenerator` *actualmente es de 10.*

Prudencia: El `allocationSize` es detectado por `SchemaTool` y transformado en una cláusula "*incremento por*" de la declaración `CREATE SEQUENCE`. Para un esquema de base de datos creado manualmente (y no por `SchemaTool`), tienes que asegurarte de que la opción de configuración `allocationSize` nunca es mayor que el valor de incremento real de las secuencias, de lo contrario puedes obtener claves duplicadas.

Nota: Es posible utilizar `strategy="AUTO"` y, al mismo tiempo, especificar un `@SequenceGenerator`. En tal caso, tu configuración de secuencias personalizada se utilizará en caso de que la estrategia preferida de la plataforma subyacente sea la secuencia, por ejemplo, para Oracle y PostgreSQL.

Claves compuestas

Doctrine 2 te permite usar claves primarias compuestas. Sin embargo, hay algunas restricciones en lugar de utilizar un identificador único. El uso de la anotación `@GeneratedValue` sólo es compatible con claves primarias simples (no compuestas), lo cual significa que sólo puedes utilizar claves compuestas si generas los valores de las claves primarias antes de llamar a `EntityManager#persist()` en la entidad.

Para designar un identificador/clave primaria compuesta, simplemente coloca la anotación `@Id` en todos los campos que componen la clave primaria.

1.5.9 Citando palabras reservadas

Posiblemente en ocasiones necesites citar un nombre de columna o tabla porque está en conflicto con una palabra reservada para el uso del *RDBMS* particular. Esto se refiere a menudo como "*Citando identificadores*". Para permitir que *Doctrine* sepa como te gustaría un nombre de tabla o columna tienes que citarlo en todas las declaraciones `SQL`, escribe el nombre de la tabla o columna entre acentos abiertos. Aquí está un ejemplo:

```
<?php
/** @Column(name="`number`", type="integer") */
private $number;
```

Entonces *Doctrine* citará este nombre de columna en todas las declaraciones `SQL` de acuerdo con la plataforma de base de datos utilizada.

Advertencia: Al citar el identificador, este no se admite para unir nombres de columna o para el discriminador de nombres de columna.

Advertencia: El citado del identificador es una característica que se destina principalmente a apoyar esquemas de bases de datos heredadas. El uso de palabras reservadas y citado del identificador generalmente se desaconseja. El citado del identificador no se debe utilizar para permitir el uso de caracteres no estándar, como un guión de una hipotética columna `nombre-de-prueba`. Además la herramienta de esquema probablemente tenga problemas cuando se use el citado por razones de mayúsculas y minúsculas (en Oracle, por ejemplo).

1.6 Asignando asociaciones

Este capítulo explica cómo se asignan las asociaciones entre entidades con *Doctrine*. Comenzamos con una explicación del concepto de propiedad y lados inversos lo cual es muy importante entender cuando trabajas con asociaciones bidireccionales. Por favor, lee cuidadosamente estas explicaciones.

1.6.1 Lado propietario y lado inverso

Cuando asignas asociaciones bidireccionales, es importante entender el concepto del lado propietario y el lado inverso. Se aplican las siguientes reglas generales:

- Las relaciones pueden ser bidireccionales o unidireccionales.
- Una relación bidireccional tiene tanto un lado propietario como un lado inverso.
- Una relación unidireccional sólo tiene un lado propietario.
- El lado propietario de una relación determina las actualizaciones a la relación en la base de datos.

Las siguientes reglas se aplican a las asociaciones *bidireccionales*:

- El lado inverso de una relación bidireccional se debe referir a su lado propietario usando el atributo `mappedBy` de las declaraciones de asignación `OneToOne`, `OneToMany` o `ManyToMany`. El atributo `mappedBy` designa el campo en la entidad, que es el propietario de la relación.
- El lado propietario de una relación bidireccional se debe referir a su lado inverso usando el atributo `inversedBy` de la declaración de asignación `OneToOne`, `ManyToOne` o `ManyToMany`. El atributo `inversedBy` designa el campo en la entidad, que es el lado inverso de la relación.

- El lado muchos relaciones bidireccionales `OneToMany`/`ManyToOne` *debe* ser la parte propietaria, por lo tanto, el elemento `mappedBy` no se puede especificar en el lado `ManyToOne`.
- Para relaciones bidireccionales `OneToOne`, la parte propietaria concuerda a la parte que contiene la clave externa correspondiente (`@JoinColumn(s)`).
- Para relaciones bidireccionales `ManyToMany` cualquiera puede ser el lado propietario (la parte que define el `@JoinTable` y/o no usa el atributo `mappedBy`, utilizando así una línea predeterminada de unión de tabla).

Especialmente importante es la siguiente:

El lado propietario de una relación determina las actualizaciones a la relación en la base de datos.

Para comprender esto, recuerda cómo se mantienen las asociaciones bidireccionales en el mundo de los objetos. Hay dos referencias a cada lado de la asociación y las dos referencias de ambos representan la misma asociación, pero puede cambiar de forma independiente la una de la otra. Por supuesto, en una correcta aplicación de la semántica, las asociaciones bidireccionales son mantenidas adecuadamente por el desarrollador de la aplicación (porque es su responsabilidad). *Doctrine* tiene que saber cuál de estas dos referencias en memoria es la que se debe conservar y cuál no. Esto es para lo que se utiliza el concepto propietario/inverso principalmente.

**Los cambios realizados sólo en el lado inverso de la asociación son ignorados. Asegúrate de actualizar ambos lados de una asociación bidireccional (o al menos el lado propietario, desde el punto de vista de Doctrine) **

El lado propietario de una asociación bidireccional es el lado "en que busca" *Doctrine* a la hora de determinar el estado de la asociación, y por lo tanto si hay algo que hacer para actualizar la asociación en la base de datos.

Nota: El "lado propietario" y el "lado inverso" son conceptos técnicos de la tecnología ORM, no conceptos de tu modelo del dominio. Lo que consideras como el lado propietario en tu modelo del dominio puede ser diferente al lado propietario de *Doctrine*. Estos no están relacionados.

1.6.2 Colecciones

En todos los ejemplos de los muchos valores de las asociaciones en este manual usaremos una interfaz de `Colección` y una implementación predeterminada correspondiente `ArrayCollection` que se definen en el espacio de nombres `Doctrine\Common\Collections`. ¿Por qué necesitamos esto? ¿Qué no es la pareja de mi modelo del dominio para *Doctrine*? Desafortunadamente, las matrices de *PHP*, si bien son buenas para muchas cosas, no se constituyen buenas para colecciones de objetos del negocio, especialmente no en el contexto de un ORM. La razón es que los arreglos planos de *PHP* no se pueden ampliar/instrumentar transparentemente en el código *PHP*, lo cual es necesario para un montón de características avanzadas del ORM. Las clases/interfaces que se acercan más a una colección OO (Orientada a Objetos) son `ArrayAccess` y `ArrayObject` pero hasta que creas instancias de este tipo se pueden utilizar en todos los lugares donde se puede utilizar una matriz plana (algo que puede suceder en *PHP*6) su utilidad es bastante limitada. "Puedes" sugerir el tipo ''ArrayAccess'' en vez de la ''Colección'', ya que la interfaz de la Colección extiende a ''ArrayAccess'', pero esto te limitará severamente en la forma en que puedes trabajar con la Colección, ya que la *API* de `ArrayAccess` (intencionalmente) es muy primitiva y es más importante porque no puedes pasar a esta Colección todas las funciones útiles de los arreglos PHP, lo cual dificulta trabajar con él.

Advertencia: La interfaz de Colección y la clase `ArrayCollection`, como todo lo demás en el espacio de nombres de *Doctrine*, no son parte del ORM, ni de DBAL, se trata de una clase PHP simple que no tiene ninguna dependencia externa, aparte de las dependencias de *PHP* en sí mismo (y SPL). Por lo tanto, con esta clase en las clases de tu dominio y en otros lugares no introduce un acoplamiento a la capa de persistencia. La clase Colección, como todo lo demás en el espacio de nombres Común, no es parte de la capa de persistencia. Incluso puedes copiar la clase a tu proyecto si deseas eliminar *Doctrine* de tu proyecto y todas las clases del dominio funcionarán igual que antes.

1.6.3 Asignación predeterminada

Antes de presentar todas las asignaciones de asociaciones en detalle, debes tener en cuenta que las definiciones `@JoinColumn` y `@JoinTable` suelen ser opcionales y tienen valores predeterminados razonables. Los valores predeterminados para una columna en una unión de asociación uno-a-uno/muchos-a-uno son los siguientes:

```
name: "<fieldname>_id"
referencedColumnName: "id"
```

Como ejemplo, considera esta asignación:

- *PHP*

```php
<?php
/** @OneToOne(targetEntity="Shipping") */
private $shipping;
```

- *XML*

```xml
<doctrine-mapping>
    <entity class="Product">
        <one-to-one field="shipping" target-entity="Shipping" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
Product:
  type: entity
  oneToOne:
    shipping:
      targetEntity: Shipping
```

Esta esencialmente es la misma asignación que la siguiente, más detallada:

- *PHP*

```php
<?php
/**
 * @OneToOne(targetEntity="Shipping")
 * @JoinColumn(name="shipping_id", referencedColumnName="id")
 */
private $shipping;
```

- *XML*

```xml
<doctrine-mapping>
    <entity class="Product">
        <one-to-one field="shipping" target-entity="Shipping">
            <join-column name="shipping_id" referenced-column-name="id" />
        </one-to-one>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
Product:
  type: entity
  oneToOne:
    shipping:
      targetEntity: Shipping
```

```
        joinColumn:
          name: shipping_id
          referencedColumnName: id
```

La definición `@JoinTable` utilizada para asignaciones muchos-a-muchos tiene predeterminados similares. Como ejemplo, considera esta asignación:

- *PHP*

```php
<?php
class User
{
    //...
    /** @ManyToMany(targetEntity="Group") */
    private $groups;
    //...
}
```

- *XML*

```xml
<doctrine-mapping>
    <entity class="User">
        <many-to-many field="groups" target-entity="Group" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  manyToMany:
    groups:
      targetEntity: Group
```

Esta esencialmente es la misma asignación que la siguiente, más detallada:

- *PHP*

```php
<?php
class User
{
    //...
    /**
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="User_Group",
     *      joinColumns={@JoinColumn(name="User_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="Group_id", referencedColumnName="id")}
     *      )
     */
    private $groups;
    //...
}
```

- *XML*

```xml
<doctrine-mapping>
    <entity class="User">
        <many-to-many field="groups" target-entity="Group">
            <join-table name="User_Group">
                <join-columns>
                    <join-column id="User_id" referenced-column-name="id" />
```

```
                </join-columns>
                <inverse-join-columns>
                    <join-column id="Group_id" referenced-column-name="id" />
                </inverse-join-columns>
            </join-table>
        </many-to-many>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  manyToMany:
    groups:
      targetEntity: Group
      joinTable:
        name: User_Group
        joinColumns:
          User_id:
            referencedColumnName: id
        inverseJoinColumns:
          Group_id
            referencedColumnName: id
```

En ese caso, el nombre de la tabla predeterminada está unida a una combinación simple, del nombre de clase sin calificar de las clases participantes, separadas por un carácter de subrayado. Los nombres de las columnas de la unión por omisión al nombre simple, del nombre de la clase din calificar a la clase destino, seguido de `\_id`. El `referencedColumnName` de manera predeterminada siempre a `id`, al igual que en asignaciones uno-a-uno o muchos-a-uno.

Si aceptas estos valores predeterminados, puedes reducir al mínimo tu código de asignación.

1.6.4 Iniciando Colecciones

Tienes que tener cuidado al utilizar los campos de entidad que contienen una colección de entidades relacionadas. Digamos que tenemos una entidad Usuario que contiene una colección de Grupos:

```
<?php
/** @Entity */
class User
{
    /** @ManyToMany(targetEntity="Group") */
    private $groups;

    public function getGroups()
    {
        return $this->groups;
    }
}
```

Con este código solo el campo `$grupos` contiene una instancia de `Doctrine\Common\Collections\Collection` si los datos del usuario son recuperados desde *Doctrine*, sin embargo, **no** después de que generas nuevas instancias del Usuario. Cuando tu entidad de Usuario `$grupos` todavía es nueva, obviamente, será nula.

Por esta razón te recomendamos iniciar todos tus campos de colección a un `ArrayCollection` vacío en el constructor de tus entidades:

```php
<?php
use Doctrine\Common\Collections\ArrayCollection;

/** @Entity */
class User
{
    /** @ManyToMany(targetEntity="Group") */
    private $groups;

    public function __construct()
    {
        $this->groups = new ArrayCollection();
    }

    public function getGroups()
    {
        return $this->groups;
    }
}
```

Ahora el siguiente código debe trabajar incluso si la Entidad aún no se ha asociado con un `EntityManager`:

```php
<?php
$group = $entityManager->find('Group', $groupId);
$user = new User();
$user->getGroups()->add($group);
```

1.6.5 Validando la asignación en tiempo de ejecución frente a desarrollo

Por motivos de rendimiento *Doctrine* 2 tiene que omitir alguna validación necesaria en las asignaciones de la asociación. Tienes que ejecutar esta validación en tu flujo de trabajo para verificar que las asociaciones están correctamente definidas.

Puedes utilizar la herramienta de línea de ordenes de *Doctrine*:

```
doctrine orm:validate-schema
```

O puedes activar la validación manualmente:

```php
<?php
use Doctrine\ORM\Tools\SchemaValidator;

$validator = new SchemaValidator($entityManager);
$errors = $validator->validateMapping();

if (count($errors) > 0) {
    // ¡Un montón de errores!
    echo implode("\n\n", $errors);
}
```

Si la asignación no es válida, la matriz de errores contiene un número positivo de elementos con mensajes de error.

Advertencia: Una opción de asignación que no se valida es al usar el nombre de la columna referida. Tienes que apuntar a la clave primaria equivalente, de otra manera *Doctrine* no funcionará.

Nota: Un error común es usar una barra inversa al frente del nombre de clase totalmente calificado. Cada vez que se representa un FQCN (por fully-qualified class-name o nombre de clase totalmente cualificado) dentro de una cadena

(como en las definiciones de asignación), tienes que quitar la barra inversa prefija. PHP hace esto con `get_class()` o métodos de reflexión por razones de compatibilidad hacia atrás.

1.6.6 Uno-A-Uno, unidireccional

A asociación unidireccional uno-a-uno es muy común. He aquí un ejemplo de un `Producto` que tiene asociado un objeto‘‘Envío‘‘. El lado `Envío` no hace referencia de nuevo al `Producto` por lo tanto es unidireccional.

- *PHP*

```php
<?php
/** @Entity */
class Product
{
    // ...

    /**
     * @OneToOne(targetEntity="Shipping")
     * @JoinColumn(name="shipping_id", referencedColumnName="id")
     */
    private $shipping;

    // ...
}

/** @Entity */
class Shipping
{
    // ...
}
```

- *XML*

```xml
<doctrine-mapping>
    <entity class="Product">
        <one-to-one field="shipping" target-entity="Shipping">
            <join-column name="shipping_id" referenced-column-name="id" />
        </one-to-one>
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Product:
  type: entity
  oneToOne:
    shipping:
      targetEntity: Shipping
      joinColumn:
        name: shipping_id
        referencedColumnName: id
```

Ten en cuenta que la `@JoinColumn` realmente no es necesaria en este ejemplo, puesto que con los valores predeterminados sería lo mismo.

Esquema MySQL generado:

```
CREATE TABLE Product (
    id INT AUTO_INCREMENT NOT NULL,
    shipping_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Shipping (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Product ADD FOREIGN KEY (shipping_id) REFERENCES Shipping(id);
```

1.6.7 Uno-A-Uno, bidireccional

Aquí hay una relación uno-a-uno bidireccional entre un `Cliente` y un `Carrito`. El `Carrito` tiene una referencia al `Cliente` por lo tanto es bidireccional.

- *PHP*

```
<?php
/** @Entity */
class Customer
{
    // ...

    /**
     * @OneToOne(targetEntity="Cart", mappedBy="customer")
     */
    private $cart;

    // ...
}

/** @Entity */
class Cart
{
    // ...

    /**
     * @OneToOne(targetEntity="Customer", inversedBy="cart")
     * @JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;

    // ...
}
```

- *XML*

```
<doctrine-mapping>
    <entity name="Customer">
        <one-to-one field="cart" target-entity="Cart" mapped-by="customer" />
    </entity>
    <entity name="Cart">
        <one-to-one field="customer" target-entity="Customer" inversed-by="cart">
            <join-column name="customer_id" referenced-column-name="id" />
        </one-to-one>
    </entity>
```

```
</doctrine-mapping>
```

- *YAML*

```
Customer:
  oneToOne:
    cart:
      targetEntity: Cart
      mappedBy: customer
Cart:
  oneToOne:
    customer:
      targetEntity Customer
      inversedBy: cart
      joinColumn:
        name: customer_id:
        referencedColumnName: id
```

Ten en cuenta que la `@JoinColumn` realmente no es necesaria en este ejemplo, puesto que con los valores predeterminados sería lo mismo.

Esquema MySQL generado:

```
CREATE TABLE Cart (
    id INT AUTO_INCREMENT NOT NULL,
    customer_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Customer (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Cart ADD FOREIGN KEY (customer_id) REFERENCES Customer(id);
```

Observa cómo la clave externa se define en el lado propietario de la relación, la tabla `Carrito`.

1.6.8 Uno-A-Uno, autoreferencia

Puedes tener una referencia a sí mismo en una relación uno-a-uno como la siguiente.

```
<?php
/** @Entity */
class Student
{
    // ...

    /**
     * @OneToOne(targetEntity="Student")
     * @JoinColumn(name="mentor_id", referencedColumnName="id")
     */
    private $mentor;

    // ...
}
```

Ten en cuenta que la `@JoinColumn` realmente no es necesaria en este ejemplo, puesto que con los valores predeterminados sería lo mismo.

Con el esquema MySQL generado:

```
CREATE TABLE Student (
    id INT AUTO_INCREMENT NOT NULL,
    mentor_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Student ADD FOREIGN KEY (mentor_id) REFERENCES Student(id);
```

1.6.9 Uno-A-Muchos, unidireccional con tablas unidas

Puedes asignar una asociación uno-a-muchos unidireccional a través de una unión de tablas. Desde el punto de vista de *Doctrine*, simplemente es asignada como una muchos-a-muchos unidireccional por lo cual una restricción única en una de las columnas de la unión refuerza la cardinalidad de uno-a-muchos. El siguiente ejemplo, se configura tal como una asociación unidireccional uno-a-muchos:

Nota: Las relaciones uno-a-muchos unidireccionales con unión de tablas sólo trabaja usando la anotación `@ManyToMany` y una restricción de unicidad.

Genera el siguiente esquema MySQL:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE users_phonenumbers (
    user_id INT NOT NULL,
    phonenumber_id INT NOT NULL,
    UNIQUE INDEX users_phonenumbers_phonenumber_id_uniq (phonenumber_id),
    PRIMARY KEY(user_id, phonenumber_id)
) ENGINE = InnoDB;

CREATE TABLE Phonenumber (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

ALTER TABLE users_phonenumbers ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE users_phonenumbers ADD FOREIGN KEY (phonenumber_id) REFERENCES Phonenumber(id);
```

1.6.10 Muchos-A-Uno, unidireccional

Fácilmente puedes implementar una asociación unidireccional muchos-a-uno con lo siguiente:

- *PHP*

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToOne(targetEntity="Address")
     * @JoinColumn(name="address_id", referencedColumnName="id")
```

```
     */
    private $address;
}

/** @Entity */
class Address
{
    // ...
}
```

- *XML*

```
<doctrine-mapping>
    <entity name="User">
        <many-to-one field="address" target-entity="Address" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  manyToOne:
    address:
      targetEntity: Address
```

Nota: La `@JoinColumn` anterior es opcional ya que por omisión siempre son `address_id` e `id`. Los puedes omitir y dejar que se utilicen los valores predeterminados.

Esquema MySQL generado:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    address_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE Address (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

ALTER TABLE User ADD FOREIGN KEY (address_id) REFERENCES Address(id);
```

1.6.11 Uno-A-Muchos, bidireccional

Las asociaciones bidireccionales uno-a-muchos son muy comunes. El siguiente código muestra un ejemplo con una clase `Producto` y `Características`:

- *XML*

```
<doctrine-mapping>
    <entity name="Product">
        <one-to-many field="features" target-entity="Feature" mapped-by="product" />
    </entity>
    <entity name="Feature">
        <many-to-one field="product" target-entity="Product" inversed-by="features">
```

```
            <join-column name="product_id" referenced-column-name="id" />
        </many-to-one>
    </entity>
</doctrine-mapping>
```

Ten en cuenta que la `@JoinColumn` realmente no es necesaria en este ejemplo, puesto que con los valores predeterminados sería lo mismo.

Esquema MySQL generado:

```
CREATE TABLE Product (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Feature (
    id INT AUTO_INCREMENT NOT NULL,
    product_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Feature ADD FOREIGN KEY (product_id) REFERENCES Product(id);
```

1.6.12 Uno-A-Muchos, con autoreferencia

También puedes configurar una asociación uno-a-muchos con autoreferencia. En este ejemplo, configuramos una jerarquía de objetos `Categoría` creando una relación que hace referencia a sí misma. Esto efectivamente modela una jerarquía de Categorías y desde la perspectiva de la base de datos se conoce como un enfoque de lista adyacente.

- *PHP*

```
<?php
/** @Entity */
class Category
{
    // ...
    /**
     * @OneToMany(targetEntity="Category", mappedBy="parent")
     */
    private $children;

    /**
     * @ManyToOne(targetEntity="Category", inversedBy="children")
     * @JoinColumn(name="parent_id", referencedColumnName="id")
     */
    private $parent;
    // ...

    public function __construct() {
        $this->children = new \Doctrine\Common\Collections\ArrayCollection();
    }
}
```

- *XML*

```
<doctrine-mapping>
    <entity name="Category">
        <one-to-many field="children" target-entity="Category" mapped-by="parent" />
        <many-to-one field="parent" target-entity="Category" inversed-by="children" />
```

```
        </entity>
    </doctrine-mapping>
```

- *YAML*

```
Category:
  type: entity
  oneToMany:
    children
      targetEntity: Category
      mappedBy: parent
  manyToOne:
    parent:
      targetEntity: Category
      inversedBy: children
```

Ten en cuenta que la `@JoinColumn` realmente no es necesaria en este ejemplo, puesto que con los valores predeterminados sería lo mismo.

Esquema MySQL generado:

```
CREATE TABLE Category (
    id INT AUTO_INCREMENT NOT NULL,
    parent_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Category ADD FOREIGN KEY (parent_id) REFERENCES Category(id);
```

1.6.13 Muchos-A-Muchos, unidireccional

Las asociaciones de muchos-a-muchos reales son menos comunes. El siguiente ejemplo muestra una asociación unidireccional entre las entidades Usuario y Grupo:

- *PHP*

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="users_groups",
     *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="group_id", referencedColumnName="id")}
     *      )
     */
    private $groups;

    // ...

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\ArrayCollection();
    }
}

/** @Entity */
```

```
class Group
{
    // ...
}
```

- *XML*

```
<doctrine-mapping>
    <entity name="User">
        <many-to-many field="groups" target-entity="Group">
            <join-table name="users_groups">
                <join-columns>
                    <join-column="user_id" referenced-column-name="id" />
                </join-columns>
                <inverse-join-columns>
                    <join-column="group_id" referenced-column-name="id" />
                </inverse-join-columns>
            </join-table>
        </many-to-many>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  manyToMany:
    groups:
      targetEntity: Group
      joinTable:
        name: users_groups
        joinColumns:
          user_id:
            referencedColumnName: id
        inverseJoinColumns:
          group_id:
            referencedColumnName: id
```

Esquema MySQL generado:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE users_groups (
    user_id INT NOT NULL,
    group_id INT NOT NULL,
    PRIMARY KEY(user_id, group_id)
) ENGINE = InnoDB;
CREATE TABLE Group (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE users_groups ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE users_groups ADD FOREIGN KEY (group_id) REFERENCES Group(id);
```

Nota: ¿Por qué muchos-a-muchos son menos comunes? Debido a que frecuentemente deseas relacionar atributos adicionales con una asociación, en cuyo caso introduces una clase de asociación. En consecuencia, la asociación

muchos-a-muchos directa desaparece y es reemplazada por las asociaciones uno-a-muchos/muchos-a-uno entre las tres clases participantes.

1.6.14 Muchos-A-Muchos, bidireccional

Aquí hay una relación muchos-a-muchos similar a la anterior, salvo que esta es bidireccional.

- *PHP*

```php
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="Group", inversedBy="users")
     * @JoinTable(name="users_groups")
     */
    private $groups;

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\ArrayCollection();
    }
}

    // ...
}

/** @Entity */
class Group
{
    // ...
    /**
     * @ManyToMany(targetEntity="User", mappedBy="groups")
     */
    private $users;

    public function __construct() {
        $this->users = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}
```

- *XML*

```xml
<doctrine-mapping>
    <entity name="User">
        <many-to-many field="groups" inversed-by="users">
            <join-table name="users_groups">
                <join-columns>
                    <join-column="user_id" referenced-column-name="id" />
                </join-columns>
                <inverse-join-columns>
                    <join-column="group_id" referenced-column-name="id" />
                </inverse-join-columns>
```

```
            </join-table>
        </many-to-many>
    </entity>

    <entity name="Group">
        <many-to-many field="users" mapped-by="groups" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  manyToMany:
    groups:
      targetEntity: Group
      inversedBy: users
      joinTable:
        name: users_groups
        joinColumns:
          user_id:
            referencedColumnName: id
        inverseJoinColumns:
          group_id:
            referencedColumnName: id

Group:
  type: entity
  manyToMany:
    users:
      targetEntity: User
      mappedBy: groups
```

El esquema MySQL es exactamente el mismo que para el caso muchos-a-muchos unidireccional anterior.

Eligiendo el lado propietario o inverso

Para asociaciones muchos-a-muchos puedes elegir cual entidad es la propietaria y cual el lado inverso. Hay una regla semántica muy simple para decidir de qué lado es más adecuado el lado propietario desde la perspectiva de los desarrolladores. Sólo tienes que preguntarte, qué entidad es la responsable de gestionar la conexión y selección, como la parte propietaria.

Tomemos un ejemplo de dos entidades `Artículo` y `Etiqueta`. Cada vez que deseas conectar un Artículo a una Etiqueta y viceversa, sobre todo es el Artículo, el que es responsable de esta relación. Cada vez que añades un nuevo Artículo, quieres conectarlo con Etiquetas existentes o nuevas. Probablemente si creas un formulario Artículo apoyes esta idea y te permita especificar las etiquetas directamente. Es por eso que debes escoger el Artículo como el lado propietario, ya que hace más comprensible tu código:

```
<?php
class Article
{
    private $etiquetas;

    public function addTag(Tag $tag)
    {
        $tag->addArticle($this); // synchronously updating inverse side
        $this->tags[] = $tag;
```

```
    }
}

class Tag
{
    private $articles;

    public function addArticle(Article $article)
    {
        $this->articles[] = $article;
    }
}
```

Esto permite agrupar la etiqueta añadiéndola en el lado `Artículo` de la asociación:

```
<?php
$article = new Article();
$article->addTag($tagA);
$article->addTag($tagB);
```

1.6.15 Muchos-A-Muchos, con autoreferencia

Incluso, puedes tener una autoreferencia en una asociación de muchos-a-muchos. Un escenario común es cuando un `Usuario` tiene amigos y la entidad destino de esa relación es un `Usuario` lo cual la hace una autoreferencia. En este ejemplo es bidireccional puesto que el `Usuario` tiene un campo llamado `$friendsWithMe` y `$myFriends`.

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="User", mappedBy="myFriends")
     */
    private $friendsWithMe;

    /**
     * @ManyToMany(targetEntity="User", inversedBy="friendsWithMe")
     * @JoinTable(name="friends",
     *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="friend_user_id", referencedColumnName="id")}
     *      )
     */
    private $myFriends;

    public function __construct() {
        $this->friendsWithMe = new \Doctrine\Common\Collections\ArrayCollection();
        $this->myFriends = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}
```

Esquema MySQL generado:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE friends (
    user_id INT NOT NULL,
    friend_user_id INT NOT NULL,
    PRIMARY KEY(user_id, friend_user_id)
) ENGINE = InnoDB;
ALTER TABLE friends ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE friends ADD FOREIGN KEY (friend_user_id) REFERENCES User(id);
```

1.6.16 Ordenando colecciones A-Muchos

En muchos casos de uso desearás ordenar colecciones cuando las recuperas de la base de datos. En el espacio de los usuarios lo haces, siempre y cuando inicialmente, no se haya guardado en la base de datos una entidad con sus asociaciones. Para recuperar una colección ordenada desde la base de datos puedes utilizar la anotación `@OrderBy` con una colección que especifica un fragmento `DQL` que se añade a todas las consultas con esta colección.

Adicional a las anotaciones `@OneToMany` o `@ManyToMany` puedes especificar la anotación `@OrderBy` de la siguiente manera:

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="Group")
     * @OrderBy({"name" = "ASC"})
     */
    private $groups;
}
```

El fragmento `DQL` en `OrderBy` sólo puede consistir de nombres de campo no calificados, sin comillas y de una opcional declaración `ASC/DESC`. Múltiples campos van separados por una coma (,). Los nombres de los campos referidos existentes en la clase `targetEntity` de la anotación `@ManyToMany` o `@OneToMany`.

La semántica de esta característica se puede describir de la siguiente manera.

- `@OrderBy` actúa como una `ORDER BY` implícita para los campos dados, la cual se anexa explícitamente a todos los elementos `ORDER BY` dados.
- Todas las colecciones del tipo ordenado siempre se recuperan de una manera ordenada.
- Para mantener un bajo impacto en la base de datos, estos elementos `ORDER BY` implícitos sólo se agregan a una consulta `DQL` si la colección se recupero de una unión en la consulta `DQL`.

Teniendo en cuenta nuestro ejemplo definido anteriormente, la siguiente no añadirá `ORDER BY`, ya que *g* no se recuperó de una unión:

```
SELECT u FROM User u JOIN u.groups g WHERE SIZE(g) > 10
```

Sin embargo, la siguiente:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10
```

...internamente se reescribirá a:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10 ORDER BY g.name ASC
```

No puedes invertir el orden con una `ORDER BY DQL` explícita:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10 ORDER BY g.name DESC
```

...internamente se reescribe a:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10 ORDER BY g.name DESC, g.name ASC
```

1.7 Asignando herencia

1.7.1 Asignando superclases

Una asignación de superclase es una clase abstracta o concreta que proporciona estado persistente a la entidad y la información de asignación de sus subclases, pero que en sí misma no es una entidad. Por lo general, el propósito de esta superclase asignada es definir la información de estado y la asignación que es común a varias clases de entidades.

Las superclases asignadas, así como regulares, no asignan las clases, pueden aparecer en medio de una jerarquía de herencia asignada de otro modo (a través de la herencia de una sola tabla o de la herencia de tablas de clase).

Nota: Una superclase asignada no puede ser una entidad, no es capaz de consultas y las relaciones persistentes las debe definir una superclase unidireccional asignada (con solamente un lado propietario). Esto significa que las asociaciones uno-a-muchos en una superclase asignada no son posibles en absoluto. Además, asignaciones muchos-a-muchos sólo son posibles si la superclase asignada sólo utiliza exactamente una entidad al momento. Para mayor apoyo de herencia, las características de herencia tienes que usar una sola tabla o unión.

Ejemplo:

```
<?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;

    // ... más campos y métodos
}

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

```
    // ... más campos y métodos
}
```

El DDL para el esquema de base de datos correspondiente se vería algo como esto (esto es para SQLite):

```
CREATE TABLE EntitySubClass (mapped1 INTEGER NOT NULL, mapped2 TEXT NOT NULL, id INTEGER NOT NULL, na
```

Como puedes ver en este fragmento DDL, sólo hay una única tabla de la subclase entidad. Todas las asignaciones de la superclase asignada fueron heredadas de la subclase como si se hubieran definido en dicha clase directamente.

1.7.2 Herencia de una sola tabla

La herencia de tabla única es una estrategia de asignación de herencias donde todas las clases de la jerarquía se asignan a una única tabla de la base de datos. A fin de distinguir qué fila representa cual tipo en la jerarquía, se utiliza una así llamada "columna discriminadora".

Ejemplo:

```
<?php
namespace MiProyecto\Model;

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

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

Cosas a tener en cuenta:

- Los `@InheritanceType`, `@DiscriminatorMap` y `@DiscriminatorColumn` se deben especificar en la clase superior que forma parte de la jerarquía de la entidad asignada.
- El `@DiscriminatorMap` especifica cuales valores de la columna discriminadora identifican una fila como de un cierto tipo. En el caso anterior un valor de "persona", define una fila como de tipo `Persona` y "empleado" identifica una fila como de tipo `Empleado`.
- Los nombres de las clases en la asignación discriminadora no tienen que estar completos si las clases están contenidas en el mismo espacio de nombres como la clase entidad en la que se aplica la asignación discriminadora.

Consideraciones en tiempo de diseño

Este enfoque de asociación funciona bien cuando la jerarquía de tipos es bastante simple y estable. Añadir un nuevo tipo a la jerarquía y agregar campos a supertipos existentes simplemente consiste en añadir nuevas columnas a la tabla,

aunque en grandes despliegues esto puede tener un impacto negativo en el índice y diseño de la columna dentro de la base de datos.

Impacto en el rendimiento

Esta estrategia es muy eficaz para consultar todos los tipos de la jerarquía o para tipos específicos. No se requieren uniones de tablas, sólo una cláusula `WHERE` listando los identificadores de tipo. En particular, las relaciones que implican tipos que emplean esta estrategia de asignación son de buen calidad.

Hay una consideración de rendimiento general, con herencia de tabla única: si utilizas una entidad STI como muchos-a-uno o una entidad uno-a-uno nunca debes utilizar una de las clases en los niveles superiores de la jerarquía de herencia como `"targetEntity"` sólo aquellas que no tienen subclases. De lo contrario *Doctrine NO PUEDE* crear instancias delegadas de esta entidad y *SIEMPRE* debe cargar la entidad impacientemente.

Consideraciones del esquema SQL

Una herencia de una sola tabla trabaja en escenarios en los que estás utilizando un esquema de base de datos existente o un esquema de base de datos autoescrito que tienes que asegurarte de que todas las columnas que no están en la entidad raíz, sino en cualquiera de las diferentes subentidades tiene que permite valores nulos. Las columnas que no tienen restricciones `NULL`, tienen que estar en la entidad raíz de la jerarquía de herencia de una sola tabla.

1.7.3 Herencia clase→tabla

La herencia clase→tabla es una estrategia de asignación de herencias, donde se asigna cada clase en una jerarquía de varias tablas: su propia tabla y las tablas de todas las clases padre. La tabla de una clase hija está relacionada con la tabla de una clase padre a través de una clave externa. *Doctrine* 2 implementa esta estrategia a través del uso de una columna discriminadora de la tabla superior de la jerarquía, porque es la forma más fácil de lograr consultas polimórficas con herencia clase→tabla.

Ejemplo:

```
<?php
namespace MiProyecto\Model;

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

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

Cosas a tener en cuenta:

- Los `@InheritanceType`, `@DiscriminatorMap` y `@DiscriminatorColumn` se deben especificar en la clase superior que forma parte de la jerarquía de la entidad asignada.

- El `@DiscriminatorMap` especifica cuales valores de la columna discriminadora identifican una fila como de qué tipo. En el caso anterior un valor de "persona", define una fila como de tipo `Persona` y "empleado" identifica una fila como de tipo `Empleado`.
- Los nombres de las clases en la asignación discriminadora no tienen que estar completos si las clases están contenidas en el mismo espacio de nombres como la clase entidad en la que se aplica la asignación discriminadora.

Nota: Cuando no utilices la `SchemaTool` para generar el código `SQL` necesario debes saber que la supresión de una herencia de tablas de la clase utiliza la propiedad `ON DELETE CASCADE` de la clave externa en todas las implementaciones de bases de datos. Una falla al implementar esto tú mismo darás lugar a filas muertas en la base de datos.

Consideraciones en tiempo de diseño

Introducir un nuevo tipo a la jerarquía, a cualquier nivel, implica simplemente intercalar una nueva tabla en el esquema. Los subtipos de este tipo se unirán automáticamente con el nuevo tipo en tiempo de ejecución. Del mismo modo, la modificación de cualquier tipo de entidad en la jerarquía agregando, modificando o eliminando campos afecta sólo a la tabla inmediata asignado ese tipo. Esta estrategia de asignación proporciona la mayor flexibilidad en tiempo de diseño, ya que los cambios de cualquier tipo son siempre limitados al tipo dedicado de la tabla.

Impacto en el rendimiento

Esta estrategia inherentemente requiere múltiples operaciones `JOIN` para realizar cualquier consulta que pueda tener un impacto negativo en el rendimiento, especialmente con tablas de gran tamaño y/o grandes jerarquías. Cuando los se permiten objetos parciales, ya sea globalmente o en la consulta específica, entonces consultar por cualquier tipo no hará que las tablas de los subtipos sean `OUTER JOIN` lo cual puede aumentar el rendimiento, pero los objetos parciales resultantes no se cargan completamente al acceder a los campos de cualquier subtipo, por lo que acceder a los campos de los subtipos después de una consulta no es seguro.

Hay una consideración de rendimiento general, con la herencia de clases de tabla: si utilizas una entidad CTI como muchos-a-uno o una entidad uno-a-uno nunca debes utilizar una de las clases en los niveles superiores de la jerarquía de herencia como `"targetEntity"`, sólo aquellas que no tienen subclases. De lo contrario *Doctrine NO PUEDE* crear instancias delegadas de esta entidad y *SIEMPRE* debe cargar la entidad impacientemente.

Consideraciones del esquema `SQL`

Para cada entidad en la jerarquía de herencia de clase→tabla todos los campos asignados tienen que ser columnas en la tabla de esta entidad. Además, cada tabla secundaria tiene que tener una columna `id` que concuerde con la definición de la columna `id` en la tabla raíz (a excepción de alguna secuencia o detalles de autoincremento). Además, cada tabla secundaria debe tener una clave externa apuntando desde la columna `id` a la columna `id` de la tabla raíz y eliminación en cascada.

1.8 Trabajando con objetos

Este capítulo te ayudará a entender el `EntityManager` y la `UnitOfWork`. Una unidad de trabajo es similar a una transacción a nivel de objeto. Una nueva unidad de trabajo se inicia implícitamente cuando inicialmente se crea un `EntityManager` o después de haber invocado a `EntityManager#flush()`. Se consigna una unidad de trabajo (y se inicia otra nueva) invocando a `EntityManager#flush()`.

Una unidad de trabajo se puede cerrar manualmente llamando al método `EntityManager#close()`. Cualquier cambio a los objetos dentro de esta unidad de trabajo que aún no se haya persistido, se pierde.

Nota: Es muy importante entender que `EntityManager#flush()` por sí mismo, nunca ejecuta las operaciones de escritura contra la base de datos. Cualquier otro método como `EntityManager#persist($entidad)` o `EntityManager#remove($entidad)`, únicamente notifica a la Unidad de trabajo que realice estas operaciones durante el vaciado.

El **no** llamar a `EntityManager#flush()` conlleva a que se pierdan todos los cambios realizados durante la petición.

1.8.1 Entidades y asignación de identidad

Las entidades son objetos con identidad. Su identidad tiene un significado conceptual dentro de tu dominio. En una aplicación CMS (por Content Manager System, en Español: Sistema de administración de contenido) cada artículo tiene un identificador único. Puedes identificar cada artículo por ese `id`.

Tomemos el siguiente ejemplo, donde encontramos un artículo titulado "Hola Mundo" con el `id` 1234:

```
<?php
$article = $entityManager->find('CMS\Article', 1234);
$article->setHeadline('¡Hola mundo duda!');

$article2 = $entityManager->find('CMS\Article', 1234);
echo $article2->getHeadline();
```

En este caso, el Artículo se puede acceder desde el administrador de la entidad en dos ocasiones, pero modificado en el medio. *Doctrine* 2 se da cuenta de esto y sólo alguna vez te dará acceso a una instancia del Artículo con `id` 1234, no importa con qué frecuencia recuperes el `EntityManager` e incluso no importa qué tipo de método de consulta estés utilizando (`find`, `Repository finder` o `DQL`). Esto se llama patrón "asignador de identidad", lo cual significa que *Doctrine* mantiene la asignación de cada entidad y los identificadores que se han recuperado por petición *PHP* y te sigue devolviendo las mismas instancias.

En el ejemplo anterior el `echo` imprime "¡Hola mundo duda!" en la pantalla. Incluso puedes comprobar que `$articulo` y `$artículo2`, de hecho, apuntan a la misma instancia ejecutando el siguiente código:

```
<?php
if ($article === $article2) {
    echo "¡Sí, somos el mismo!";
}
```

A veces querrás borrar la asignación de identidad de un `EntityManager` para empezar de nuevo. Usamos esta regularidad en nuestra unidad de pruebas para forzar de nuevo la carga de objetos desde la base de datos en lugar de servirlas desde la asignación de identidad. Puedes llamar a `EntityManager#clear()` para lograr este resultado.

1.8.2 Gráfico transversal del objeto Entidad

Aunque *Doctrine* permite una separación completa de tu modelo del dominio (clases de Entidad) nunca habrá una situación en la que los objetos "desaparezcan" cuando se atraviesan asociaciones. Puedes recorrer todas las asociaciones dentro de las entidades de tu modelo tan profundamente como quieras.

Tomemos el siguiente ejemplo de una única entidad `Artículo` recuperada por el `EntityManager` recién abierto.

```
<?php
/** @Entity */
class Article
{
    /** @Id @Column(type="integer") @GeneratedValue */
```

```
    private $id;

    /** @Column(type="string") */
    private $headline;

    /** @ManyToOne(targetEntity="User") */
    private $author;

    /** @OneToMany(targetEntity="Comment", mappedBy="article") */
    private $comments;

    public function __construct {
        $this->comments = new ArrayCollection();
    }

    public function getAuthor() { return $this->author; }
    public function getComments() { return $this->comments; }
}

$article = $em->find('Article', 1);
```

Este código sólo recupera la instancia del `Artículo`, con `id` 1 ejecutando una sola instrucción `SELECT` en la tabla Usuario de la base de datos. Todavía puedes tener acceso a las propiedades asociadas Autor y Comentarios y los objetos asociados que contienen.

Esto funciona utilizando el patrón de carga diferida. En lugar de pasar una instancia real del Autor y una colección de comentarios, *Doctrine* creará instancias delegadas para ti. Sólo si tienes acceso a estos delegados la primera vez vas a ir a través del `EntityManager` y cargará su estado desde la base de datos.

Este proceso de carga diferida, ocurre detrás del escenario, oculto desde tu código. Ve el siguiente fragmento:

```
<?php
$article = $em->find('Article', 1);

// acceder a un método de la instancia de usuario activa la carga diferida
echo "Author: " . $article->getAuthor()->getName() . "\n";

// carga diferida de delegados para las pruebas de instancia:
if ($article->getAuthor() instanceof User) {
    // un Usuario delegado es una clase "UserProxy" generada
}

// acceder a los comentarios como iterador activa la carga diferida
// recupera TODOS los comentarios de este artículo desde la base de
// datos usando una única declaración SELECT
foreach ($article->getComments() AS $comment) {
    echo $comment->getText() . "\n\n";
}

// Article::$comments pasa las pruebas de instancia para la interfaz Collection
// Pero la interfaz ArrayCollection NO debe pasar
if ($article->getComments() instanceof \Doctrine\Common\Collections\Collection) {
    echo "This will always be true!";
}
```

Un trozo de código generado por la clase delegada se parece al siguiente fragmento de código. Una clase delegada real redefine *TODOS* los métodos públicos a lo largo de las líneas del método `getName()` mostrado a continuación:

```php
<?php
class UserProxy extends User implements Proxy
{
    private function _load()
    {
        // código de carga diferida
    }

    public function getName()
    {
        $this->_load();
        return parent::getName();
    }
    // .. otros métodos públicos de Usuario
}
```

Advertencia: Atravesar el objeto gráfica por las piezas que son cargadas de manera diferida fácilmente activará un montón de consultas SQL y un mal resultado si se utiliza mucho. Asegúrate de utilizar DQL para recuperar uniones de todas las partes del objeto gráfico que necesitas lo más eficientemente posible.

1.8.3 Persistiendo entidades

Una entidad se puede persistir pasándola al método `EntityManager#persist($entidad)`. Al aplicar la operación de persistencia en una entidad, esa entidad se convierte en GESTIONADA, lo cual significa que su persistencia a partir de ahora es gestionada por un `EntityManager`. Como resultado, el estado persistente de una entidad posteriormente se sincroniza correctamente con la base de datos cuando invocas a `EntityManager#flush()`.

Nota: Invocar al método `persist` en la entidad no provoca la inmediata publicación de una `INSERT SQL` en la base de datos. *Doctrine* aplica una estrategia llamada "transacción de escritura en segundo plano", lo cual significa que se demoran muchas declaraciones SQL hasta que se invoque a `EntityManager#flush()`, el cual emitirá todas las declaraciones SQL necesarias para sincronizar los objetos con la base de datos de la manera más eficiente y en una sola, breve transacción, teniendo cuidado de mantener la integridad referencial.

Ejemplo:

```php
<?php
$user = new User;
$user->setName('Mr.Right');
$em->persist($user);
$em->flush();
```

Nota: Los identificadores de entidad / claves primarias generadas está garantizado que estén disponibles después en la próxima operación de vaciado exitosa que involucre a la entidad en cuestión. No puedes confiar en que un identificador generado esté disponible inmediatamente después de invocar a `persist`. La inversa también es cierta. No puedes confiar en que un identificador generado no esté disponible después de una fallida operación de vaciado.

Las semánticas de la operación de persistencia, aplicadas en una entidad X, son las siguientes:

- Si X es una nueva entidad, se convierte en gestionada. La entidad X se deberá incluir en la base de datos como resultado de la operación de vaciado.
- Si X es una entidad gestionada preexistente, es ignorada por la operación persistencia. Sin embargo, la operación de persistencia en cascada para las entidades a que X hace referencia, si la relación de X a estas otras entidades

se asignaron con `cascade=PERSIST` o `cascade=ALL` (consulta la sección "Persistencia transitiva").

- Si X es una entidad removida, se convierte en gestionada.
- Si X es una entidad independiente, lanzará una excepción en el vaciado.

1.8.4 Removiendo entidades

Una entidad se puede remover del almacenamiento persistente pasándola al método `EntityManager#remove($entidad)`. Al aplicaa la operación `remove` a alguna entidad, esa entidad se convierte en REMOVIDA, lo cual significa que su estado persistente se cancelará una vez invocado el `EntityManager#flush()`.

Nota: Al igual que `persist`, invocar a `remove` en una entidad no provoca la publicación inmediata de una declaración `DELETE SQL` en la base de datos. La entidad se eliminará en la siguiente llamada a `EntityManager#flush()` que implica a esa entidad. Esto significa que las entidades programadas para remoción aún se pueden consultar y aparecen en la consulta y resultados de la colección. Ve la sección en *Base de datos y unidad de trabajo desincronizadas* para más información.

Ejemplo:

```
<?php
$em->remove($user);
$em->flush();
```

Las semánticas de la operación de remoción, aplicadas a una entidad X son las siguientes:

- Si X es una nueva entidad, esta es ignorada por la operación de remoción. Sin embargo, la operación de remoción en cascada se aplica a las entidades referidas en X, si la relación de X a estas otras entidades se asignaron con `cascade=REMOVE` o `cascade=ALL`. (Ve "Persistencia transitiva")
- Si X es una entidad gestionada, la operación de remoción hace que se elimine. La operación de remoción se propaga a las entidades referidas en X, si la relación de X a estas otras entidades se asignó con `cascade=REMOVE` o `cascade=ALL` (consulta la sección "Persistencia transitiva").
- Si X es una entidad independiente, lanzará un `InvalidArgumentException`.
- Si X es una entidad removida, es ignorada por la operación de remoción.
- Una entidad X removida se eliminará de la base de datos como resultado de la operación de vaciado.

Después de eliminar una entidad su estado en memoria es el mismo que antes de su remoción, a excepción de los identificadores generados.

Al remover la entidad automáticamente se eliminan los registros existentes en tablas unidas muchos-a-muchos que enlazan a esta entidad. Las acciones adoptadas en función del valor del atributo `@JoinColumn` asignado a "onDelete". Ya sea cuestión de que *Doctrine* tenga una declaración `DELETE` dedicada para los registros de cada tabla unida o depende de la semántica en la clave externa de `cascade=onDelete`.

Puedes conseguir eliminar un objeto con todos sus objetos asociados de varias maneras con muy diferentes impactos en el rendimiento.

1. Si una asociación está marcada como `CASCADE=REMOVE` *Doctrine* 2 recuperará esta asociación. Si es una sola asociación pasará esta entidad a `EntityManager#remove()`. Si la asociación es una colección, *Doctrine* la repetirá en todos sus elementos y los pasará a ''EntityManager#remove()'. En ambos casos, la semántica de eliminación en cascada se aplica recurrentemente. Para grandes objetos gráficos, esta estrategia de eliminación puede ser muy costosa.

2. Usar un `DELETE DQL` te permite borrar varias entidades de un tipo con una única orden y sin la hidratación de estas entidades. Esto puede ser muy eficaz para eliminar grandes objetos gráficos de la base de datos.
3. Usar la semántica de clave externas `onDelete="CASCADE"` puede obligar a la base de datos a eliminar internamente todos los objetos asociados. Esta estrategia es un poco difícil lograrla bien, pero puede ser muy potente y rápida. Debes estar consciente, sin embargo, que al usar la estrategia 1 (`CASCADE=REMOVE`) evades completamente cualquier opción de clave externa `onDelete=CASCADE`, porque no obstante, *Doctrine* explícitamente busca y elimina todas las entidades asociadas.

1.8.5 Disociando entidades

Una entidad se disocia de un `EntityManager` y por lo tanto ya no es gestionada invocando en ella al método `EntityManager#detach($entidad)` o al propagar en cascada las operaciones de disociación. Los cambios realizados a la entidad independiente, en su caso (incluyendo la eliminación de la entidad), no se sincronizarán con la base de datos después de haber disociado la entidad.

Doctrine no mantendrá ninguna referencia a una entidad disociada.

Ejemplo:

```
<?php
$em->detach($entity);
```

Las semánticas de la operación de disociación, aplicadas a una entidad X son las siguientes:

- Si X es una entidad gestionada, la operación de disociación provoca que se desprenda. La operación de disociación se propaga en cascada a las entidades referidas en X, si la relación de X a estas otras entidades se asignó con `cascade=DETACH` o `cascade=ALL` (consulta la sección "Persistencia transitiva"). Las entidades que anteriormente hacían referencia a X continuarán haciendo referencia a X.
- Si X es una nueva entidad o es independiente, esta es ignorada por la operación de disociación.
- Si X es una entidad removida, la operación de disociación es propagada en cascada a las entidades referidas en X, si la relación de X a estas otras entidades se asignó con `cascade=DETACH` o `cascade=ALL` (consulta la sección "Persistencia transitiva"). Las entidades que anteriormente hacían referencia a X continuarán haciendo referencia a X.

Hay varias situaciones en las cuales una entidad es disociada automáticamente sin necesidad de invocar el método `detach`:

- Cuando se invoca a `EntityManager#clear()`, disocia todas las entidades que son gestionadas actualmente por la instancia de `EntityManager`.
- Al serializar una entidad. La entidad recuperada en posteriores deserializaciones se disociará (Este es el caso de todas las entidades que se serializan se almacenen en caché, es decir, cuando utilizas la caché de resultados de consulta).

La operación `detach` no suele ser necesaria tan frecuentemente y se utiliza como `persist` y `remove`.

1.8.6 Fusionando entidades

La fusión de entidades se refiere a la combinación de entidades (por lo general independientes) en el contexto de un `EntityManager` para que sean gestionadas de nuevo. Para combinar el estado de una entidad en un `EntityManager` utiliza el método `EntityManager#merge($entidad)`. El estado de la entidad pasada se fusionará en una copia gestionada de esta entidad y, posteriormente, la copia será devuelta.

Ejemplo:

```php
<?php
$detachedEntity = unserialize($serializedEntity); // alguna entidad disociada
$entity = $em->merge($detachedEntity);
// $entity ahora se refiere a la copia completamente gestionada revuelta por la operación de combina
// el EntityManager $em ahora gestiona la persistencia de $entity de la manera usual.
```

Nota: Cuando deseas serializar/deserializar entidades tienes que hacer todas las propiedades de la entidad protegidas, nunca privadas. La razón de esto es que, si serializas una clase que antes era una instancia delegada, las variables privadas no se pueden serializar y lanzará un Aviso de *PHP*.

Las semánticas de la operación de fusión, aplicadas a una entidad X, son las siguientes:

- Si X es una entidad independiente, el estado de X se copia en una instancia preexistente de la entidad X gestionada con la misma identidad.
- Si X es una nueva instancia de la entidad, se creará una nueva copia gestionada de X y el estado de X se copiará a esta instancia gestionada.
- Si X es una instancia removida de la entidad, se lanzará un `InvalidArgumentException`.
- Si X es una entidad gestionada, es ignorada por la operación de fusión, sin embargo, la operación de fusión se propaga en cascada a las entidades referidas en las relaciones de X, si estas relaciones se han asignado con el valor del elemento en cascada `MERGE` o `ALL` (consulta la sección "Persistencia transitiva").
- Para todas las entidades Y referidas en las relaciones de X que tienen el valor del elemento en cascada `MERGE` o `ALL`, Y se fusiona recurrentemente como Y'. Para todas las referencias de Y por X, X' se establece como referencia a Y'. (Ten en cuenta que si X es gestionada entonces X es el mismo objeto que X').
- Si X es una entidad fusionada a X', con una referencia a otra entidad Y, donde no se especifica `cascade=MERGE` o `cascade=ALL`, entonces navegando en la misma asociación de X' produce una referencia a un objeto administrado Y' con la misma identidad persistente que Y.

La operación `merge` lanzará un `OptimisticLockException` si la entidad fusionada utiliza el bloqueo optimista a través de un campo de versión y las versiones de la entidad fusionada y la copia gestionada no coinciden. Esto generalmente significa que la entidad fue modificada, mientras se disociaba.

La operación `merge` no suele ser necesaria tan frecuentemente y se utiliza como `persist` y `remove`. El escenario más común para la operación `merge` es para volver a colocar las entidades en un `EntityManager` que proviene de alguna caché (y por lo tanto, disociada) y que deseas modificar y persistir como entidad.

Advertencia: Si necesitas realizar fusiones de múltiples entidades que comparten ciertas subpartes de sus objetos gráficos y las fusionas en cascada, entonces tienes que llamar a `EntityManager#clear()` entre las llamadas sucesivas a `EntityManager#merge()` . De lo contrario, podrías terminar con varias copias del "mismo" objeto en la base de datos, sin embargo, con diferentes identificadores.

Nota: Si cargas algunas entidades disociadas desde una memoria caché y no es necesario persistirlas, borrarlas o hacer uso de ellas sin la necesidad de servicios de persistencia, no hay necesidad de usar `merge`. Es decir, sólo tienes que pasar objetos disociados desde una caché directamente a la vista.

1.8.7 Sincronizando con la base de datos

El estado de las entidades persistentes se sincroniza con la base de datos al vaciar el `EntityManager` el cual consigna las `UnitOfWork` subyacentes. La sincronización involucra la escritura de cualquier actualización a las entidades persistentes y sus relaciones a la base de datos. Con ello se conservan las relaciones bidireccionales en

base a las referencias mantenidas por el lado propietario de la relación, como se explica en el capítulo *Asignando asociaciones*.

Cuando invocas a `EntityManager#flush()`, *Doctrine* inspecciona todas las entidades gestionadas, nuevas y removidas y lleva a cabo las siguientes operaciones.

Efectos de una base de datos y unidad de trabajo desincronizadas

Tan pronto como comiences a cambiar el estado de las entidades, llamas a `persist` o eliminas contenido de la `UnitOfWork`, la base de datos estará fuera de sincronía. Sólo puedes sincronizarla llamando a `EntityManager#flush()`. Esta sección describe los efectos producidos cuando la base de datos y la `UnitOfWork` están fuera de sincronía.

- Las entidades que se han programado para eliminarse aún se pueden consultar desde la base de datos. Estas se devuelven desde consultas `DQL` y del repositorio y son visibles en las colecciones.
- Las entidades pasadas a `EntityManager#persist` no aparecen en el resultado de la consulta.
- Las entidades que han cambiado no se sobrescribe con el estado de la base de datos. Esto es así porque la asignación de identidad detecta la construcción de una entidad ya existente y asume que esta es la versión actualizada.

`EntityManager#flush()` nunca es llamado implícitamente por *Doctrine*. Siempre tienes que activarlo manualmente.

Sincronizando entidades nuevas y gestionadas

La operación de vaciado se aplica a una entidad gestionada con la semántica siguiente:

- La propia entidad se sincroniza con la base de datos utilizando una declaración `UPDATE SQL`, sólo si por lo menos un campo persistente ha cambiado.
- No hay actualizaciones `SQL` a ejecutar si la entidad no ha cambiado.

La operación de vaciado se aplica a una nueva entidad con la semántica siguiente:

- La propia entidad se sincroniza con la base de datos usando una instrucción `INSERT SQL`.

Para todas las relaciones (iniciadas) de la nueva o gestionada entidad la semántica se aplica a cada entidad X asociada:

- Si X es nueva y las operaciones de persistencia se han configurado en cascada en la relación, X se conservará.
- Si X es nueva y las operaciones de persistencia no se han configurado en cascada en la relación, una excepción será lanzada ya que esto indica un error de programación.
- Si X es eliminada y las operaciones de persistencia se configuraron en cascada sobre la relación, una excepción será lanzada ya que esto indica un error de programación (X sería persistida de nuevo por la cascada).
- Si X es independiente y las operaciones de persistencia se han configurado en cascada en la relación, una excepción será lanzada (esto semánticamente es lo mismo que pasar X a `persist()`).

Sincronizando entidades removidas

La operación de vaciado se aplica a una entidad removida al eliminar de la base de datos su estado persistente. No hay opciones en cascada relevantes para las entidades removidas en el vaciado, las opciones en cascada de `remove` ya están ejecutadas durante el `EntityManager#remove($entidad)`.

El tamaño de la unidad de trabajo

El tamaño de una unidad de trabajo se refiere principalmente al número de entidades administradas en un determinado punto en el tiempo.

El costo del vaciado

Qué tan costosa es una operación de vaciado, depende principalmente de dos factores:

- El tamaño de la `UnitOfWork` del `EntityManager` actual.
- La configuración de las políticas del control de cambios

Puedes obtener el tamaño de una `UnitOfWork` de la siguiente manera:

```
<?php
$uowSize = $em->getUnitOfWork()->size();
```

El tamaño representa el número de entidades administradas en la unidad de trabajo. Este tamaño afecta al rendimiento de `flush()`, debido a las operaciones del control de cambios (consulta "Políticas del control de cambios") y, por supuesto, la memoria consumida, así que posiblemente desees comprobarlo de vez en cuando durante el desarrollo.

Nota: No invoques a `flush` después de cada cambio a una entidad o en cada invocación a `persist/remove/merge/...` Este es un antipatrón e innecesariamente reduces el rendimiento de tu aplicación. En cambio, organiza las unidades de trabajo que operan en tus objetos y cuando hayas terminado invoca a `flush`. Mientras sirves una sola petición *HTTP* por lo general no hay necesidad de invocar a `flush` más de 0-2 veces.

Accediendo directamente a una unidad de trabajo

Puedes acceder directamente a la unidad de trabajo llamando a `EntityManager#getUnitOfWork()`. Esto devolverá la instancia de la `UnitOfWork` del `EntityManager` utilizada actualmente.

```
<?php
$uow = $em->getUnitOfWork();
```

Nota: No recomendamos manipular directamente una unidad de trabajo. Cuando trabajes directamente con la *API* de la `UnitOfWork`, respeta los métodos de marcado como `INTERNAL` para no usarlos y lee cuidadosamente la documentación de la *API*.

Estado de la entidad

Como indicamos en la descripción de la arquitectura una entidad puede estar en uno de cuatro estados posibles: `NEW`, `MANAGED`, `REMOVED`, `DETACHED`. Si necesitas explícitamente averiguar cuál es el estado actual de la entidad estando en el contexto de un cierto `EntityManager` puedes consultar la `UnitOfWork` subyacente:

```
<?php
switch ($em->getUnitOfWork()->getEntityState($entity)) {
    case UnitOfWork::STATE_MANAGED:
        ...
    case UnitOfWork::STATE_REMOVED:
        ...
    case UnitOfWork::STATE_DETACHED:
        ...
```

```
    case UnitOfWork::STATE_NEW:
        ...
}
```

La entidad se encuentra en estado GESTIONADO si está asociada con un `EntityManager` y que no se ha REMOVIDO.

La entidad se encuentra en estado REMOVIDO después de que se ha pasado a `EntityManager#remove()` y hasta la siguiente operación de vaciado del mismo `EntityManager`. Una entidad REMOVIDA todavía está asociada con un `EntityManager` hasta que la siguiente operación de vaciado.

La entidad se encuentra en estado DISOCIADO si tiene estado persistente e identidad, pero no está asociada con un `EntityManager`.

La entidad se encuentra en estado NUEVO si no tiene un estado persistente e identidad y no está asociada con un `EntityManager` (por ejemplo, la acabas de crear a través del operador `new`).

1.8.8 Consultando

Doctrine 2 proporciona los siguientes medios, por nivel de potencia incremental y flexibilidad, para consultar objetos persistentes. Siempre debes comenzar con el más simple que se adapte a tus necesidades.

Por clave primaria

La forma de consulta más básica para un objeto persistente es su identificador/clave primaria usando el método `EntityManager#find($nombreEntidad, $id)`. Aquí está un ejemplo:

```
<?php
// $em instancia del EntityManager
$user = $em->find('MiProyecto\Domain\User', $id);
```

El valor de retorno es o bien la instancia entidad encontrada o nulo si no se puede encontrar una instancia con el identificador dado.

Esencialmente, `EntityManager#find()` sólo es un atajo para lo siguiente:

```
<?php
// $em instancia del EntityManager
$user = $em->getRepository('MiProyecto\Domain\User')->find($id);
```

`EntityManager#getRepository($nombreEntidad)` devuelve un objeto repositorio que ofrece muchas maneras de recuperar entidades del tipo especificado. De manera predeterminada, la instancia del repositorio es del tipo `Doctrine\ORM\EntityRepository`. También puedes utilizar clases repositorio personalizadas como se indica más adelante.

Por condiciones sencillas

Para consultar una o más entidades basándote en una serie de condiciones que forman una conjunción lógico, usa los métodos `findBy` y `findOneBy` en un repositorio de la siguiente manera:

```
<?php
// $em instancia del EntityManager

// Todos los usuario de 20 años o más
$users = $em->getRepository('MiProyecto\Domain\User')->findBy(array('age' => 20));
```

```
// Todos los usuarios de 20 años o más que se apellidan 'Miller'
$users = $em->getRepository('MiProyecto\Domain\User')->findBy(array('age' => 20, 'surname' => 'Mille

// Un sólo usuario por sobrenombre
$user = $em->getRepository('MiProyecto\Domain\User')->findOneBy(array('nickname' => 'romanb'));
```

También puedes cargar asociaciones por el lado propietario a través del repositorio:

```
<?php
$number = $em->find('MiProyecto\Domain\Phonenumber', 1234);
$user = $em->getRepository('MiProyecto\Domain\User')->findOneBy(array('phone' => $number->getId()));
```

Ten cuidado porque esto sólo funciona pasando el `id` de la entidad asociada, no pasando la entidad asociada.

El método `EntityRepository#findBy()`, además, acepta ordenamientos, limites y desplazamiento como segundo al cuarto parámetros:

```
<?php
$tenUsers = $em->getRepository('MiProyecto\Domain\User')->findBy(array('age' => 20), array('name' => '
```

Si pasas una matriz de valores, *Doctrine* automáticamente convertirá la consulta en una consulta `WHERE campo IN (..)`:

```
<?php
$users = $em->getRepository('MiProyecto\Domain\User')->findBy(array('age' => array(20, 30, 40)));
// la traduce más o menos en: SELECT * FROM users WHERE age IN (20, 30, 40)
```

Un `EntityRepository` también proporciona un mecanismo para llamadas más concisas usando `__call`. Por lo tanto, los dos siguientes ejemplos son equivalentes:

```
<?php
// Un sólo usuario por sobrenombre
$user = $em->getRepository('MiProyecto\Domain\User')->findOneBy(array('nickname' => 'romanb'));

// Un sólo usuario por sobrenombre (__call magic)
$user = $em->getRepository('MiProyecto\Domain\User')->findOneByNickname('romanb');
```

Por carga impaciente

Cada vez que consultas por una entidad que cuenta con asociaciones persistentes y estas asociaciones se asignaron como `EAGER`, se cargarán automáticamente junto con la entidad consultada y por lo tanto inmediatamente disponibles para tu aplicación.

Por carga diferida

Siempre que tengas a mano una instancia de entidad gestionada, la puedes recorrer y usar las asociaciones de esa entidad que están configuradas `LAZY` como si ya estuvieran en memoria. *Doctrine* cargará automáticamente los objetos asociados bajo demanda a través del concepto de carga diferida.

Por `DQL`

El método de consulta más potente y flexible para objetos persistentes es el lenguaje de consulta de *Doctrine*, un lenguaje de consulta orientado a objetos. `DQL` te permite consultar objetos persistentes en el lenguaje de objetos. `DQL` comprende clases, campos, herencia y asociaciones. `DQL` sintácticamente es muy similar al familiar `SQL`, pero *no es SQL*.

Una consulta DQL es representada por una instancia de la clase Doctrine\ORM\Query. Tú creas una consulta utilizando EntityManager#CreateQuery($DQL). He aquí un ejemplo simple:

```
<?php
// $em instancia del EntityManager

// Todos los usuarios con una edad entre 20 y 30 años (inclusive).
$q = $em->createQuery("select u from MyDomain\Model\User u where u.age >= 20 and u.age <= 30");
$users = $q->getResult();
```

Ten en cuenta que esta consulta no contiene ningún conocimiento sobre el esquema relacional, sólo en los objetos del modelo. DQL es compatible con parámetros posicionales, así como nombrados, muchas funciones, uniones (fetch), agregadas, subconsultas y mucho más. Puedes encontrar información más detallada sobre DQL y su sintaxis, así como la clase *Doctrine* en el *capítulo dedicado*. Para construir programáticamente consultas basadas en condiciones que sólo se conocen en tiempo de ejecución, *Doctrine* ofrece la clase especial Doctrine\ORM\QueryBuilder. Puedes encontrar más información sobre la construcción de consultas con un generador de consultas en el capítulo del *Constructor de consultas*.

Por consultas nativas

Como alternativa a DQL o como un repliegue para las declaraciones especiales SQL nativas puedes utilizar. Las consultas nativas se construyen usando una consulta SQL a mano y un ResultSetMapping que describe como debe transformar *Doctrine* el conjunto de resultados SQL. Puedes encontrar más información sobre las consultas nativas en el capítulo dedicado.

Repositorios personalizados

Por omisión, el EntityManager devuelve una implementación predeterminada de Doctrine\ORM\EntityRepository cuando invocas a EntityManager#getRepository($entityClass). Puedes redefinir este comportamiento especificando el nombre de clase de tu propio repositorio de Entidad en metadatos Anotación, XML o *YAML*. En aplicaciones de gran tamaño que requieren gran cantidad de consultas DQL especializadas, utilizar un repositorio personalizado es una forma recomendada de agrupar estas consultas en un lugar central.

```
<?php
namespace MyDomain\Model;

use Doctrine\ORM\EntityRepository;

/**
 * @entity(repositoryClass="MyDomain\Model\UserRepository")
 */
class User
{

}

class UserRepository extends EntityRepository
{
    public function getAllAdminUsers()
    {
        return $this->_em->createQuery('SELECT u FROM MyDomain\Model\User u WHERE u.status = "admin"'
                          ->getResult();
    }
}
```

Ahora, puedes acceder a tu repositorio llamando a:

```
<?php
// $em instancia del EntityManager

$admins = $em->getRepository('MyDomain\Model\User')->getAllAdminUsers();
```

1.9 Trabajando con asociaciones

Las asociaciones entre entidades se representan como en *PHP* regular orientado a objetos, con referencias a otros objetos o colecciones de objetos. Cuando se trata de la persistencia, es importante entender tres cosas principales:

- El *concepto del lado propietario e inverso* en asociaciones bidireccionales.
- Si la entidad es removida de una colección, la asociación se retira, no la propia entidad. Una colección de entidades siempre representa únicamente la asociación a las entidades que contiene, no a la propia entidad.
- Colección valuada, los *campos persistentes* tienen que ser instancias de la interfaz `Doctrine\Common\Collections\Collection`.

Los cambios a las asociaciones en tu código no se sincronizan con la base de datos directamente, sino al llamar a `EntityManager#flush()`.

Para describir todos los conceptos de trabajar con asociaciones introduciremos un conjunto específico de entidades de ejemplo que muestran todos los distintos sabores de la gestión de asociaciones en *Doctrine*.

1.9.1 Ejemplo de asociación de entidades

Vamos a utilizar un sistema de comentarios simple con Usuarios y Comentarios como entidades para mostrar ejemplos de gestión de la asociación. Ve los `docblocks PHP` de cada asociación en el siguiente ejemplo para información sobre su tipo y si es el lado propietario o inverso.

```
<?php
/** @Entity */
class User
{
    /** @Id @GeneratedValue @Column(type="string") */
    private $id;

    /**
     * Bidireccional - Muchos usuarios tienen muchos comentarios favoritos (Lado propietario)
     *
     * @ManyToMany(targetEntity="Comment", inversedBy="userFavorites")
     * @JoinTable(name="user_favorite_comments",
     *   joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *   inverseJoinColumns={@JoinColumn(name="favorite_comment_id", referencedColumnName="id")}
     * )
     */
    private $favorites;

    /**
     * Unidireccional - Muchos usuarios han marcado muchos comentarios como leídos
     *
     * @ManyToMany(targetEntity="Comment")
     * @JoinTable(name="user_read_comments",
     *   joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
```

```
     *   inverseJoinColumns={@JoinColumn(name="comment_id", referencedColumnName="id")}
     * )
     */
    private $commentsRead;

    /**
     * Bidireccional - Uno-A-Muchos (Lado inverso)
     *
     * @OneToMany(targetEntity="Comment", mappedBy="author")
     */
    private $commentsAuthored;

    /**
     * Unidireccional - Muchos-A-Uno
     *
     * @ManyToOne(targetEntity="Comment")
     */
    private $firstComment;
}

/** @Entity */
class Comment
{
    /** @Id @GeneratedValue @Column(type="string") */
    private $id;

    /**
     * Bidireccional - Muchos comentarios son favoritos de muchos usuarios (Lado inverso)
     *
     * @ManyToMany(targetEntity="User", mappedBy="favorites")
     */
    private $userFavorites;

    /**
     * Bidireccional - Muchos comentarios fueron redactados por un usuario (Lado propietario)
     *
     * @ManyToOne(targetEntity="User", inversedBy="authoredComments")
     */
     private $author;
}
```

Estas dos entidades generarán el siguiente esquema MySQL (omitiendo la definición de claves externas):

```
CREATE TABLE User (
    id VARCHAR(255) NOT NULL,
    firstComment_id VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE Comment (
    id VARCHAR(255) NOT NULL,
    author_id VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE user_favorite_comments (
    user_id VARCHAR(255) NOT NULL,
    favorite_comment_id VARCHAR(255) NOT NULL,
    PRIMARY KEY(user_id, favorite_comment_id)
```

```
) ENGINE = InnoDB;

CREATE TABLE user_read_comments (
    user_id VARCHAR(255) NOT NULL,
    comment_id VARCHAR(255) NOT NULL,
    PRIMARY KEY(user_id, comment_id)
) ENGINE = InnoDB;
```

1.9.2 Estableciendo asociaciones

Crear una asociación entre dos entidades es directo. Estos son algunos ejemplos de las relaciones unidireccionales del `Usuario`:

```
<?php
class User
{
    // ...
    public function getReadComments() {
         return $this->commentsRead;
    }

    public function setFirstComment(Comment $c) {
        $this->firstComment = $c;
    }
}
```

El código de interacción se debería ver como en el siguiente fragmento (`$em` He aquí un ejemplo del EntityManager):

```
<?php
$user = $em->find('User', $userId);
// unidireccional muchos a muchos
$comment = $em->find('Comment', $readCommentId);
$user->getReadComments()->add($comment);

$em->flush();

// unidireccional muchos a uno
$myFirstComment = new Comment();
$user->setFirstComment($myFirstComment);

$em->persist($myFirstComment);
$em->flush();
```

En el caso de las asociaciones bidireccionales tienes que actualizar los campos en ambos lados:

```
<?php
class User
{
    // ..

    public function getAuthoredComments() {
        return $this->commentsAuthored;
    }

    public function getFavoriteComments() {
        return $this->favorites;
    }
```

```
}

class Comment
{
    // ...

    public function getUserFavorites() {
        return $this->userFavorites;
    }

    public function setAuthor(User $author = null) {
        $this->author = $author;
    }
}

// Muchos-A-Muchos
$user->getFavorites()->add($favoriteComment);
$favoriteComment->getUserFavorites()->add($user);

$em->flush();

// Muchos-A-Uno / Uno-A-Muchos Bidireccional
$newComment = new Comment();
$user->getAuthoredComments()->add($newComment);
$newComment->setAuthor($user);

$em->persist($newComment);
$em->flush();
```

Observa cómo siempre se actualizan ambos lados de la asociación bidireccional. Las asociaciones unidireccionales anteriores eran más sencillas de manejar.

1.9.3 Removiendo asociaciones

La remoción de una asociación entre dos entidades es igualmente sencilla. Hay dos estrategias para hacerlo, por clave y por elemento. He aquí algunos ejemplos:

```
<?php
// Remueve por Elementos
$user->getComments()->removeElement($comment);
$comment->setAuthor(null);

$user->getFavorites()->removeElement($comment);
$comment->getUserFavorites()->removeElement($user);

// Remueve por clave
$user->getComments()->remove($ithComment);
$comment->setAuthor(null);
```

Tienes que llamar a `$em->flush()` para que estos cambios se guarden en la base de datos permanentemente.

Observa cómo ambos lados de la asociación bidireccional siempre están actualizados. Las asociaciones unidireccionales, en consecuencia, son más sencillas de manejar. Also note that if you use type-hinting in your methods, i.e. `setAddress(Address $address)`, PHP will only allow null values if `null` is set as default value. De lo contrario `setAddress(null)` fallará para la remoción de la asociación. Si insistes en sugerir el tipo como una manera tradicional para hacer frente a esto es proporcionar un método especial, como `removeAddress()`. Esto también te puede proporcionar una mejor encapsulación ya que oculta el significado interno de no tener una dirección.

Cuando trabajes con colecciones, ten en cuenta que una colección esencialmente es un mapa ordenado (al igual que un arreglo PHP). Es por eso que la operación `remove` acepta un índice/clave. `removeElement` es un método independiente que tiene *O(n)* complejidad usando `array_search`, donde *n* es el tamaño del mapa.

Nota: Ya que *Doctrine* siempre ve sólo el lado propietario de una asociación bidireccional para las actualizaciones, no es necesario - para las operaciones de escritura - actualizar una colección inversa de una asociación bidireccional uno-a-muchos o muchos-a-muchos. Este conocimiento a menudo se puede utilizar para mejorar el rendimiento evitando la carga de la colección inversa.

También puedes borrar el contenido de una colección completa con el método `Collections::clear()`. Debe tener en cuenta que el uso de este método puede llevar a una base de datos directamente y optimizar la remoción o actualización durante la llamada a la operación de vaciado que no tiene conocimiento de las entidades que se han añadido de nuevo a la colección.

Digamos que borraste una colección de etiquetas llamando a `$post->getTags()->clear();` y luego invocaste a `$post->getTags()->add($tag)`. This will not recognize the tag having already been added previously and will consequently issue two separate database calls.

1.9.4 Métodos para gestionar asociaciones

Generalmente es buena idea encapsular la gestión de la asociación adecuada dentro de las clases Entidad. Esto facilita la utilización de la clase correctamente y puedes encapsular los detalles sobre cómo se mantiene la asociación.

El siguiente código muestra las actualizaciones del Usuario y Comentario del ejemplo anterior que encapsulan gran parte del código de gestión de la asociación:

```
<?php
class User
{
    //...
    public function markCommentRead(Comment $comment) {
        // Collections implement ArrayAccess
        $this->commentsRead[] = $comment;
    }

    public function addComment(Comment $comment) {
        if (count($this->commentsAuthored) == 0) {
            $this->setFirstComment($comment);
        }
        $this->comments[] = $comment;
        $comment->setAuthor($this);
    }

    private function setFirstComment(Comment $c) {
        $this->firstComment = $c;
    }

    public function addFavorite(Comment $comment) {
        $this->favorites->add($comment);
        $comment->addUserFavorite($this);
    }

    public function removeFavorite(Comment $comment) {
        $this->favorites->removeElement($comment);
        $comment->removeUserFavorite($this);
    }
```

```
}

class Comment
{
    // ..

    public function addUserFavorite(User $user) {
        $this->userFavorites[] = $user;
    }

    public function removeUserFavorite(User $user) {
        $this->userFavorites->removeElement($user);
    }
}
```

Notarás que `addUserFavorite` y `removeUserFavorite` no llaman a `addFavorite` y `removeFavorite`, por tanto, estrictamente hablando la asociación bidireccional todavía está incompleta. Sin embargo, si ingenuamente agregas el `addFavorite` en `addUserFavorite`, terminarás con un bucle infinito, por lo tanto necesitas trabajar más. Como puedes ver, el manejo adecuado de la asociación bidireccional en programación orientada a objetos simple es una tarea no trivial y resumir todos los detalles dentro de las clases puede ser un reto.

Nota: Si deseas asegurarte de que tus colecciones están perfectamente encapsuladas no debes regresar de un método `getCollectionName()` directamente, sino llamar a `$colección->toArray()`. De esta manera un programador cliente de la entidad no puede eludir la lógica que implementes en la entidad para gestionar la asociación. Por ejemplo:

```
<?php
class User {
    public function getReadComments() {
        return $this->commentsRead->toArray();
    }
}
```

Sin embargo, esto siempre inicia la colección, con todas los contratiempos de rendimiento, dado el tamaño. En algunos escenarios de grandes colecciones incluso podría ser una buena idea ocultar completamente el acceso de lectura detrás de los métodos del `EntityRepository`.

No hay una sola, mejor manera para gestionar la asociación. Esto también depende en gran medida de las necesidades de tu modelo del dominio, así como de tus preferencias.

1.9.5 Sincronizando colecciones bidireccionales

In the case of Many-To-Many associations you as the developer have the responsibility of keeping the collections on the owning and inverse side

> in sync when you apply changes to them. *Doctrine* sólo puede garantizar un estado consistente para la hidratación, no para tu código cliente.

Utilizando las entidades Usuario-Comentarios anterior, un ejemplo muy simple puede mostrar las posibles advertencias que te puedes encontrar:

```
<?php
$user->getFavorites()->add($favoriteComment);
// no llama a $favoriteComment->getUserFavorites()->add($user);
```

```
$user->getFavorites()->contains($favoriteComment); // TRUE
$favoriteComment->getUserFavorites()->contains($user); // FALSE
```

Hay dos enfoques para abordar este problema en tu código:

1. No hagas caso de la actualización del lado inverso de las colecciones bidireccionales, PERO nunca las leas en las peticiones que cambian su estado. En la próxima petición *Doctrine* hidrata consistentemente de nuevo el estado de la colección.
2. Siempre mantén sincronizadas las colecciones bidireccionales con los métodos que gestionan asociaciones. Entonces lee de las colecciones directamente después de que los cambios sean consistentes.

1.9.6 Persistencia transitiva / operaciones en cascada

Persistiendo, removiendo, disociando y fusionando las entidades individuales puede llegar a ser bastante engorroso, especialmente cuando un objeto gráfico muy pesado está involucrado. Por lo tanto *Doctrine* 2 proporciona un mecanismo para la persistencia transitiva a través de estas operaciones en cascada. Puedes configurar cada asociación a otra entidad o conjunto de entidades para automáticamente propagar en cascada ciertas operaciones. Por omisión, no hay ninguna operación en cascada.

Existen las siguientes opciones en cascada:

- `persist` : Propaga en cascada las operaciones de persistencia a las entidades asociadas.
- `remove` : Propaga en cascada las operaciones de remoción a las entidades asociadas.
- `merge` : Propaga en cascada las operaciones de fusión a las entidades asociadas.
- `detach` : Propaga en cascada las operaciones de disociación a las entidades asociadas.
- `all` : Propaga en cascada las operaciones de persistencia, remoción, fusión y disociación a todas las entidades asociadas.

Nota: Las operaciones de propagación en cascada se realizan en memoria. Esto significa que las colecciones y entidades relacionadas se recuperan en memoria, incluso si todavía están marcadas como diferidas cuando la operación en cascada está a punto de llevarse a cabo. Sin embargo, este enfoque permite que los eventos del ciclo de vida de la entidad se realizan para cada una de estas operaciones.

Sin embargo, arrastrar objetos gráficos en la memoria de cascada puede causar una considerable sobrecarga de rendimiento, sobre todo cuando son grandes colecciones en cascada. Asegúrate de sopesar las ventajas y desventajas de cada operación en cascada que definas.

Para confiar en las operaciones en cascada a nivel de la base de datos para la operación de remoción, en su lugar, puedes configurar cada columna de la unión con la opción `onDelete`. Consulta los capítulos respectivos de los controladores de asignación para más información.

El siguiente ejemplo es una extensión del ejemplo Usuario-Comentarios de este capítulo. Supongamos que en nuestra aplicación se crea un usuario cuando escribe su primer comentario. En este caso deberemos usar el siguiente código:

```
<?php
$user = new User();
$myFirstComment = new Comment();
$user->addComment($myFirstComment);

$em->persist($user);
$em->persist($myFirstComment);
$em->flush();
```

Incluso si *persistes* un nuevo Usuario que contiene nuestro nuevo Comentario este código fallará si eliminas la llamada a `EntityManager#persist($myFirstComment)`. *Doctrine* 2 no persiste la operación en cascada a todas las entidades anidadas que son nuevas también.

More complicated is the deletion of all of a user's comments when he is removed from the system:

```
$user = $em->find('User', $deleteUserId);

foreach ($user->getAuthoredComments() AS $comment) {
    $em->remove($comment);
}
$em->remove($user);
$em->flush();
```

Sin el bucle sobre todos los comentarios redactados, *Doctrine* utiliza una instrucción `UPDATE` sólo para establecer la clave externa a `NULL` y sólo el Usuario será borrado de la base de datos durante la operación de vaciado.

Para lograr que *Doctrine* maneje ambos casos automáticamente puedes cambiar la propiedad `User#commentsAuthored` a cascada tanto en la operación de "persistencia" como en la operación de "remoción".

```
<?php
class User
{
    //...
    /**
     * Bidireccional - Uno-A-Muchos (Lado inverso)
     *
     * @OneToMany(targetEntity="Comment", mappedBy="author", cascade={"persist", "remove"})
     */
    private $commentsAuthored;
    //...
}
```

A pesar de que se propaga en cascada automáticamente es conveniente la utilices con sumo cuidado. No apliques `cascade=all` ciegamente en a todas las asociaciones, puesto que innecesariamente degradará el rendimiento de tu aplicación. Para cada operación en cascada que se activa *Doctrine* también aplica la operación de la asociación, ya sea sola o una colección valorada.

Persistencia por accesibilidad: Persistencia en cascada

Hay una semántica adicional que se aplica a la operación de persistencia en cascada. Durante cada operación `flush()` *Doctrine* detecta si hay nuevas entidades en la colección y pueden ocurrir tres posibles casos:

1. Nuevas entidades en una colección marcada como persistencia en cascada mantenidas por *Doctrine* directamente.
2. Nuevas entidades en una colección que no están marcadas como persistencia en cascada producirán una Excepción y la operación `flush()` las restaurará.
3. Se omiten las colecciones sin nuevas entidades.

Este concepto se denomina Persistencia por accesibilidad: Nuevas entidades que se encuentran en las entidades que ya están gestionadas automáticamente, siempre y cuando se persistió la asociación definida como persistencia en cascada.

1.9.7 Remoción huérfana

Hay otro concepto de cascada que sólo es relevante cuando remueves entidades de colecciones. Si una Entidad del tipo `A` contiene referencias a propiedades privadas de las Entidades `B`, entonces, si la referencia de `A` a `B` se retira de la Entidad `B`, también dr debe eliminar, porque ya no se utiliza.

La remoción huérfana funciona tanto con asociaciones uno-a-uno como con uno-a-muchos.

Nota: Cuando usas la opción `orphanRemoval=true` *Doctrine* hace la asunción de que las entidades son de propiedad privada y **NO** se deben reutilizar por otras entidades. Si descuidas este supuesto tus entidades serán eliminadas por *Doctrine*, incluso si has asignado la entidad huérfana a otra.

Como un mejor ejemplo considera una aplicación Libreta de direcciones, donde tienes Contactos, Direcciones e Información adicional:

```
<?php

namespace Addressbook;

use Doctrine\Common\Collections\ArrayCollection;

/**
 * @Entity
 */
class Contact
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;

    /** @OneToOne(targetEntity="StandingData", orphanRemoval=true) */
    private $standingData;

    /** @OneToMany(targetEntity="Address", mappedBy="contact", orphanRemoval=true) */
    private $addresses;

    public function __construct()
    {
        $this->addresses = new ArrayCollection();
    }

    public function newStandingData(StandingData $sd)
    {
        $this->standingData = $sd;
    }

    public function removeAddress($pos)
    {
        unset($this->addresses[$pos]);
    }
}
```

Now two examples of what happens when you remove the references:

```
<?php

$contact = $em->find("Addressbook\Contact", $contactId);
$contact->newStandingData(new StandingData("Firstname", "Lastname", "Street"));
$contact->removeAddress(1);
```

```
$em->flush();
```

In this case you have not only changed the `Contact` entity itself but you have also removed the references for standing data and as well as one address reference. When flush is called not only are the references removed but both the old standing data and the one address entity are also deleted from the database.

1.10 Transacciones y concurrencia

1.10.1 Demarcando transacciones

La demarcación de transacciones es la tarea de definir los límites de tu transacción. La adecuada demarcación de transacción es muy importante porque si no se hace correctamente puede afectar negativamente el rendimiento de tu aplicación. Muchas bases de datos y capas de abstracción de la base de datos como PDO de manera predeterminada funcionan en modo autoconsigna, lo cual significa que cada declaración `SQL` individual está envuelta en una pequeña operación. Sin ningún tipo de demarcación de transacción explícito por tu parte, esto se traduce rápidamente en un bajo rendimiento porque las transacciones no son baratas.

En su mayor parte, *Doctrine* 2 ya se encarga de demarcar correctamente las transacciones por ti: Todas las operaciones de escritura (`INSERT/UPDATE/DELETE`) se ponen en la cola hasta que se invoca a `EntityManager#flush()` el cual envuelve todos estos cambios en un sola transacción.

Sin embargo, *Doctrine* 2 también permite (y te estimula) a asumir el control y coordinar tú mismo la demarcación de transacciones.

Estas dos formas de tratar con transacciones cuando utilizas el ORM de *Doctrine* y ahora las describiremos en más detalle.

Enfoque 1: Implícitamente

El primer enfoque es usar la manipulación de transacción implícita proporcionada por el `EntityManager` del ORM de *Doctrine*. Dado el siguiente fragmento de código, sin ningún tipo de demarcación de transacción explícito:

```
<?php
// $em es instancia de EntityManager
$user = new User;
$user->setName('George');
$em->persist($user);
$em->flush();
```

Debido a que no haces ninguna demarcación de transacciones personalizada en el código anterior, `EntityManager#flush()` comenzará y consignará/revertirá una transacción. Este comportamiento es posible gracias a la agregación de las operaciones DML por el ORM de *Doctrine* y es suficiente si toda la manipulación de datos forma parte de una unidad de trabajo que pasa a través del modelo de dominio y por lo tanto el ORM.

Enfoque 2: Explícitamente

La alternativa explícita usa directamente la *API* `Doctrine\DBAL\Connection` para controlar los límites de la transacción. El código tendrá el siguiente aspecto:

```
<?php
// $em es instancia de EntityManager
$em->getConnection()->beginTransaction(); // suspende la consignación automática
try {
```

```
    //... realiza alguna tarea
    $user = new User;
    $user->setName('George');
    $em->persist($user);
    $em->flush();
    $em->getConnection()->commit();
} catch (Exception $e) {
    $em->getConnection()->rollback();
    $em->close();
    throw $e;
}
```

La demarcación de transacción explícita es necesaria cuando deseas incluir operaciones DBAL personalizadas en una unidad de trabajo o cuando deseas utilizar algunos métodos de la *API* del `EntityManager` que requieren una transacción activa. Tales métodos lanzarán una `TransactionRequiredException` para informarte de este requisito.

Una alternativa más conveniente para demarcar transacciones explícitamente es usando las abstracciones de control provistas en forma de `Connection#transactional($func)` y `EntityManager#transactional($func)`. Cuando se usan, estas abstracciones de control aseguran que nunca te olvidas de deshacer la operación o cerrar el `EntityManager`, aparte de la ovbia reducción de código. Un ejemplo que funcionalmente es equivalente al código mostrado anteriormente es el siguiente:

```
<?php
// $em es instancia de EntityManager
$em->transactional(function($em) {
    //... realiza alguna tarea
    $user = new User;
    $user->setName('George');
    $em->persist($user);
});
```

La diferencia entre `Connection#transactional($func)` y `EntityManager#transactional($func)` es que la capa de abstracción vacía el segundo `EntityManager` antes de consignar la transacción y cierra el `EntityManager` correctamente cuando se produce una excepción (además de revertir la transacción).

Manejando excepciones

Cuando utilizas la demarcación de transacción implícita y se produce una excepción durante el `EntityManager#flush()`, la transacción se revierte automáticamente y se cierra el `EntityManager`.

Cuando utilizas la demarcación de transacciones explícita y se produce una excepción, la transacción se debe revertir de inmediato y cerrar el `EntityManager` invocando a `EntityManager#close()` y subsecuentemente desecharlos, como muestra el ejemplo anterior. Esto lo puedes manejar elegantemente con las abstracciones de control mostradas anteriormente. Ten en cuenta que cuando se captura la `Excepción` generalmente debes volver a lanzar la excepción. Si tienes la intención de recuperarte de algunas excepciones, las tienes que capturar explícitamente en los anteriores bloques `catch` (pero no te olvides revertir la operación y cerrar el `EntityManager` allí también). Todas las otras buenas prácticas del manejo de excepciones se aplican de manera similar (es decir, ya sea registrandolas o relanzandolas, ninguna de las dos, etc.)

Como resultado de este procedimiento, todas las instancias gestionadas previamente o instancias removidas del `EntityManager` se vuelven disociadas. El estado de los objetos disociados será el estado en el punto en el que se revirtió la operación. El estado de los objetos de ninguna manera se revierte y por lo tanto los objetos están desincronizados con la base de datos. La aplicación puede seguir utilizando los objetos disociados, a sabiendas de que su estado potencialmente ya no es exacto.

Si tienes intención de iniciar una nueva unidad de trabajo después de que se ha producido una excepción lo debes hacer con un nuevo `EntityManager`.

1.10.2 Apoyando el bloqueo

Doctrine 2 ofrece apoyo nativo a las estrategias de bloqueo pesimista y optimista. Esto te permite tomar el control finamente granular sobre qué tipo de bloqueo es necesario para las entidades en tu aplicación.

Bloqueo optimista

Las transacciones de bases de datos están muy bien para controlar la concurrencia durante una sola petición. Sin embargo, una transacción de base de datos no debe abarcar todas las peticiones, el así llamado "tiempo para pensar del usuario". Por lo tanto, una larga "transacción del negocio", que abarca múltiples peticiones, tiene que implicar varias transacciones de base de datos. Por lo tanto, las transacciones de bases de datos solo pueden controlar la concurrencia durante una larga transacción del negocio. El control de concurrencia se convierte en parte de la responsabilidad de la propia aplicación.

Doctrine tiene integrada la capacidad de bloqueo optimista automático a través de un campo versión. En este enfoque, cualquier entidad que se deba proteger contra modificaciones simultáneas en largas transacciones del negocio obtienes un campo versión el cual es un número simple (tipo de asignación: entero) o una marca de tiempo (correspondiente al tipo: fecha y hora - `datetime`). Cuando los cambios a dicha entidad se persistan al final de una larga conversación la versión de la entidad es comparada con la versión de la base de datos y si no coinciden, se lanzará una `OptimisticLockException`, indicando que la entidad ya se ha modificado por alguien más.

Puedes designar un campo de versión en una entidad de la siguiente manera. En este ejemplo vamos a utilizar un número entero.

```
<?php
class User
{
    // ...
    /** @Version @Column(type="integer") */
    private $version;
    // ...
}
```

Alternativamente, puedes utilizar un tipo `datetime` (el cual corresponde a un `timestamp` o `datetime` de `SQL`):

```
<?php
class User
{
    // ...
    /** @Version @Column(type="datetime") */
    private $version;
    // ...
}
```

Sin embargo, son preferibles los números de versión (no marcas de tiempo) ya que no pueden entrar en conflicto en un entorno altamente concurrente, a diferencia de las marcas de tiempo en que esta es una posibilidad, dependiendo de la resolución de fecha y hora en la plataforma de base de datos particular.

Cuando se encuentra un conflicto de versión en `EntityManager#flush()`, una `OptimisticLockException` es lanzada y la transacción activa es revertida (o marcada para reversión). Puedes capturar y manejar esta excepción. Las posibles respuestas a una `OptimisticLockException` han de presentar el conflicto al usuario o actualizar ‘y volver a cargar los objetos en una nueva transacción y luego volver a intentar la operación.

Con PHP promoviendo una arquitectura de no compartición, el tiempo entre que muestra una actualización del formulario y modifica realmente la entidad puede - en el peor de los casos - ser tan largo como el tiempo de espera de sesión de tus aplicaciones. Si los cambios ocurren a la entidad en ese tiempo quieres saber directamente cuando recuperar la entidad que alcanzó una excepción de bloqueo optimista:

Siempre puedes verificar la versión de una entidad durante una petición, ya sea cuando llamas a `EntityManager#find()`:

```
<?php
use Doctrine\DBAL\LockMode;
use Doctrine\ORM\OptimisticLockException;

$theEntityId = 1;
$expectedVersion = 184;

try {
    $entity = $em->find('User', $theEntityId, LockMode::OPTIMISTIC, $expectedVersion);

    // realiza alguna tarea

    $em->flush();
} catch(OptimisticLockException $e) {
    echo "Sorry, but someone else has already changed this entity. Please apply the changes again!";
}
```

O puedes usar `EntityManager#lock()` para averiguarlo:

```
<?php
use Doctrine\DBAL\LockMode;
use Doctrine\ORM\OptimisticLockException;

$theEntityId = 1;
$expectedVersion = 184;

$entity = $em->find('User', $theEntityId);

try {
    // afirma la versión
    $em->lock($entity, LockMode::OPTIMISTIC, $expectedVersion);

} catch(OptimisticLockException $e) {
    echo "Sorry, but someone else has already changed this entity. Please apply the changes again!";
}
```

Notas importantes de implementación

Puedes conseguir fácilmente el flujo de trabajo del bloqueo optimista incorrecto si comparas las versiones incorrectas. Digamos que Alicia y Bob necesitan acceder a una hipotética cuenta bancaria:

- Alicia, lee el titular de la entrada en el blog “Foo”, en la versión de bloqueo optimista 1 (petición GET)
- Bob, lee el titular de la entrada en el blog “Foo”, en la versión de bloqueo optimista 1 (petición GET)
- Bob actualiza el título a “Bar”, actualizando la versión del bloqueo optimista a 2 (petición POST de un formulario)
- Alicia actualiza el título a “Baz”, ... (petición POST de un formulario)

Ahora bien, en la última etapa de este escenario, la entrada en el blog se tiene que leer de nuevo desde la base de datos antes de poder aplicar el titular de Alicia. En este punto, quieres comprobar si el blog sigue en la versión 1 (el cual no es este escenario).

Usando el bloqueo optimista correctamente, *tienes* que añadir la versión como un campo oculto adicional (o en la Sesión para mayor seguridad). De lo contrario, no puedes verificar si la versión sigue siendo la que se leyó al principio desde la base de datos cuando Alicia realizó su petición GET de la entrada en el blog. Si esto sucede, puedes ver la pérdida de actualizaciones que querías evitar con el bloqueo optimista.

Ve el código de ejemplo, el formulario (petición GET):

```
<?php
$post = $em->find('BlogPost', 123456);

echo '<input type="hidden" name="id" value="' . $post->getId() . '" />';
echo '<input type="hidden" name="version" value="' . $post->getCurrentVersion() . '" />';
```

Y la acción de cambio de titular (petición POST):

```
<?php
$postId = (int)$_GET['id'];
$postVersion = (int)$_GET['version'];

$post = $em->find('BlogPost', $postId, \Doctrine\DBAL\LockMode::OPTIMISTIC, $postVersion);
```

Bloqueo pesimista

Doctrine 2 es compatible con el bloqueo pesimista a nivel de base de datos. No estás tratando de implementar el bloqueo pesimista dentro de *Doctrine*, más bien específicas tu proveedor y las ordenes `ANSI-SQL` utilizadas para adquirir el bloqueo a nivel de fila. Cada entidad puede ser parte de un bloqueo pesimista, no hay necesidad de metadatos especiales para utilizar esta característica.

Sin embargo, para que el bloqueo pesimista trabaje tienes que desactivar el modo de consignación automática de tu base de datos e iniciar una transacción en torno a las instancias de tu bloqueo pesimista con el "Enfoque 2: Demarcación de transacción explícita" descrito anteriormente. *Doctrine* 2 lanzará una excepción si intentas adquirir un bloqueo pesimista y no hay ninguna transacción en marcha.

Doctrine 2 actualmente es compatible con dos modos de bloqueo pesimista:

- Escritura pesimista (`Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE`), bloquea las filas de la base de datos subyacente para operaciones simultáneas de lectura y escritura.
- Lectura pesimista (`Doctrine\DBAL\LockMode::PESSIMISTIC_READ`), bloquea otras peticiones simultáneas que intenten actualizar o bloquear filas en modo de escritura.

Puedes usar bloqueos pesimistas en tres diferentes escenarios:

1. Usando `EntityManager#find($className, $id, \Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE)` o `EntityManager#find($className, $id, \Doctrine\DBAL\LockMode::PESSIMISTIC_READ)`
2. Usando `EntityManager#lock($entity, \Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE)` o `EntityManager#lock($entity, \Doctrine\DBAL\LockMode::PESSIMISTIC_READ)`
3. Usando `Query#setLockMode(\Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE)` o `Query#setLockMode(\Doctrine\DBAL\LockMode::PESSIMISTIC_READ)`

1.11 Eventos

Doctrine 2 cuenta con un sistema de eventos ligero que es parte del paquete Común.

1.11.1 El sistema de eventos

El sistema de eventos está controlado por el `EventManager`. Es el punto central del sistema de escucha de eventos de *Doctrine*. Los escuchas están registrados en el gestor de eventos y se envían a través del gestor.

```
<?php
$evm = new EventManager();
```

Ahora podemos añadir algunos escuchas de eventos al `$evm`. Vamos a crear una clase `EventTest` para jugar.

```
<?php
class EventTest
{
    const preFoo = 'preFoo';
    const postFoo = 'postFoo';

    private $_evm;

    public $preFooInvoked = false;
    public $postFooInvoked = false;

    public function __construct($evm)
    {
        $evm->addEventListener(array(self::preFoo, self::postFoo), $this);
    }

    public function preFoo(EventArgs $e)
    {
        $this->preFooInvoked = true;
    }

    public function postFoo(EventArgs $e)
    {
        $this->postFooInvoked = true;
    }
}

// Crea una nueva instancia
$test = new EventTest($evm);
```

Puedes despachar eventos utilizando el método `dispatchEvent()`.

```
<?php
$evm->dispatchEvent(EventTest::preFoo);
$evm->dispatchEvent(EventTest::postFoo);
```

Puedes eliminar un escucha con el método `removeEventListener()`.

```
<?php
$evm->removeEventListener(array(self::preFoo, self::postFoo), $this);
```

El sistema de eventos de *Doctrine* 2, además, tiene un concepto simple de los suscriptores de eventos. Podemos definir una simple clase `TestEventSubscriber` que implemente la interfaz

`\Doctrine\Common\EventSubscriber` e implemente un método `getSubscribedEvents()` que devuelva un arreglo de eventos a los que debe estar suscrito.

```
<?php
class TestEventSubscriber implements \Doctrine\Common\EventSubscriber
{
    public $preFooInvoked = false;

    public function preFoo()
    {
        $this->preFooInvoked = true;
    }

    public function getSubscribedEvents()
    {
        return array(TestEvent::preFoo);
    }
}

$eventSubscriber = new TestEventSubscriber();
$evm->addEventSubscriber($eventSubscriber);
```

Ahora, cuando despaches un evento cualquier suscriptor del evento será notificado de ese evento.

```
<?php
$evm->dispatchEvent(TestEvent::preFoo);
```

Ahora puedes probar la instancia de `$eventSubscriber` para ver si el método `preFoo()` fue invocado.

```
<?php
if ($eventSubscriber->preFooInvoked) {
    echo 'pre foo invoked!';
}
```

Convención de nomenclatura

Los eventos utilizados con el `EventManager` de *Doctrine* 2 se nombran mejor con mayúsculas intercaladas y el valor de la constante correspondiente debe ser el nombre de la misma constante, incluso con la misma ortografía. Esto tiene varias razones:

- Es fácil de leer.
- Simplicidad.
- Cada método dentro de un `EventSubscriber` lleva el nombre de la constante correspondiente. Si el nombre de la constante y valor constante difieren, debes utilizar el nuevo valor y por lo tanto, el código podría ser objeto de cambios de código cuando cambia el valor. Esto contradice la intención de una constante.

Puedes encontrar un ejemplo de una anotación correcta en el `EventTest` anterior.

1.11.2 Ciclo de vida de los eventos

El `EntityManager` y la `UnitOfWork` desencadenan un montón de eventos durante el tiempo de vida de sus entidades registradas.

- `preRemove` - El evento `preRemove` lo produce una determinada entidad antes de ejecutar la operación de remoción del respectivo `EntityManager` de esa entidad. Este no es llamado por una declaración `DELETE` de `DQL`.

- `postRemove` - El evento `postRemove` ocurre por una entidad, después de haber borrado la entidad. Este se debe invocar después de las operaciones de eliminación de la base de datos. Este no es llamado por una declaración `DELETE` de `DQL`.
- `prePersist` - El evento `prePersist` se produce por una determinada entidad antes de que el `EntityManager` respectivo ejecute las operaciones de persistencia de esa entidad.
- `postPersist` - El evento `postPersist` ocurre por una entidad, después de hacer permanente la entidad. Este se debe invocar después de las operaciones de inserción de la base de datos. Genera valores de clave primaria disponibles en el evento `postPersist`.
- `preUpdate` - El evento `preUpdate` ocurre antes de que las operaciones de actualización de datos de la entidad pasen a la base de datos. No es llamada por una declaración `UPDATE` de `DQL`.
- `postUpdate` - El evento `postUpdate` se produce después de las operaciones de actualización de datos de la entidad en la base de datos. No es llamada por una declaración `UPDATE` de `DQL`.
- `postLoad` - El evento `postLoad` ocurre para una entidad, después que la entidad se ha cargado en el `EntityManager` actual de la base de datos o después de que la operación de actualización se ha aplicado a la misma.
- `loadClassMetadata` - El evento `loadClassMetadata` se produce después de que los metadatos de asignación para una clase se han cargado desde una fuente de asignación (anotaciones/xml/yaml).
- `onFlush` - El evento `onFlush` ocurre después de computar el conjunto de cambios de todas las entidades gestionadas. Este evento no es un ciclo de vida de la retrollamada.

Advertencia: Ten en cuenta que el evento `postLoad` ocurre para una entidad antes de haber iniciado las asociaciones. Por lo tanto, no es seguro acceder a las asociaciones en una retrollamada `postLoad` o controlador del evento.

Puedes acceder a las constantes del Evento desde la clase `Events` en el paquete ORM.

```
<?php
use Doctrine\ORM\Events;
echo Events::preUpdate;
```

Estos se pueden conectar a dos diferentes tipos de escuchas de eventos:

- El ciclo de vida de las retrollamadas son métodos de las clases entidad que se llaman cuando se lanza el evento. Ellos no reciben absolutamente ningún argumento y están diseñados específicamente para permitir cambios en el interior del estado de las clases entidad.
- Los escuchas del ciclo de vida de eventos son clases con métodos de retrollamada específicos que reciben algún tipo de instancia `EventArgs` que dan acceso a la entidad, al `EntityManager` o a otros datos pertinentes.

Nota: Todos los eventos del ciclo de vida que se producen durante el `flush()` de un `EntityManager` tienen restricciones muy específicas sobre las operaciones permitidas que pueden ejecutar. Por favor, lee muy cuidadosamente la sección *Implementando escuchas de eventos* para entender qué operaciones se permiten en qué ciclo de vida del evento.

1.11.3 Ciclo de vida de las retrollamadas

El ciclo de vida de un evento es un evento regular con la característica adicional de proporcionar un mecanismo para registrar las retrollamadas directamente dentro de las clases Entidad correspondientes que se ejecuta cuando se produce el ciclo de vida del evento.

```php
<?php

/** @Entity @HasLifecycleCallbacks */
class User
{
    // ...

    /**
     * @Column(type="string", length=255)
     */
    public $valor;

    /** @Column(name="created_at", type="string", length=255) */
    private $createdAt;

    /** @PrePersist */
    public function doStuffOnPrePersist()
    {
        $this->createdAt = date('Y-m-d H:m:s');
    }

    /** @PrePersist */
    public function doOtherStuffOnPrePersist()
    {
        $this->value = 'changed from prePersist callback!';
    }

    /** @PostPersist */
    public function doStuffOnPostPersist()
    {
        $this->value = 'changed from postPersist callback!';
    }

    /** @PostLoad */
    public function doStuffOnPostLoad()
    {
        $this->value = 'changed from postLoad callback!';
    }

    /** @PreUpdate */
    public function doStuffOnPreUpdate()
    {
        $this->value = 'changed from preUpdate callback!';
    }
}
```

Ten en cuenta que cuando usas anotaciones tienes que aplicar el marcador de anotación `@HasLifecycleCallbacks` en la clase Entidad.

Si deseas registrar el ciclo de vida de la retrollamada desde YAML o XML, lo puedes hacer con el siguiente.

```
User:
  type: entity
  fields:
# ...
    name:
      type: string(50)
  lifecycleCallbacks:
    prePersist: [ doStuffOnPrePersist, doOtherStuffOnPrePersistToo ]
```

```
    postPersist: [ doStuffOnPostPersist ]
```

XML sería algo como esto:

```
<?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
                          /Users/robo/dev/php/Doctrine/doctrine-mapping.xsd">

    <entity name="User">

        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist" method="doStuffOnPrePersist"/>
            <lifecycle-callback type="postPersist" method="doStuffOnPostPersist"/>
        </lifecycle-callbacks>

    </entity>

</doctrine-mapping>
```

Sólo tienes que asegurarte de definir los método públicos `doStuffOnPrePersist()` y `doStuffOnPostPersist()` en tu modelo `User`.

```
<?php
// ...

class User
{
    // ...

    public function doStuffOnPrePersist()
    {
        // ...
    }

    public function doStuffOnPostPersist()
    {
        // ...
    }
}
```

La `key` del `lifecycleCallbacks` es el nombre del método y el valor es el tipo del evento. Los tipos de eventos permitidos son los que figuran en el ciclo de vida de la sección Eventos anterior.

1.11.4 Escuchando el ciclo de vida de los eventos

Los escuchas del ciclo de vida son mucho más poderosos que el sencillo ciclo de vida de las retrollamadas definidos en las clases Entidad. Estos permiten implementar comportamientos reutilizables entre diferentes clases Entidad, sin embargo, requiere un conocimiento mucho más detallado sobre el funcionamiento interno de `EntityManager` y `UnitOfWork`. Por favor, si estás tratando de escribir tu propio escucha, lee cuidadosamente la sección *Implementando escuchas de eventos*.

Para registrar un escucha de eventos tienes que conectarlo al `EventManager` pasado a la fábrica de `EntityManager`:

```
<?php
$eventManager = new EventManager();
$eventManager->addEventListener(array(Events::preUpdate), MyEventListener());
$eventManager->addEventSubscriber(new MyEventSubscriber());

$entityManager = EntityManager::create($dbOpts, $config, $eventManager);
```

También puedes recuperar la instancia del gestor de eventos después de creado el `EntityManager`:

```
<?php
$entityManager->getEventManager()->addEventListener(array(Events::preUpdate), MyEventListener());
$entityManager->getEventManager()->addEventSubscriber(new MyEventSubscriber());
```

1.11.5 Implementando escuchas de eventos

En esta sección se explica qué está y qué no está permitido durante el ciclo de vida de los eventos específicos de la `UnitOfWork`. A pesar de que obtienes el `EntityManager` pasado a todos estos eventos, hay que seguir estas restricciones con mucho cuidado ya que las operaciones en caso de error pueden producir una gran cantidad de errores diferentes, tales como datos inconsistentes y actualizaciones/persistencias/remociones perdidas.

Para los eventos descritos, los cuales también son eventos del ciclo de vida de la retrollamada de las restricciones también se aplican, con la restricción adicional de que no tienes acceso a la API de `EntityManager` o `UnitOfWork` dentro de estos eventos.

prePersist

Hay dos maneras para activar el evento `prePersist`. Una de ellas, obviamente, es cuando llamas a `EntityManager#persist()`. El evento también es propagado en cascada para todas las asociaciones.

Hay otra manera para invocar a `prePersist`, dentro del método `flush()` cuando se calculan los cambios a las asociaciones y esta asociación está marcada como persistente en cascada. Cualquier nueva entidad encontrada durante esta operación también es persistida y se invoca a `prePersist` en ella. Esto se conoce cómo "persistencia por accesibilidad".

En ambos casos, obtienes una instancia de `LifecycleEventArgs` la cual tiene acceso a la entidad y al gestor de la entidad.

Las siguientes restricciones se aplican a `prePersist`:

- Si estás usando un generador de identidad `prePersist` tal como secuencias el valor `id` *No* está disponible en cualquier evento `prePersist`.
- *Doctrine* no reconocerá los cambios realizados en las relaciones en un evento `prePersist` llamado por "accesibilidad" a través de una persistencia en cascada, a menos que uses la *API* interna de `UnitOfWork`. No recomendamos este tipo de operaciones en la persistencia del contexto de accesibilidad, así que esta bajo tu propio riesgo y, posiblemente, con el apoyo de tus pruebas unitarias.

preRemove

El evento `preRemove` es llamado en cada entidad, cuando es pasado al método `EntityManager#remove()`. Este es propagado en cascada para todas las asociaciones que se han marcado como remoción en cascada.

No hay restricciones a los métodos que se pueden llamar dentro del evento `preRemove`, excepto cuando el método `remove` en sí fue llamado durante una operación de vaciado.

onFlush

`OnFlush` es un evento muy poderoso. Se llama dentro de `EntityManager#flush()` después de gestionar los cambios de todas las entidades y haber calculado sus asociaciones. Esto significa que el evento `onFlush` tiene acceso a los grupos de:

- Entidades programadas para inserción
- Entidades programadas para actualización
- Entidades programadas para remoción
- Colecciones programadas para actualización
- Colecciones programadas para remoción

Para utilizar el evento `onFlush` tienes que estar familiarizado con la *API* interna de `UnitOfWork`, el cual te otorga acceso a los conjuntos antes mencionados. Ve este ejemplo:

```
<?php
class FlushExampleListener
{
    public function onFlush(OnFlushEventArgs $eventArgs)
    {
        $em = $eventArgs->getEntityManager();
        $uow = $em->getUnitOfWork();

        foreach ($uow->getScheduledEntityInsertions() AS $entity) {

        }

        foreach ($uow->getScheduledEntityUpdates() AS $entity) {

        }

        foreach ($uow->getScheduledEntityDeletions() AS $entity) {

        }

        foreach ($uow->getScheduledCollectionDeletions() AS $col) {

        }

        foreach ($uow->getScheduledCollectionUpdates() AS $col) {

        }
    }
}
```

Las siguientes restricciones se aplican a los eventos `onFlush`:

- Llamar a `EntityManager#persist()` no es suficiente para desencadenar la persistencia en una entidad. Tienes que ejecutar una llamada adicional a `$UnitOfWork->computeChangeSet($classMetadata, $entidad)`.
- El cambio de campos primitivos o asociaciones requiere activar explícitamente un nuevo cálculo de los cambios de la entidad afectada. Esto se puede hacer ya sea llamando a `$UnitOfWork->computeChangeSet($classMetadata, $entidad)` o a `$UnitOfWork->recomputeSingleEntityChangeSet($classMetadata, $entidad)`. El segundo método tiene menor costo, pero sólo vuelve a calcular los campos primitivos, nunca las asociaciones.

preUpdate

`PreUpdate` es el más restrictivo para usar eventos, ya que se llama justo antes de una instrucción de actualización, es llamado por una entidad dentro del método `EntityManager#flush()`.

En este caso, nunca se permiten cambios a las asociaciones de la entidad actualizada, ya que *Doctrine* no puede garantizar el manejo correcto de la integridad referencial en este momento de la operación de vaciado. Sin embargo, este evento tiene una poderosa característica, es ejecutado con una instancia de `PreUpdateEventArgs`, la cual contiene una referencia al conjunto de cambios calculado de esta entidad.

Esto significa que tienes acceso a todos los campos que han cambiado para esta entidad con los antiguos y nuevos valores. Los siguientes métodos están disponibles en el `PreUpdateEventArgs`:

- `getEntity()` para tener acceso a la entidad real.
- `getEntityChangeSet()` para obtener una copia de la matriz de cambios. Los cambios en esta matriz devuelta no afectan a la actualización.
- `HasChangedField($fieldName)` para comprobar si cambió el nombre del campo dado de la entidad actual.
- `getOldValue($fieldName)` y `getNewValue($fieldName)` para acceder a los valores de un campo.
- `setNewValue($fieldName, $valor)` para cambiar el valor de un campo que se actualizará.

Un sencillo ejemplo de este evento se ve así:

```
<?php
class NeverAliceOnlyBobListener
{
    public function preUpdate(PreUpdateEventArgs $eventArgs)
    {
        if ($eventArgs->getEntity() instanceof User) {
            if ($eventArgs->hasChangedField('name') && $eventArgs->getNewValue('name') == 'Alice') {
                $eventArgs->setNewValue('name', 'Bob');
            }
        }
    }
}
```

También puedes utilizar este escucha para implementar la validación de todos los campos que han cambiado. Esto es más eficaz que utilizar el ciclo de vida de una retrollamada, cuando hay costosas validaciones llama a:

```
<?php
class ValidCreditCardListener
{
    public function preUpdate(PreUpdateEventArgs $eventArgs)
    {
        if ($eventArgs->getEntity() instanceof Account) {
            if ($eventArgs->hasChangedField('creditCard')) {
                $this->validateCreditCard($eventArgs->getNewValue('creditCard'));
            }
        }
    }

    private function validateCreditCard($no)
    {
        // lanza una excepción para interrumpir el evento de vaciado. Debes revertir la transacción.
    }
}
```

Restricciones para este evento:

- Los cambios en las asociaciones de las entidades pasado no son reconocidos por la operación de vaciado más.
- Los cambios en los campos de las entidades pasadas no son reconocidos más por la operación de vaciado, utiliza el conjunto de cambios calculado pasado al evento para modificar los valores primitivos de los campos.
- Todas las llamadas a `EntityManager#persist()` o a `EntityManager#remove()`, incluso en combinación con la *API* de `UnitOfWork` se desalientan absolutamente y no funcionan como se espera fuera de la operación de vaciado.

postUpdate, postRemove, postPersist

Los tres eventos posteriores son llamados dentro del `EntityManager#remove()`. Changes in here are not relevant to the persistence in the database, but you can use these events to alter non-persistable items, like non-mapped fields, logging or even associated classes that are directly mapped by Doctrine.

postLoad

Este evento es llamado después de que el `EntityManager` construye una entidad.

1.11.6 Los eventos cargan la `ClassMetadata`

Cuando se lee la información de asignación para una entidad, esta rellena una instancia de `ClassMetadataInfo`. Te puedes conectar a este proceso y manipular la instancia.

```
<?php
$test = new EventTest();
$metadataFactory = $em->getMetadataFactory();
$evm = $em->getEventManager();
$evm->addEventListener(Events::loadClassMetadata, $test);

class EventTest
{
    public function loadClassMetadata(\Doctrine\ORM\Event\LoadClassMetadataEventArgs $eventArgs)
    {
        $classMetadata = $eventArgs->getClassMetadata();
        $fieldMapping = array(
            'fieldName' => 'about',
            'type' => 'string',
            'length' => 255
        );
        $classMetadata->mapField($fieldMapping);
    }
}
```

1.12 Procesamiento masivo

Este capítulo muestra cómo lograr eficientemente mayores inserciones, actualizaciones y remociones con *Doctrine*. El principal problema con las operaciones masivas, por lo general, es no quedarse sin memoria y esto, especialmente, es para lo cual las estrategias presentadas aquí proporcionan ayuda.

Advertencia: Una herramienta ORM, ante todo, no es muy adecuada para inserciones, actualizaciones o supresiones masivas. Cada RDBMS tiene su propia forma, más eficaz de tratar con este tipo de operaciones y si las opciones descritas a continuación no son suficientes para tus propósitos te recomendamos que utilices las herramientas de tu RDBMS particular para estas operaciones masivas.

1.12.1 Inserciones masivas

Las inserciones masivas en *Doctrine* se realizan mejor en grupos, aprovechando la transacción de escritura detrás del comportamiento de un `EntityManager`. El siguiente código muestra un ejemplo para insertar 10,000 objetos con una dimensión de 20 en el lote. Posiblemente tengas que experimentar con la dimensión del lote para encontrar el tamaño que mejor te funcione. Un mayor tamaño de los lotes significa reutilizar internamente declaraciones más elaboradas, pero también significa más trabajo durante el vaciado.

```
<?php
$batchSize = 20;
for ($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(); // ¡se desprende de todos los objetos de *Doctrine*!
    }
}
```

1.12.2 Actualizaciones masivas

Hay 2 posibilidades para actualizaciones masivas con *Doctrine*.

Actualización DQL

La manera en gran medida más eficaz para actualizaciones masivas es utilizar una consulta `DQL UPDATE`. Ejemplo:

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

Iterando en los resultados

Una solución alternativa para las actualizaciones masivas es utilizar la utilidad `Query#iterate()` para iterar paso a paso los resultados de la consulta en lugar de cargar todo el resultado en memoria a la vez. El siguiente ejemplo muestra cómo hacerlo, combinando la iteración con la estrategia del procesamiento masivo que ya utilizamos para las inserciones masivas:

```
<?php
$batchSize = 20;
$i = 0;
$q = $em->createQuery('select u from MiProyecto\Model\User u');
$iterableResult = $q->iterate();
foreach($iterableResult AS $row) {
```

```
    $user = $row[0];
    $user->increaseCredit();
    $user->calculateNewBonuses();
    if (($i % $batchSize) == 0) {
        $em->flush(); // Ejecuta todas las actualizaciones.
        $em->clear(); // ¡desvincula todos los objetos de *Doctrine*!
    }
    ++$i;
}
```

Nota: No es posible iterar en los resultados con las consultas que recuperan uniones a una colección de valores de la asociación. La naturaleza de tales conjuntos de resultados `SQL` no es adecuada para la hidratación incremental.

1.12.3 Eliminaciones masivas

Hay dos posibilidades para remociones masivas con *Doctrine*. Puedes emitir una consulta `DQL DELETE` o puedes iterar sobre los resultados de la eliminación de uno en uno.

DQL DELETE

La manera en gran medida más eficiente para la eliminación masiva es utilizar una consulta `DQL DELETE`.

Ejemplo:

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

Iterando en los resultados

Una solución alternativa para la eliminación masiva es usar la utilidad `Query#iterate()` para iterar paso a paso en los resultados de la consulta en lugar de cargar en memoria todo el resultado a la vez. El siguiente ejemplo muestra cómo hacerlo:

```
<?php
$batchSize = 20;
$i = 0;
$q = $em->createQuery('select u from MiProyecto\Model\User u');
$iterableResult = $q->iterate();
while (($row = $iterableResult->next()) !== false) {
    $em->remove($row[0]);
    if (($i % $batchSize) == 0) {
        $em->flush(); // Ejecuta todas las eliminaciones.
        $em->clear(); // ¡desvincula todos los objetos de *Doctrine*!
    }
    ++$i;
}
```

Nota: No es posible iterar en los resultados con las consultas que recuperan uniones a una colección de valores de la asociación. La naturaleza de tales conjuntos de resultados `SQL` no es adecuada para la hidratación incremental.

1.12.4 Iterando grandes resultados para el procesamiento de datos

Puedes utilizar el método `iterate()` sólo para iterar sobre un resultado grande y no intentar `UPDATE` o `DELETE`. La instancia de `IterableResult` devuelta por `$query->iterate()` implementa la interfaz Iterador para que puedas procesar un resultado de gran tamaño sin problemas de memoria utilizando el siguiente método:

```
<?php
$q = $this->_em->createQuery('select u from MiProyecto\Model\User u');
$iterableResult = $q->iterate();
foreach ($iterableResult AS $row) {
    // hace algo con los datos en la fila, $row[0] siempre es el objeto

    // se desprende de Doctrine, para que de inmediato se pueda recoger con la basura
    $this->_em->detach($row[0]);
}
```

Nota: No es posible iterar en los resultados con las consultas que recuperan uniones a una colección de valores de la asociación. La naturaleza de tales conjuntos de resultados `SQL` no es adecuada para la hidratación incremental.

1.13 Lenguaje de consulta *Doctrine*

DQL es el lenguaje de consulta de *Doctrine* y es un objeto derivado del lenguaje de consulta que es muy similar al lenguaje de consulta Hibernate (HQL (Hibernate Query Language)) o al lenguaje de consulta persistente de Java (JPQL (Java Persistence Query Language)).

En esencia, DQL proporciona potentes capacidades de consulta sobre los objetos del modelo. Imagina que todos los objetos yacen por ahí en algún almacenamiento (como una base de datos de objetos). Al escribir consultas DQL, piensas en la consulta que recupera un determinado subconjunto de objetos del almacenamiento.

Nota: Un error muy común para los principiantes es olvidar que DQL sólo es una forma de `SQL` y por lo tanto trata de usar nombres de tablas y columnas o uniones arbitrarias de tablas en una consulta. Es necesario pensar en DQL como un lenguaje de consulta para tu modelo de objetos, no para tu esquema relacional.

`DQL` es insensible a mayúsculas y minúsculas, salvo en nombres de clase, campos y espacios de nombres, en los cuales sí distingue entre mayúsculas y minúsculas.

1.13.1 Tipos de consultas `DQL`

`DQL` como un lenguaje de consulta tiene construcciones `SELECT`, `UPDATE` y `DELETE` que asignan a sus correspondientes tipos de declaraciones `SQL`. Las instrucciones `INSERT` no se permiten en `DQL`, porque las entidades y sus relaciones se tienen que introducir en el contexto de la persistencia a través de `EntityManager#persist()` para garantizar la coherencia de los objetos de tu modelo.

Las instrucciones `SELECT` de `DQL` son una forma muy poderosa de recuperar partes de tu modelo de dominio que no son accesibles a través de asociaciones. Además te permiten recuperar entidades y sus asociaciones en una sola consulta `SQL` en la cual puedes hacer una gran diferencia en el rendimiento en contraste al uso de varias consultas.

Las instrucciones `UPDATE` y `DELETE` de `DQL` ofrecen una manera de ejecutar cambios masivos en las entidades de tu modelo de dominio. Esto, a menudo es necesario cuando no puedes cargar en memoria todas las entidades afectadas por una actualización masiva.

1.13.2 Consultas SELECT

Cláusula SELECT DQL

La cláusula SELECT de una consulta DQL especifica lo que aparece en el resultado de la consulta. La composición de todas las expresiones en la cláusula select también influye en la naturaleza de los resultados de la consulta.

He aquí un ejemplo que selecciona todos los usuarios con una edad mayor de 20 años:

```
<?php
$query = $em->createQuery('SELECT u FROM MiProyecto\Model\User u WHERE u.age > 20');
$users = $query->getResult();
```

Examinemos la consulta:

- u es una - así llamada - variable de identificación o alias que se refiere a la clase MiProyecto\Model\User. Al colocar este alias en la cláusula SELECT específicamente queremos que todas las instancias de la clase Usuario que coinciden con esta consulta aparezcan en el resultado de la consulta.
- La palabra clave FROM siempre va seguida de un nombre de clase completo, el cual a su vez, es seguido por una variable de identificación o alias para ese nombre de clase. Esta clase designa la raíz de nuestra consulta desde la que podemos navegar a través de nuevas combinaciones (explicadas más adelante) y expresiones de ruta.
- La expresión u.age en la cláusula WHERE es una expresión de ruta. Las expresiones de ruta en DQL son fáciles de identificar por el uso del operador '.' que se utiliza para la construcción de rutas. La expresión de ruta u.age se refiere al campo edad en la clase Usuario.

El resultado de esta consulta sería una lista de objetos Usuario dónde todos los usuarios son mayores de 20 años.

La cláusula SELECT permite especificar las variables de identificación de clase que señalan la hidratación de una clase Entidad completa o sólo los campos de la Entidad con la sintaxis u.name. También se permite la combinación de ambas y es posible envolver los campos y valores de identificación en Funciones agregadas y DQL. Los campos numéricos pueden ser parte de cálculos utilizando las operaciones matemáticas. Ve la subsección sobre *funciones ''DQL'*, agregados y operaciones <#dqlfn>'_ para más información.

Uniones

Una consulta SELECT puede contener uniones. Hay dos tipos de uniones: Uniones "regulares" y uniones de "recuperación".

Uniones regulares: Se utilizan para limitar los resultados y/o calcular valores agregados.

Uniones de recuperación: Además de utilizarla como unión regular: Se utiliza para buscar entidades relacionadas y las incluye en el resultado hidratado de una consulta.

No hay ninguna palabra clave DQL especial que distinga una unión regular de una unión de recuperación. Una unión (ya sea una combinación interna o externa) se convierte en un "unión de recuperación" tan pronto como los campos de la entidad unida aparecen en la parte SELECT de la consulta DQL fuera de una función agregada. Por lo demás es una "unión regular".

Ejemplo:

Unión regular por dirección:

```
<?php
$query = $em->createQuery("SELECT u FROM User u JOIN u.address a WHERE a.city = 'Berlin'");
$users = $query->getResult();
```

Unión de recuperación por la dirección:

```
<?php
$query = $em->createQuery("SELECT u, a FROM User u JOIN u.address a WHERE a.city = 'Berlin'");
$users = $query->getResult();
```

Cuando *Doctrine* hidrata una consulta con una unión de recuperación devuelve la clase en la cláusula `FROM` en el nivel raíz de la matriz resultante. El ejemplo anterior devuelve una matriz de instancias de Usuario y la dirección de cada usuario se recupera e hidrata en la variable `User#address`. Si accedes a la dirección, *Doctrine* no necesita cargar la asociación con otra consulta diferida.

Nota: *Doctrine* te permite recorrer todas las asociaciones entre todos los objetos en el modelo del dominio. Los objetos que no se han cargado ya desde la base de datos son reemplazados con instancias delegadas cargadas de manera diferida. Las colecciones no cargadas también se sustituyen por instancias cargadas de manera diferida que recuperan todos los objetos contenidos en el primer acceso. Sin embargo, confiar en el mecanismo de carga diferida conlleva la ejecución de muchas pequeñas consultas a la base de datos, las cuales pueden afectar significativamente el rendimiento de tu aplicación. Las **uniones de recuperación** son la solución para hidratar la mayoría o la totalidad de las entidades que necesitas en una consulta `SELECT`.

Parámetros posicionales y nombrados

`DQL` es compatible con ambos parámetros nombrados y posicionales, sin embargo, en contraste con muchos dialectos `SQL`, los parámetros posicionales se especifican con números, por ejemplo "?1", "?2" y así sucesivamente. Los parámetros nombrados se especifican con ":nombre1", ":nombre2" y así sucesivamente.

Al hacer referencia a los parámetros en `Query#setParameter($param, $valor)`, ambos parámetros nombrados y posicionales se utilizan **sin** sus prefijos.

Ejemplos `SELECT` `DQL`

Esta sección contiene una gran cantidad de consultas `DQL` y algunas explicaciones de lo que está sucediendo. El resultado real también depende del modo de hidratación.

Hidrata todas las entidades Usuario:

```
<?php
$query = $em->createQuery('SELECT u FROM MiProyecto\Model\User u');
$users = $query->getResult(); // matriz de objetos User
```

Recupera los identificadores de todos los CmsUsers:

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u');
$ids = $query->getResult(); // matriz de identificadores de CmsUser
```

Recupera los identificadores de todos los usuarios que han escrito un artículo:

```
<?php
$query = $em->createQuery('SELECT DISTINCT u.id FROM CmsArticle a JOIN a.user u');
$ids = $query->getResult(); // matriz de identificadores de CmsUser
```

Recupera todos los artículos y los ordena por el nombre del artículo de la instancia Usuarios:

```
<?php
$query = $em->createQuery('SELECT a FROM CmsArticle a JOIN a.user u ORDER BY u.name ASC');
$articles = $query->getResult(); // array of CmsArticle objects
```

Recupera el Nombre de usuario y Nombre de un `CmsUser`:

```
<?php
$query = $em->createQuery('SELECT u.username, u.name FROM CmsUser u');
$users = $query->getResults(); // matriz de valores username y name desde CmsUser
echo $users[0]['username'];
```

Recupera un `ForumUser` y su entidad asociada:

```
<?php
$query = $em->createQuery('SELECT u, a FROM ForumUser u JOIN u.avatar a');
$users = $query->getResult(); // matriz de objetos ForumUser con su avatar asociado cargado
echo get_class($users[0]->getAvatar());
```

Recupera un CmsUser y hace una unión para recuperar todos los números telefónicos que tiene:

```
<?php
$query = $em->createQuery('SELECT u, p FROM CmsUser u JOIN u.phonenumbers p');
$users = $query->getResult(); // matriz de objetos CmsUser con los números telefónicos asociados carg
$phonenumbers = $users[0]->getPhonenumbers();
```

Hidrata un resultado organizado en ordena ascendente:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id ASC');
$users = $query->getResult(); // matriz de objetos ForumUser
```

O en orden descendente:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id DESC');
$users = $query->getResult(); // matriz de objetos ForumUser
```

Usando funciones agregadas:

```
<?php
$query = $em->createQuery('SELECT COUNT(u.id) FROM Entities\User u');
$count = $query->getSingleScalarResult();

$query = $em->createQuery('SELECT u, count(g.id) FROM Entities\User u JOIN u.groups g GROUP BY u.id')
$result = $query->getResult();
```

Con cláusula `WHERE` y parámetros posicionales:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$users = $query->getResult(); // matriz de objetos ForumUser
```

Con cláusula `WHERE` y parámetros nombrados:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.username = :name');
$query->setParameter('name', 'Bob');
$users = $query->getResult(); // matriz de objetos ForumUser
```

Con condiciones anidadas en la cláusula `WHERE`:

```
<?php
$query = $em->createQuery('SELECT u from ForumUser u WHERE (u.username = :name OR u.username = :name2
$query->setParameters(array(
```

```
    'name' => 'Bob',
    'name2' => 'Alice',
    'id' => 321,
));
$users = $query->getResult(); // matriz de objetos ForumUser
```

Con `COUNT DISTINCT`:

```
<?php
$query = $em->createQuery('SELECT COUNT(DISTINCT u.name) FROM CmsUser');
$users = $query->getResult(); // matriz de objetos ForumUser
```

Con expresiones aritméticas en la cláusula `WHERE`:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE ((u.id + 5000) * u.id + 3) < 10000000');
$users = $query->getResult(); // matriz de objetos ForumUser
```

Usando una `LEFT JOIN` para hidratar todos los identificadores de usuario y, opcionalmente, los identificadores de artículos asociados:

```
<?php
$query = $em->createQuery('SELECT u.id, a.id as article_id FROM CmsUser u LEFT JOIN u.articles a');
$results = $query->getResult(); // matriz de identificadores y cada article_id de cada usuario
```

Restringiendo una cláusula `JOIN` con condiciones adicionales:

```
<?php
$query = $em->createQuery("SELECT u FROM CmsUser u LEFT JOIN u.articles a WITH a.topic LIKE '%foo%'")
$users = $query->getResult();
```

Usando varias `JOIN`:

```
<?php
$query = $em->createQuery('SELECT u, a, p, c FROM CmsUser u JOIN u.articles a JOIN u.phonenumbers p
$users = $query->getResult();
```

`BETWEEN` en una cláusula `WHERE`:

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id BETWEEN ?1 AND ?2');
$query->setParameter(1, 123);
$query->setParameter(2, 321);
$usernames = $query->getResult();
```

Funciones `DQL` en cláusulas `WHERE`:

```
<?php
$query = $em->createQuery("SELECT u.name FROM CmsUser u WHERE TRIM(u.name) = 'someone'");
$usernames = $query->getResult();
```

La expresión IN():

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id IN(46)');
$usernames = $query->getResult();

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id IN (1, 2)');
$users = $query->getResult();
```

```
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id NOT IN (1)');
$users = $query->getResult();
```

Función `CONCAT() DQL`:

```
<?php
$query = $em->createQuery("SELECT u.id FROM CmsUser u WHERE CONCAT(u.name, 's') = ?1");
$query->setParameter(1, 'Jess');
$ids = $query->getResult();

$query = $em->createQuery('SELECT CONCAT(u.id, u.name) FROM CmsUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$idUsernames = $query->getResult();
```

`EXISTS` en una cláusula `WHERE` con una subconsulta correlacionada

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE EXISTS (SELECT p.phonenumber FROM CmsPhor
$ids = $query->getResult();
```

Recupera todos los usuarios que son miembros de `$grupo`.

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE :groupId MEMBER OF u.groups');
$query->setParameter('groupId', $grupo);
$ids = $query->getResult();
```

Recupera todos los usuarios que tienen más de un número telefónico

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE SIZE(u.phonenumbers) > 1');
$users = $query->getResult();
```

Recupera todos los usuarios que no tienen número telefónico

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.phonenumbers IS EMPTY');
$users = $query->getResult();
```

Recupera todas las instancias de un tipo específico, para usar con jerarquías de herencia:

```
<?php
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u INSTAN
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u INSTAN
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u NOT IN
```

Sintaxis parcial de objeto

Por omisión, cuando ejecutas una consulta `DQL` en *Doctrine* y seleccionas sólo un subconjunto de los campos de una determinada entidad, no recibes objetos. En su lugar, sólo recibirás matrices como un conjunto de resultados, similar a cómo lo harías si estuvieras utilizando `SQL` directamente y uniendo algunos datos.

Si deseas seleccionar objetos parciales, puedes utilizar la palabra clave `partial` de `DQL`:

```
<?php
$query = $em->createQuery('SELECT partial u.{id, username} FROM CmsUser u');
$users = $query->getResult(); // array of partially loaded CmsUser objects
```

También utilizas la sintaxis parcial cuando haces uniones:

```
<?php
$query = $em->createQuery('SELECT partial u.{id, username}, partial a.{id, name} FROM CmsUser u JOIN
$users = $query->getResult(); // matriz de objetos CmsUser cargados parcialmente
```

Usando INDEX BY

La construcción INDEX BY no hace otra cosa que traducirla directamente a SQL, pero que afecta a objetos y a la hidratación de la matriz. Después de cada cláusula JOIN y FROM debes especificar el campo por el cual se debe indexar esta clase en los resultados. Por omisión, un resultado se incrementa en claves numéricas comenzando en 0. Sin embargo, con INDEX BY puedes especificar cualquier otra columna para que sea la clave de tu resultado, sin embargo, esto sólo tiene sentido con campos primarios o únicos:

```
SELECT u.id, u.status, upper(u.name) nameUpper FROM User u INDEX BY u.id
JOIN u.phonenumbers p INDEX BY p.phonenumber
```

Devuelve una matriz indexada tanto en user-id como en phonenumber-id:

```
array
  0 =>
    array
      1 =>
        object(stdClass)[299]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser' (length=33)
          public 'id' => int 1
          ..
      'nameUpper' => string 'ROMANB' (length=6)
  1 =>
    array
      2 =>
        object(stdClass)[298]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser' (length=33)
          public 'id' => int 2
          ...
      'nameUpper' => string 'JWAGE' (length=5)
```

1.13.3 Consultas UPDATE

DQL no sólo te permite seleccionar tus entidades usando nombres de campo, también puedes ejecutar actualizaciones masivas en un conjunto de entidades mediante una consulta UPDATE. La sintaxis de una consulta de actualización funciona como esperabas, como muestra el siguiente ejemplo:

```
UPDATE MiProyecto\Model\User u SET u.password = 'new' WHERE u.id IN (1, 2, 3)
```

Las referencias a entidades relacionadas sólo son posibles en la cláusula WHERE y usando subselecciones.

Advertencia: Las instrucciones UPDATE de DQL se transfieren directamente a una instrucción UPDATE de la bases de datos y por lo tanto evitan cualquier esquema de bloqueo, eventos y no incrementan la columna de versión. Las entidades que ya están cargadas en el contexto de persistencia *NO* se sincronizan con el estado actualizado de la base de datos. Es altamente recomendable llamar a EntityManager#clear() y recuperar nuevas instancias de cualquier entidad afectada.

1.13.4 Consultas DELETE

También puedes especificar consultas DELETE con DQL y su sintaxis es tan simple como la sintaxis de UPDATE:

```
DELETE MiProyecto\Model\User u WHERE u.id = 4
```

Aplican las mismas restricciones para la referencia de entidades relacionadas.

Advertencia: DELETE de DQL se transfiere directamente a una DELETE de la base de datos y por lo tanto, evitan eventos y comprobaciones de la columna de versión si no se agregan explícitamente a la cláusula WHERE de la consulta. Además elimina entidades especificas *NO* propagadas en cascada a entidades relacionadas, incluso si se ha especificado en los metadatos.

1.13.5 Funciones, operadores, agregados

Funciones DQL

Las siguientes funciones son compatibles con SELECT, WHERE y HAVING:

- ABS(arithmetic_expression)
- CONCAT(cadena1, cadena2)
- CURRENT_DATE() - Devuelve la fecha actual
- CURRENT_TIME() - Devuelve la hora actual
- CURRENT_TIMESTAMP() - Devuelve un timestamp de la fecha y hora actual.
- LENGTH(cadena) - Devuelve la longitud de la cadena dada
- LOCATE(aguja, pajar [, desplazamiento]) - Localiza la primer aparición de la subcadena (aguja) en la cadena (pajar).
- LOWER(cadena) - Devuelve la cadena en minúsculas.
- MOD(a, b) - Devuelve a RESIDUO b.
- SIZE(coleccion) - Devuelve el número de elementos en la colección especificada
- SQRT(q) - Devuelve la raíz cuadrada de q.
- SUBSTRING(cadena, inicio [, longitud]) - Devuelve una subcadena de la cadena dada.
- TRIM([LEADING \| TRAILING \| BOTH] ['carácter' FROM] cadena) - Recorta la cadena por el carácter de recorte, por omisión es el espacio en blanco.
- UPPER(cadena) - Devuelve en mayúsculas la cadena dada.
- DATE_ADD(date, days) - Add the number of days to a given date.
- DATE_SUB(date, days) - Substract the number of days from a given date.
- DATE_DIFF(date1, date2) - Calculate the difference in days between date1-date2.

Operadores aritméticos

Puedes hacer matemáticas en DQL utilizando valores numéricos, por ejemplo:

```
SELECT person.salary * 1.5 FROM CompanyPerson person WHERE person.salary < 100000
```

Funciones agregadas

Los siguientes funciones agregadas se permiten en las cláusulas `SELECT`} y ``GROUP BY`: AVG, COUNT, MIN, MAX, SUM

Otras expresiones

DQL ofrece una amplia gama de expresiones adicionales que conocen desde SQL, aquí hay una lista de todas las construcciones compatibles:

- `ALL/ANY/SOME` - Usadas en una cláusula `WHERE` seguida por una subselección, esto funciona como las construcciones SQL equivalentes.
- `BETWEEN a AND b` y `NOT BETWEEN a AND b` - las puedes utilizar para coincidir con rangos de valores aritméticos.
- `IN (x1, x2, ...)` y `NOT IN (x1, x2, ..)` - las puedes utilizar para coincidir con un conjunto de valores.
- `LIKE ..` y `NOT LIKE ..` - emparejan partes de una cadena o texto usando **%** como comodín.
- `IS NULL` e `IS NOT NULL` - para comprobar si hay valores nulos
- `EXISTS` y `NOT EXISTS` en combinación con una subselección

Añadiendo tus propias funciones al lenguaje DQL

Por omisión DQL viene con funciones que son parte fundamental de las grandes bases de datos subyacentes. Sin embargo, debes eligir una plataforma de base de datos al inicio de tu proyecto y lo más probable es que nunca la cambies. Para estos casos, fácilmente puedes extender el analizador DQL con tus propias funciones especializadas en la plataforma.

Puedes registrar las funciones DQL personalizadas en la configuración de tu ORM:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($nombre, $class);
$config->addCustomNumericFunction($nombre, $class);
$config->addCustomDatetimeFunction($nombre, $class);

$em = EntityManager::create($dbParams, $config);
```

Las funciones que devuelven o bien un valor de cadena, numérico o de fecha y hora dependiendo del tipo de función registrado. Como ejemplo vamos a añadir la funcionalidad `FLOOR()` específica de MySQL. Todas las clases dadas tienen que implementar la clase base:

```
<?php
namespace MiProyecto\Query\AST;

use \Doctrine\ORM\Query\AST\Functions\FunctionsNode;

class MysqlFloor extends FunctionNode
{
    public $simpleArithmeticExpression;

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'FLOOR(' . $sqlWalker->walkSimpleArithmeticExpression(
```

```
            $this->simpleArithmeticExpression
        ) . ')';
    }

    public function parse(\Doctrine\ORM\Query\Parser $analizador)
    {
        $lexer = $analizador->getLexer();

        $analizador->match(Lexer::T_ABS);
        $analizador->match(Lexer::T_OPEN_PARENTHESIS);

        $this->simpleArithmeticExpression = $analizador->SimpleArithmeticExpression();

        $analizador->match(Lexer::T_CLOSE_PARENTHESIS);
    }
}
```

Debemos registrar la función por llamar y luego la podemos utilizar:

```
<?php
\Doctrine\ORM\Query\Parser::registerNumericFunction('FLOOR', 'MiProyecto\Query\MysqlFloor');
$dql = "SELECT FLOOR(person.salary * 1.75) FROM CompanyPerson person";
```

1.13.6 Consultando las clases heredadas

En esta sección se muestra cómo puedes consultar las clases heredadas y qué tipo de resultados puede esperar.

Tabla única

La herencia de tabla única es una estrategia de asignación de herencias donde todas las clases de la jerarquía se asignan a una única tabla de la base de datos. A fin de distinguir qué fila representa cual tipo en la jerarquía, se utiliza una así llamada "columna discriminadora".

En primer lugar tenemos que estructurar un conjunto de entidades para usarlo en el ejemplo. En este escenario, son una Persona y Empleado genéricos de ejemplo:

```
<?php
namespace Entities;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    /**
     * @Id @Column(type="integer")
     * @GeneratedValue
     */
    protected $id;

    /**
     * @Column(type="string", length=50)
     */
```

```
    protected $nombre;

    // ...
}

/**
 * @Entity
 */
class Employee extends Person
{
    /**
     * @Column(type="string", length=50)
     */
    private $department;

    // ...
}
```

Primero fíjate que el SQL generado para crear las tablas de estas entidades tiene el siguiente aspecto:

```
CREATE TABLE Person (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    discr VARCHAR(255) NOT NULL,
    department VARCHAR(50) NOT NULL
)
```

Ahora, cuando persistas una nueva instancia de Empleado esta fijará el valor discriminador por nosotros automáticamente:

```
<?php
$employee = new \Entities\Employee();
$employee->setName('test');
$employee->setDepartment('testing');
$em->persist($employee);
$em->flush();
```

Ahora vamos a ejecutar una consulta sencilla para recuperar el Empleado que acabamos de crear:

```
SELECT e FROM Entities\Employee e WHERE e.name = 'test'
```

Si revisamos el SQL generado te darás cuenta de que tiene añadidas ciertas condiciones especiales para garantizar que sólo devuelve entidades Empleado:

```
SELECT p0_.id AS id0, p0_.name AS name1, p0_.department AS department2,
       p0_.discr AS discr3 FROM Person p0_
WHERE (p0_.name = ?) AND p0_.discr IN ('employee')
```

Herencia clase→tabla

La herencia clase→tabla es una estrategia de asignación de herencias, donde se asigna cada clase en una jerarquía de varias tablas: su propia tabla y las tablas de todas las clases padre. La tabla de una clase hija está relacionada con la tabla de una clase padre a través de una clave externa. *Doctrine* 2 implementa esta estrategia a través del uso de una columna discriminadora de la tabla superior de la jerarquía, porque es la forma más fácil de lograr consultas polimórficas con herencia clase→tabla.

El ejemplo de la herencia clase→tabla es el mismo que el de la tabla única, sólo hay que cambiar el tipo de herencia de SINGLE_TABLE a JOINED:

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

Ahora echa un vistazo al código SQL generado para crear la tabla, verás que hay algunas diferencias:

```
CREATE TABLE Person (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    discr VARCHAR(255) NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Employee (
    id INT NOT NULL,
    department VARCHAR(50) NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Employee ADD FOREIGN KEY (id) REFERENCES Person(id) ON DELETE CASCADE
```

- Los datos se dividen entre dos tablas
- Existe una clave externa entre las dos tablas

Ahora bien, si insertásemos el mismo Empleado como lo hicimos en el ejemplo SINGLE_TABLE y ejecutamos la consulta del mismo ejemplo, generará SQL diferente uniendo automáticamente la información de Persona:

```
SELECT p0_.id AS id0, p0_.name AS name1, e1_.department AS department2,
       p0_.discr AS discr3
FROM Employee e1_ INNER JOIN Person p0_ ON e1_.id = p0_.id
WHERE p0_.name = ?
```

1.13.7 La clase Query

Una instancia de la clase Doctrine\ORM\Query representa una consulta DQL. Creas una instancia de Query llamando a EntityManager#CreateQuery($DQL), pasando la cadena de consulta DQL. Alternativamente, puedes crear una instancia vacía Query y después invocas a Query#setDql($DQL). He aquí algunos ejemplos:

```php
<?php
// $em es una instancia de EntityManager

// ejemplo1: pasando una cadena a DQL
$q = $em->createQuery('select u from MiProyecto\Model\User u');

// ejemplo2: usando setDql
$q = $em->createQuery();
$q->setDql('select u from MiProyecto\Model\User u');
```

Formatos del resultado de la consulta

El formato en el que se devuelve el resultado de una consulta DQL SELECT puede estar influenciado por el así llamado modo de hidratación. Un modo de hidratación especifica de manera particular en que se transforma un conjunto de resultados SQL. Cada modo tiene su propio método de hidratación dedicado en la clase Query. Aquí están:

- Query#getResult(): Recupera una colección de objetos. El resultado es una colección de objetos simple (pura) o una matriz, donde los objetos están anidados en las filas del resultado (mixtos).
- Query#getSingleResult(): Recupera un único objeto. Si el resultado contiene más de un objeto, lanza una excepción. La distinción puro/mixto no aplica.
- Query#getArrayResult(): Recupera una matriz gráfica (una matriz anidada), que es en gran medida intercambiable con los objetos gráficos generados por Query#getResult() de sólo lectura.

  Nota: Una matriz gráfica puede diferir del objeto gráfico correspondiente en ciertos escenarios, debido a la diferencia de la semántica de identidad entre matrices y objetos.

- Query#getScalarResult(): Recupera un resultado plano/rectangular del conjunto de valores escalares que puede contener datos duplicados. La distinción puro/mixto no aplica.
- Query#getSingleScalarResult(): Recupera un valor escalar único a partir del resultado devuelto por el DBMS. Si el resultado contiene más de un valor escalar único, lanza una excepción. La distinción puro/mixto no aplica.

En lugar de usar estos métodos, también puedes usar el método de propósito general Query#execute(array $params = array(), $hydrationMode = Query::HYDRATE_OBJECT). Usando este método puedes suministrar directamente - como segundo parámetro - el modo de hidratación a través de una de las constantes Query. De hecho, los métodos mencionados anteriormente son sólo convenientes atajos para el método execute. Por ejemplo, el método Query#getResult() internamente llama a execute, pasando un Query::HYDRATE_OBJECT como el modo de hidratación.

Es preferible usar los métodos mencionados anteriormente, ya que conducen a un código más conciso.

Resultados puros y mixtos

La naturaleza de un resultado devuelto por una consulta DQL SELECT recuperado a través de Query#getResult() o Query#getArrayResult() puede ser de dos formas: **puro** y **mixto**. En los sencillos ejemplos anteriores, ya habías experimentado un resultado de Query "puro", con sólo objetos. De manera predeterminada, el tipo de resultado es **puro** pero **tan pronto como aparezcan valores escalares, tal como valores agregados u otros valores escalares que no pertenecen a una entidad, en la parte ‘‘SELECT‘‘ de la consulta ‘‘DQL‘‘, el resultado se convierte en mixto**. Un resultado mixto tiene una estructura diferente que un resultado puro con el fin de adaptarse a los valores escalares.

A resultado puro generalmente tiene la siguiente apariencia:

```
$dql = "SELECT u FROM User u";

array
    [0] => Object
    [1] => Object
    [2] => Object
    ...
```

Por otro lado, un resultado mixto tiene la siguiente estructura general:

```
$dql = "SELECT u, 'some scalar string', count(u.groups) AS num FROM User u JOIN u.groups g GROUP BY u

array
    [0]
        [0] => Object
        [1] => "some scalar string"
        ['num'] => 42
        // ... más valores escalares, bien indexados numéricamente o con un nombre
    [1]
        [0] => Object
        [1] => "some scalar string"
        ['num'] => 42
        // ... más valores escalares, bien indexados numéricamente o con un nombre
```

Para comprender mejor los resultados mixtos, considera la siguiente consulta `DQL`:

```
SELECT u, UPPER(u.name) nameUpper FROM MiProyecto\Model\User u
```

Esta consulta utiliza la función `DQL UPPER` que devuelve un valor escalar y debido a que ahora hay un valor escalar en la cláusula `SELECT`, obtienes un resultado mixto.

Las siguientes convenciones son para resultados mixtos:

- El objeto a recuperar en la cláusula `FROM` siempre se coloca con la clave '0'.
- Cada escalar sin nombre es numerado en el orden dado en la consulta, comenzando en 1.
- A cada alias escalar le es dado su nombre alias como clave. Manteniendo las mayúsculas y minúsculas del nombre.
- Si se recuperan varios objetos de la cláusula `FROM` estos se alternan cada fila.

Así es como se podría ver el resultado:

```
array
    array
        [0] => User (Object)
        ['nameUpper'] => "ROMAN"
    array
        [0] => User (Object)
        ['nameUpper'] => "JONATHAN"
    ...
```

Y aquí está cómo lo debes acceder en el código *PHP*:

```
<?php
foreach ($results as $row) {
    echo "Name: " . $row[0]->getName();
    echo "Name UPPER: " . $row['nameUpper'];
}
```

Recuperando múltiples entidades `FROM`

Si recuperas varias entidades enumeradas en la cláusula `FROM` entonces la hidratación devuelve las filas iterando las diferentes entidades del nivel superior.

```
$dql = "SELECT u, g FROM User u, Group g";

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

```
    [1] => Object (Group)
    [2] => Object (User)
    [3] => Object (Group)
```

Modos de hidratación

Cada uno de los modos de hidratación hace suposiciones sobre cómo se devuelve el resultado al espacio del usuario. Debes conocer todos los detalles para hacer el mejor uso de los diferentes formatos del resultado:

Las constantes para los diferentes modos de hidratación son las siguientes:

- Query::HYDRATE_OBJECT
- Query::HYDRATE_ARRAY
- Query::HYDRATE_SCALAR
- Query::HYDRATE_SINGLE_SCALAR

Objeto `Hydration`

El objeto `Hydration` hidrata el conjunto de resultados en el objeto gráfico:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_OBJECT);
```

Hidratación de matriz

Puedes ejecutar la misma consulta con la hidratación de matriz y el conjunto de resultados se hidrata en una matriz que representa al objeto gráfico:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_ARRAY);
```

Puedes utilizar el acceso directo `getArrayResult()`, así:

```
<?php
$users = $query->getArrayResult();
```

Hidratación escalar

Si quieres devolver un resultado conjunto plano en lugar de un objeto gráfico puedes usar la hidratación escalar:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_SCALAR);
echo $users[0]['u_id'];
```

Tomando las siguientes asunciones sobre los campos seleccionados con hidratación escalar:

1. Los campos de las clases están prefijados por el alias `DQL` en el resultado. Una consulta del tipo `SELECT u.name ..` devuelve una clave `u\_name` en las filas del resultado.

hidratación escalar única

Si una consulta sólo devuelve un valor escalar único puedes utilizar la hidratación escalar única:

```
<?php
$query = $em->createQuery('SELECT COUNT(a.id) FROM CmsUser u LEFT JOIN u.articles a WHERE u.username
$query->setParameter(1, 'jwage');
$numArticles = $query->getResult(Query::HYDRATE_SINGLE_SCALAR);
```

Puedes utilizar el acceso directo `getSingleScalarResult()`, así:

```
<?php
$numArticles = $query->getSingleScalarResult();
```

Modos de hidratación personalizados

Puedes agregar fácilmente tu propio modo de hidratación personalizado creando primero una clase que extienda a `AbstractHydrator`:

```
<?php
namespace MiProyecto\Hydrators;

use Doctrine\ORM\Internal\Hydration\AbstractHydrator;

class CustomHydrator extends AbstractHydrator
{
    protected function _hydrateAll()
    {
        return $this->_stmt->fetchAll(PDO::FETCH_ASSOC);
    }
}
```

A continuación sólo hay que añadir la clase a la configuración del ORM:

```
<?php
$em->getConfiguration()->addCustomHydrationMode('CustomHydrator', 'MiProyecto\Hydrators\CustomHydrato
```

Ahora, la hidratante está lista para utilizarla en tus consultas:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$results = $query->getResult('CustomHydrator');
```

Iterando en grandes conjuntos de resultados

Hay situaciones en las que una consulta que deseas ejecutar devuelve un gran conjunto de resultados que necesitas procesar. Todos los modos de hidratación descritos anteriormente cargan completamente un conjunto de resultados en la memoria, lo cual podría no ser factible con grandes conjuntos de resultados. Ve la sección Procesamiento masivo para los detalles de cómo recorrer grandes conjuntos de resultados.

Funciones

Existen los siguientes métodos en el `AbstractQuery` que extienden ambas `Query` y *NativeQuery*‘.

Parámetros

Las instrucciones preparadas que usen comodines numérico o denominados, requieren parámetros adicionales para que sean ejecutable contra la base de datos. Para pasar parámetros a la consulta puedes utilizar los siguientes métodos:

- `AbstractQuery::setParameter($param, $valor)` - Ajusta el comodín numérico o nombrado con el valor dado.
- `AbstractQuery::setParameters(array $params)` - Designa una matriz de parámetros de pares clave-valor.
- `AbstractQuery::getParameter($param)`
- `AbstractQuery::getParameters()`

Ambos nombre y parámetros posicionales son pasados a estos métodos sin su *?* o : prefijo.

API relacionada a la caché

Puedes almacenar en caché los resultados de consultas basadas ya sea en todas las variables que definen el resultado (`SQL`, modo de hidratación, parámetros y sugerencias) o claves de caché definidas por el usuario. Sin embargo, por omisión los resultados de la consulta no se almacenan en caché en absoluto. Tienes que habilitar la caché de resultados en base a cada consulta. El siguiente ejemplo muestra un completo flujo de trabajo utilizando la *API* de la caché de Resultados:

```
<?php
$query = $em->createQuery('SELECT u FROM MiProyecto\Model\User u WHERE u.id = ?1');
$query->setParameter(1, 12);

$query->setResultCacheDriver(new ApcCache());

$query->useResultCache(true)
      ->setResultCacheLifeTime($seconds = 3600);

$result = $query->getResult(); // cache miss

$query->expireResultCache(true);
$result = $query->getResult(); // forced expire, cache miss

$query->setResultCacheId('my_query_result');
$result = $query->getResult(); // saved in given result cache id.

// o llama a useResultCache() con todos los parámetros:
$query->useResultCache(true, $seconds = 3600, 'my_query_result');
$result = $query->getResult(); // cache hit!

**TIP** Puedes configurar globalmente el controlador de la caché de resultados en la instancia de ``C
```

Sugerencias de consulta

Puedes pasar pistas al analizador de consultas e hidratantes usando el método `AbstractQuery::setHint($nombre, $valor)`. Actualmente existen sugerencias de consulta sobre todo internas que no se pueden consumir en el entorno de usuario. Sin embargo, en el entorno de usuario vamos a utilizar las siguientes sugerencias:

- `Query::HINT\_FORCE\_PARTIAL\_LOAD` - Permite hidratar objetos, aunque no recupera todas sus columnas. Esta sugerencia de consulta se puede utilizar para manejar los problemas de consumo de memoria

con grandes conjuntos de resultados que contienen datos de caracteres o binarios. *Doctrine* no tiene ninguna manera implícita para volver a cargar estos datos. Los objetos cargados parcialmente se tienen que pasar a `EntityManager::refresh()` para poder volver a cargarlos por completo desde la base de datos.

- `Query::HINT\_REFRESH` - Esta consulta la utiliza `EntityManager::refresh()` internamente y también la puedes utilizar en modo usuario. Si especificas esta sugerencia y una consulta devuelve los datos de una entidad que ya esté gestionada por el `UnitOfWork`, los campos de la entidad existente se actualizarán. En operación normal, un conjunto de resultados que cargó datos de una entidad ya existente se descarta en favor de la entidad existente.
- `Query::HINT\_CUSTOM\_TREE\_WALKERS` - Una matriz adicional de instancias de `Doctrine\ORM\Query\TreeWalker` que están conectadas al proceso de análisis de la consulta `DQL`.

Caché de consultas (consultas `DQL` solamente)

Analiza una consulta `DQL` y la convierte en una consulta `SQL` en la plataforma de base de datos subyacente, obviamente, tiene cierta sobrecarga en contraste con la ejecución directa de las consultas `SQL` nativas. Es por eso que existe una caché dedicada a las consultas para almacenar en caché los resultados del analizador `DQL`. En combinación con el uso de comodines que pueden reducir el número de consultas analizadas en la producción a cero.

De manera predeterminada, el controlador de caché de consultas es pasado desde la instancia de `Doctrine\ORM\Configuration` a cada instancia de `Doctrine\ORM\Query` y está activado por omisión. Esto también significa que normalmente no tienes que maniobrar con los parámetros de la caché de consultas, no obstante, si lo hace hay varios métodos para interactuar con ellos:

- `Query::setQueryCacheDriver($driver)` - Permite establecer una instancia de Cache
- `Query::setQueryCacheLifeTime($seconds = 3600)` - Ajusta el tiempo de vida del almacenamiento en caché de las consultas.
- `Query::expireQueryCache($bool)` - Hace cumplir la expiración de la caché de consultas si se establece en true.
- `Query::getExpireQueryCache()`
- `Query::getQueryCacheDriver()`
- `Query::getQueryCacheLifeTime()`

Primer elemento y máximos resultados (consultas `DQL` solamente)

Puedes limitar el número de resultados devueltos por una consulta `DQL`, así como especificar el desplazamiento inicial, *Doctrine* utiliza una estrategia de manipulación de consultas de selección para devolver sólo el número de resultados solicitado:

- `Query::setMaxResults($maxResults)`
- `Query::setFirstResult($offset)`

Nota: Si tu consulta contiene una colección recuperada desde uniones, especificar los métodos para limitar el resultado no funciona como cabría esperar. El máximos conjunto de resultados restringe el número de filas recuperadas de bases de datos, sin embargo en el caso de colecciones recuperadas de uniones puede aparecer una entidad raíz en muchas filas, hidratando eficientemente menos que el número especificado de resultados.

Temporarily change fetch mode in DQL

While normally all your associations are marked as lazy or extra lazy you will have cases where you are using DQL and don't want to fetch join a second, third or fourth level of entities into your result, because of the increased cost of the SQL JOIN. You can mark a many-to-one or one-to-one association as fetched temporarily to batch fetch these entities using a WHERE .. IN query.

```
<?php
$query = $em->createQuery("SELECT u FROM MyProject\User u");
$query->setFetchMode("MyProject\User", "address", "EAGER");
$query->execute();
```

Given that there are 10 users and corresponding addresses in the database the executed queries will look something like:

```
SELECT * FROM users;
SELECT * FROM address WHERE id IN (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

1.13.8 EBNF

El siguiente contexto libre de gramática, escrito en una variante EBNF, describe el lenguaje de consulta de *Doctrine*. Puedes consultar esta gramática cuando no estés seguro de las posibilidades de DQL o cual es la sintaxis correcta para una consulta en particular.

Sintaxis de documento:

- No terminales comienzan con un carácter en mayúsculas
- Terminales comienzan con un carácter en minúscula
- Los paréntesis (...) se utilizan para agrupar
- Los corchetes [...] se utilizan para definir una parte opcional, por ejemplo, cero o una vez
- Las llaves {...} se utilizan para repetición, por ejemplo, cero o más veces
- Las comillas "..." definen una cadena terminal, una barra vertical | representa una alternativa

Terminales

- identificador (nombre, correoe, ...)
- string ('foo', 'bar"s casa', ' %ninja %', ...)
- char ('/', '\', ' ', ...)
- integer (-1, 0, 1, 34, ...)
- float (-0.23, 0.007, 1.245342E+8, ...)
- boolean (false, true)

Lenguaje de consulta

```
QueryLanguage ::= DeclaraciónDeSelección | DeclaraciónDeActualización | DeclaraciónDeEliminación
```

Declaraciones

```
SelectStatement ::= SelectClause FromClause [WhereClause] [GroupByClause] [HavingClause] [OrderByClau
UpdateStatement ::= UpdateClause [WhereClause]
DeleteStatement ::= DeleteClause [WhereClause]
```

Identificadores

```
/* Alias Identification usage (the "u" of "u.name") */
IdentificationVariable ::= identifier

/* Alias Identification declaration (the "u" of "FROM User u") */
AliasIdentificationVariable :: = identifier

/* identifier that must be a class name (the "User" of "FROM User u") */
AbstractSchemaName ::= identifier

/* identifier that must be a field (the "name" of "u.name") */
/* This is responsible to know if the field exists in Object, no matter if it's a relation or a simpl
FieldIdentificationVariable ::= identifier

/* identifier that must be a collection-valued association field (to-many) (the "Phonenumbers" of "u.
CollectionValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be a single-valued association field (to-one) (the "Group" of "u.Group") */
SingleValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be an embedded class state field (for the future) */
EmbeddedClassStateField ::= FieldIdentificationVariable

/* identifier that must be a simple state field (name, email, ...) (the "name" of "u.name") */
/* The difference between this and FieldIdentificationVariable is only semantical, because it points
SimpleStateField ::= FieldIdentificationVariable

/* Alias ResultVariable declaration (the "total" of "COUNT(*) AS total") */
AliasResultVariable = identifier

/* ResultVariable identifier usage of mapped field aliases (the "total" of "COUNT(*) AS total") */
ResultVariable = identifier
```

Expresiones de ruta

```
/* "u.Group" or "u.Phonenumbers" declarations */
JoinAssociationPathExpression             ::= IdentificationVariable "." (CollectionValuedAssociation

/* "u.Group" or "u.Phonenumbers" usages */
AssociationPathExpression                 ::= CollectionValuedPathExpression | SingleValuedAssociati

/* "u.name" or "u.Group" */
SingleValuedPathExpression                ::= StateFieldPathExpression | SingleValuedAssociationPath

/* "u.name" or "u.Group.name" */
StateFieldPathExpression                  ::= IdentificationVariable "." StateField | SingleValuedAss

/* "u.Group" */
```

```
SingleValuedAssociationPathExpression     ::= IdentificationVariable "." SingleValuedAssociationFiel

/* "u.Group.Permissions" */
CollectionValuedPathExpression            ::= IdentificationVariable "." {SingleValuedAssociationFie

/* "name" */
StateField                                ::= {EmbeddedClassStateField "."}* SimpleStateField

/* "u.name" or "u.address.zip" (address = EmbeddedClassStateField) */
SimpleStateFieldPathExpression            ::= IdentificationVariable "." StateField
```

Cláusulas

```
SelectClause        ::= "SELECT" ["DISTINCT"] SelectExpression {"," SelectExpression}*
SimpleSelectClause  ::= "SELECT" ["DISTINCT"] SimpleSelectExpression
UpdateClause        ::= "UPDATE" AbstractSchemaName ["AS"] AliasIdentificationVariable "SET" UpdateIt
DeleteClause        ::= "DELETE" ["FROM"] AbstractSchemaName ["AS"] AliasIdentificationVariable
FromClause          ::= "FROM" IdentificationVariableDeclaration {"," IdentificationVariableDeclarati
SubselectFromClause ::= "FROM" SubselectIdentificationVariableDeclaration {"," SubselectIdentificatio
WhereClause         ::= "WHERE" ConditionalExpression
HavingClause        ::= "HAVING" ConditionalExpression
GroupByClause       ::= "GROUP" "BY" GroupByItem {"," GroupByItem}*
OrderByClause       ::= "ORDER" "BY" OrderByItem {"," OrderByItem}*
Subselect           ::= SimpleSelectClause SubselectFromClause [WhereClause] [GroupByClause] [HavingC
```

Elementos

```
UpdateItem  ::= IdentificationVariable "." (StateField | SingleValuedAssociationField) "=" NewValue
OrderByItem ::= (ResultVariable | StateFieldPathExpression) ["ASC" | "DESC"]
GroupByItem ::= IdentificationVariable | SingleValuedPathExpression
NewValue    ::= ScalarExpression | SimpleEntityExpression | "NULL"
```

From, Join e Index by

```
IdentificationVariableDeclaration          ::= RangeVariableDeclaration [IndexBy] {JoinVariableDeclar
SubselectIdentificationVariableDeclaration ::= IdentificationVariableDeclaration | (AssociationPathEx
JoinVariableDeclaration                    ::= Join [IndexBy]
RangeVariableDeclaration                   ::= AbstractSchemaName ["AS"] AliasIdentificationVariable
Join                                       ::= ["LEFT" ["OUTER"] | "INNER"] "JOIN" JoinAssociationPat
                                               ["AS"] AliasIdentificationVariable ["WITH" Conditional
IndexBy                                    ::= "INDEX" "BY" SimpleStateFieldPathExpression
```

Expresiones Select

```
SelectExpression       ::= IdentificationVariable | PartialObjectExpression | (AggregateExpression |
SimpleSelectExpression ::= ScalarExpression | IdentificationVariable |
                           (AggregateExpression [["AS"] AliasResultVariable])
PartialObjectExpression ::= "PARTIAL" IdentificationVariable "." PartialFieldSet
PartialFieldSet ::= "{" SimpleStateField {"," SimpleStateField}* "}"
```

Expresiones condicionales

```
ConditionalExpression       ::= ConditionalTerm {"OR" ConditionalTerm}*
ConditionalTerm             ::= ConditionalFactor {"AND" ConditionalFactor}*
ConditionalFactor           ::= ["NOT"] ConditionalPrimary
ConditionalPrimary          ::= SimpleConditionalExpression | "(" ConditionalExpression ")"
SimpleConditionalExpression ::= ComparisonExpression | BetweenExpression | LikeExpression |
                                InExpression | NullComparisonExpression | ExistsExpression |
                                EmptyCollectionComparisonExpression | CollectionMemberExpression
```

Expresiones de colección

```
EmptyCollectionComparisonExpression ::= CollectionValuedPathExpression "IS" ["NOT"] "EMPTY"
CollectionMemberExpression          ::= EntityExpression ["NOT"] "MEMBER" ["OF"] CollectionValuedPath
```

Valores literales

```
Literal     ::= string | char | integer | float | boolean
InParameter ::= Literal | InputParameter
```

Parámetros de entrada

```
InputParameter      ::= PositionalParameter | NamedParameter
PositionalParameter ::= "?" integer
NamedParameter      ::= ":" string
```

Expresiones aritméticas

```
ArithmeticExpression       ::= SimpleArithmeticExpression | "(" Subselect ")"
SimpleArithmeticExpression ::= ArithmeticTerm {("+" | "-") ArithmeticTerm}*
ArithmeticTerm             ::= ArithmeticFactor {("*" | "/") ArithmeticFactor}*
ArithmeticFactor           ::= [("+" | "-")] ArithmeticPrimary
ArithmeticPrimary          ::= SingleValuedPathExpression | Literal | "(" SimpleArithmeticExpression
                               | FunctionsReturningNumerics | AggregateExpression | FunctionsReturnin
                               | FunctionsReturningDatetime | IdentificationVariable | InputParameter
```

Expresiones escalares y de tipo

```
ScalarExpression       ::= SimpleArithmeticExpression | StringPrimary | DateTimePrimary | StateFieldP
                           BooleanPrimary | EntityTypeExpression | CaseExpression
StringExpression       ::= StringPrimary | "(" Subselect ")"
StringPrimary          ::= StateFieldPathExpression | string | InputParameter | FunctionsReturningStr
BooleanExpression      ::= BooleanPrimary | "(" Subselect ")"
BooleanPrimary         ::= StateFieldPathExpression | boolean | InputParameter
EntityExpression       ::= SingleValuedAssociationPathExpression | SimpleEntityExpression
SimpleEntityExpression ::= IdentificationVariable | InputParameter
DatetimeExpression     ::= DatetimePrimary | "(" Subselect ")"
DatetimePrimary        ::= StateFieldPathExpression | InputParameter | FunctionsReturningDatetime | A
```

Nota: Algunas partes de las expresiones `CASE` todavía no están implementadas.

Expresiones agregadas

```
AggregateExpression ::= ("AVG" | "MAX" | "MIN" | "SUM") "(" ["DISTINCT"] StateFieldPathExpression ")"
                        "COUNT" "(" ["DISTINCT"] (IdentificationVariable | SingleValuedPathExpressio
```

Case Expressions

```
CaseExpression        ::= GeneralCaseExpression | SimpleCaseExpression | CoalesceExpression | NullifE
GeneralCaseExpression ::= "CASE" WhenClause {WhenClause}* "ELSE" ScalarExpression "END"
WhenClause            ::= "WHEN" ConditionalExpression "THEN" ScalarExpression
SimpleCaseExpression  ::= "CASE" CaseOperand SimpleWhenClause {SimpleWhenClause}* "ELSE" ScalarExpres
CaseOperand           ::= StateFieldPathExpression | TypeDiscriminator
SimpleWhenClause      ::= "WHEN" ScalarExpression "THEN" ScalarExpression
CoalesceExpression    ::= "COALESCE" "(" ScalarExpression {"," ScalarExpression}* ")"
NullifExpression      ::= "NULLIF" "(" ScalarExpression "," ScalarExpression ")"
```

Otras expresiones

QUANTIFIED/BETWEEN/COMPARISON/LIKE/NULL/EXISTS

```
QuantifiedExpression     ::= ("ALL" | "ANY" | "SOME") "(" Subselect ")"
BetweenExpression        ::= ArithmeticExpression ["NOT"] "BETWEEN" ArithmeticExpression "AND" Arithm
ComparisonExpression     ::= ArithmeticExpression ComparisonOperator ( QuantifiedExpression | Arithme
InExpression             ::= StateFieldPathExpression ["NOT"] "IN" "(" (InParameter {"," InParameter]
LikeExpression           ::= StringExpression ["NOT"] "LIKE" string ["ESCAPE" char]
NullComparisonExpression ::= (SingleValuedPathExpression | InputParameter) "IS" ["NOT"] "NULL"
ExistsExpression         ::= ["NOT"] "EXISTS" "(" Subselect ")"
ComparisonOperator       ::= "=" | "<" | "<=" | "<>" | ">" | ">=" | "!="
```

Funciones

```
FunctionDeclaration ::= FunctionsReturningStrings | FunctionsReturningNumerics | FunctionsReturningDa

FunctionsReturningNumerics ::=
        "LENGTH" "(" StringPrimary ")" |
        "LOCATE" "(" StringPrimary "," StringPrimary ["," SimpleArithmeticExpression]")" |
        "ABS" "(" SimpleArithmeticExpression ")" | "SQRT" "(" SimpleArithmeticExpression ")" |
        "MOD" "(" SimpleArithmeticExpression "," SimpleArithmeticExpression ")" |
        "SIZE" "(" CollectionValuedPathExpression ")"

FunctionsReturningDateTime ::= "CURRENT_DATE" | "CURRENT_TIME" | "CURRENT_TIMESTAMP"

FunctionsReturningStrings ::=
        "CONCAT" "(" StringPrimary "," StringPrimary ")" |
        "SUBSTRING" "(" StringPrimary "," SimpleArithmeticExpression "," SimpleArithmeticExpression "
        "TRIM" "(" [["LEADING" | "TRAILING" | "BOTH"] [char] "FROM"] StringPrimary ")" |
        "LOWER" "(" StringPrimary ")" |
        "UPPER" "(" StringPrimary ")"
```

1.14 El generador de consultas

EL generador de consultas o `QueryBuilder` proporciona una *API* que está diseñada para construir condicionalmente una consulta `DQL` en varios pasos.

Proporciona un conjunto de clases y métodos que es capaz de construir consultas mediante programación, y también proporciona una fluida *API*. Esto significa que puedes cambiar entre una metodología u otra como quieras, y también escoger una, si lo prefieres.

1.14.1 Construyendo un nuevo objeto `QueryBuilder`

De la misma manera que creas una consulta normal, construir un objeto `QueryBuilder`, sólo proporciona el nombre correcto del método. He aquí un ejemplo de cómo construir un objeto `QueryBuilder`:

```
<?php
// $em instancia de EntityManager

// ejemplo1: creando una instancia del QueryBuilder
$qb = $em->createQueryBuilder();
```

Una vez que hayas creado una instancia del generador de consultas, esta proporciona un conjunto de útiles funciones informativas que puedes utilizar. Un buen ejemplo es el inspeccionar qué tipo de objeto es `QueryBuilder`.

```
<?php
// $qb instancia de QueryBuilder

// ejemplo2: recupera el tipo del QueryBuilder
echo $qb->getType(); // Imprime: 0
```

Actualmente hay tres posibles valores devueltos por `getType()`:

- `QueryBuilder::SELECT`, el cual devuelve un valor de 0
- `QueryBuilder::DELETE`, devuelve el valor 1
- `QueryBuilder::UPDATE`, el cual devuelve el valor 2

Es posible recuperar el `EntityManager` asociado del `QueryBuilder` actual, su `DQL` y también un objeto `Query` cuando hayas terminado de construir tu `DQL`.

```
<?php
// $qb instancia de QueryBuilder

// ejemplo3: recupera el EntityManager asociado
$em = $qb->getEntityManager();

// ejemplo4: recupera la cadena DQL definida en el QueryBuilder
$dql = $qb->getDql();

// ejemplo5: recupera el objeto Query asociado con la DQL procesada
$q = $qb->getQuery();
```

Internamente, `QueryBuilder` trabaja con una memoria caché `DQL` para incrementar el rendimiento. Cualquier cambio que pueda afectar la `DQL` generada en realidad modifica el estado del `QueryBuilder` a una etapa que llamamos `ESTADO\_SUCIO`. Un `QueryBuilder` puede estar en dos diferentes estados:

- `QueryBuilder::STATE_CLEAN`, lo cual significa que la `DQL` no ha sido alterada desde la última recuperación o nada se han añadido desde la creación de su instancia

- `QueryBuilder::STATE_DIRTY`, significa que la consulta `DQL` debe (y) se procesa en la siguiente recuperación

1.14.2 Trabajando con `QueryBuilder`

Todos los métodos ayudantes en `QueryBuilder` realmente confían en un solo: `add()`. Este método es responsable de construir cada pieza `DQL`. Este toma 3 parámetros: `$dqlPartName`, `$dqlPart` y `$append` (default=false)

- `$dqlPartName`: Donde se debe colocar el `$dqlPart`. Posibles valores: `select`, `from`, `groupBy`, `having`, `orderBy`
- `$dqlPart`: el cuál se debe colocar en `$dqlPartName`. Acepta una cadena o cualquier instancia de `Doctrine\ORM\Query\Expr\*`
- `$append`: Bandera opcional (`default=false`) si `$dqlPart` debe reemplazar todos los elementos definidos previamente en `$dqlPartName` o no
-

```
<?php
// $qb instancia de QueryBuilder

// ejemplo6: cómo definir: "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name ASC" using QueryBuil
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = ?1')
   ->add('orderBy', 'u.name ASC');
```

Enlazando parámetros a tu consulta

Doctrine admite el enlaces dinámicos de parámetros para tu consulta, similar a la preparación de consultas. Puedes utilizar ambos cadenas y números como marcadores de posición, aunque ambos tienen una sintaxis ligeramente diferente. Además, debes hacer tu elección: No está permitido mezclar ambos estilos. El enlace de parámetros sólo se puede lograr de la siguiente manera:

```
<?php
// $qb instancia de QueryBuilder

// ejemplo6: cómo definir: "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name ASC" using QueryBuil
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = ?1')
   ->add('orderBy', 'u.name ASC');
   ->setParameter(1, 100); // Fija de 1 a 100, y por lo tanto recuperaremos un usuario con u.id = 10
```

No estás obligado a enumerar tus marcadores de posición puesto que está disponible una sintaxis alternativa:

```
<?php
// $qb instancia de QueryBuilder

// ejemplo6: cómo definir: "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name ASC" usando el apoyo
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = :identifier')
   ->add('orderBy', 'u.name ASC');
   ->setParameter('identifier', 100); // Fija :identifier a 100, y por lo tanto recuperamos un usuari
```

Ten en cuenta que los marcadores de posición numéricos comienzan con un signo de interrogación (?) seguido por un número, mientras que los marcadores de posición nombrados comienzan con dos puntos (:) seguidos de una cadena.

Si tienes varios parámetros para enlazar a la consulta, también puedes utilizar `setParameters()` en lugar de `setParameter()` con la siguiente sintaxis:

```
<?php
// $qb instancia de QueryBuilder

// La consulta aquí...
$qb->setParameters(array(1 => 'value for ?1', 2 => 'value for ?2'));
```

La obtención de los parámetros enlazados es fácil - sólo tienes que utilizar la sintaxis mencionada con `getParameter()` o `getParameters()`:

```
<?php
// $qb instancia de QueryBuilder

// Ve el ejemplo anterior
$params = qb->getParameters(array(1, 2));
// Equivalente a
$param  = array($qb->getParameter(1), $qb->getParameter(2));
```

Nota: Si tratas de obtener un parámetro que no estaba vinculado, no obstante, `getParameter()` simplemente devuelve `NULL`.

Limitando el resultado

Para limitar el resultado, el generador de consultas tiene algunos métodos comunes con el objeto `Query` que puedes recuperar desde `EntityManager#createQuery()`.

```
<?php
// $qb instancia de QueryBuilder
$offset = (int)$_GET['offset'];
$limit = (int)$_GET['limit'];

$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('orderBy', 'u.name ASC')
   ->setFirstResult( $offset )
   ->setMaxResults( $limit );
```

Ejecutando una consulta

El generador de consultas sólo es un objeto generador, no significa que realmente tenga los medios para ejecutar la consulta. Además, un conjunto de parámetros tales como sugerencias de consulta no se puede establecer en el propio Generador de consultas. Es por eso que siempre tienes que convertir una instancia del generador de consultas en un objeto `Query`:

```
<?php
// $qb instancia de QueryBuilder
$query = $qb->getQuery();

// Fija opciones de consulta adicionales
$query->setQueryHint('foo', 'bar');
$query->useResultCache('my_cache_id');
```

```
// Ejecuta la consulta
$result = $query->getResult();
$single = $query->getSingleResult();
$array = $query->getArrayResult();
$scalar = $query->getScalarResult();
$singleScalar = $query->getSingleScalarResult();
```

Clases `Expr\*`

Cuando llamas a `add()` con una cadena, esta internamente evalúa a una instancia de la clase `Doctrine\ORM\Query\Expr\Expr\*`. Esta es la misma consulta del ejemplo 6 escrita usando clases `Doctrine\ORM\Query\Expr\Expr\*`:

```
<?php
// $qb instancia de QueryBuilder

// ejemplo7: cómo definir: "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name ASC" usando el genera
$qb->add('select', new Expr\Select(array('u')))
   ->add('from', new Expr\From('User', 'u'))
   ->add('where', new Expr\Comparison('u.id', '=', '?1'))
   ->add('orderBy', new Expr\OrderBy('u.name', 'ASC'));
```

Por supuesto, esta es la forma más difícil de crear una consulta `DQL` en *Doctrine*. Para simplificar un poco estos esfuerzos, presentamos lo que nosotros llamamos la clase ayudante `Expr`.

La clase `Expr`

To workaround some of the issues that `add()` method may cause, Doctrine created a class that can be considered as a helper for building expressions. This class is called `Expr`, which provides a set of useful methods to help build expressions:

```
<?php
// $qb instancia de QueryBuilder

// ejemplo8: portación al QueryBuilder de: "SELECT u FROM User u WHERE u.id = ? OR u.nickname LIKE ?
$qb->add('select', new Expr\Select(array('u')))
   ->add('from', new Expr\From('User', 'u'))
   ->add('where', $qb->expr()->orX(
       $qb->expr()->eq('u.id', '?1'),
       $qb->expr()->like('u.nickname', '?2')
   ))
   ->add('orderBy', new Expr\OrderBy('u.name', 'ASC'));
```

A pesar de que todavía suena complejo, la capacidad de crear condiciones mediante programación son la característica principal de `Expr`. Aquí está una lista completa de los métodos ayudantes apoyados:

```
<?php
class Expr
{
    /** Conditional objects **/

    // Example - $qb->expr()->andX($cond1 [, $condN])->add(...)->...
    public function andX($x = null); // Returns Expr\AndX instance

    // Example - $qb->expr()->orX($cond1 [, $condN])->add(...)->...
    public function orX($x = null); // Returns Expr\OrX instance
```

```
/** Comparison objects **/

// Example - $qb->expr()->eq('u.id', '?1') => u.id = ?1
public function eq($x, $y); // Returns Expr\Comparison instance

// Example - $qb->expr()->neq('u.id', '?1') => u.id <> ?1
public function neq($x, $y); // Returns Expr\Comparison instance

// Example - $qb->expr()->lt('u.id', '?1') => u.id < ?1
public function lt($x, $y); // Returns Expr\Comparison instance

// Example - $qb->expr()->lte('u.id', '?1') => u.id <= ?1
public function lte($x, $y); // Returns Expr\Comparison instance

// Example - $qb->expr()->gt('u.id', '?1') => u.id > ?1
public function gt($x, $y); // Returns Expr\Comparison instance

// Example - $qb->expr()->gte('u.id', '?1') => u.id >= ?1
public function gte($x, $y); // Returns Expr\Comparison instance

// Example - $qb->expr()->isNull('u.id') => u.id IS NULL
public function isNull($x); // Returns string

// Example - $qb->expr()->isNotNull('u.id') => u.id IS NOT NULL
public function isNotNull($x); // Returns string


/** Arithmetic objects **/

// Example - $qb->expr()->prod('u.id', '2') => u.id * 2
public function prod($x, $y); // Returns Expr\Math instance

// Example - $qb->expr()->diff('u.id', '2') => u.id - 2
public function diff($x, $y); // Returns Expr\Math instance

// Example - $qb->expr()->sum('u.id', '2') => u.id + 2
public function sum($x, $y); // Returns Expr\Math instance

// Example - $qb->expr()->quot('u.id', '2') => u.id / 2
public function quot($x, $y); // Returns Expr\Math instance


/** Pseudo-function objects **/

// Example - $qb->expr()->exists($qb2->getDql())
public function exists($subquery); // Returns Expr\Func instance

// Example - $qb->expr()->all($qb2->getDql())
public function all($subquery); // Returns Expr\Func instance

// Example - $qb->expr()->some($qb2->getDql())
public function some($subquery); // Returns Expr\Func instance

// Example - $qb->expr()->any($qb2->getDql())
public function any($subquery); // Returns Expr\Func instance

// Example - $qb->expr()->not($qb->expr()->eq('u.id', '?1'))
```

```
    public function not($restriction); // Returns Expr\Func instance

    // Example - $qb->expr()->in('u.id', array(1, 2, 3))
    // Make sure that you do NOT use something similar to $qb->expr()->in('value', array('stringvalue
    // En su lugar, usa $qb->expr()->in('value', array('?1')) y enlaza tus parámetros a ?1 (ve la se
    public function in($x, $y); // Devuelve una instancia de Expr\Func

    // Ejemplo - $qb->expr()->notIn('u.id', '2')
    public function notIn($x, $y); // Devuelve una instancia de Expr\Func

    // Ejemplo - $qb->expr()->like('u.firstname', $qb->expr()->literal('Gui%'))
    public function like($x, $y); // Devuelve Expr\Comparison instance

    // Ejemplo - $qb->expr()->between('u.id', '1', '10')
    public function between($val, $x, $y); // Devuelve Expr\Func


    /** Objetos función **/

    // Ejemplo - $qb->expr()->trim('u.firstname')
    public function trim($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->concat('u.firstname', $qb->expr()->concat(' ', 'u.lastname'))
    public function concat($x, $y); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->substr('u.firstname', 0, 1)
    public function substr($x, $from, $len); // Returns Expr\Func

    // Ejemplo - $qb->expr()->lower('u.firstname')
    public function lower($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->upper('u.firstname')
    public function upper($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->length('u.firstname')
    public function length($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->avg('u.age')
    public function avg($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->max('u.age')
    public function max($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->min('u.age')
    public function min($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->abs('u.currentBalance')
    public function abs($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->sqrt('u.currentBalance')
    public function sqrt($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->count('u.firstname')
    public function count($x); // Devuelve Expr\Func

    // Ejemplo - $qb->expr()->countDistinct('u.surname')
    public function countDistinct($x); // Devuelve Expr\Func
}
```

Métodos ayudantes

Hasta ahora hemos descrito el más bajo nivel (considerado como el método duro) de la creación de consultas. Puede ser útil trabajar a este nivel con fines de optimización, pero la mayoría de las veces es preferible trabajar a un nivel de abstracción más alto. Para simplificar aún más la forma de crear una consulta en *Doctrine*, podemos aprovechar lo que llamamos métodos ayudantes. Para todo el código base, hay un conjunto de métodos útiles para simplificar la vida del programador. Para ilustrar cómo trabajar con ellos, aquí está el mismo ejemplo 6 reescrito con métodos ayudantes del `QueryBuilder`:

```
<?php
// $qb instancia de QueryBuilder

// ejemplo9: cómo definir: "SELECT u FROM User u WHERE u.id = ?1 ORDER BY u.name ASC" usando métodos
$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name ASC');
```

Los métodos ayudantes del `QueryBuilder` se consideran la forma estándar para crear consultas `DQL`. A pesar de que es compatible, debes evitar usar consultas basadas en cadenas y te recomendamos usar métodos `$qb->expr()->*`. He aquí el ejemplo 8 convertido a la forma sugerida estándar para crear consultas:

```
<?php
// $qb instanceof QueryBuilder

// example8: QueryBuilder port of: "SELECT u FROM User u WHERE u.id = ?1 OR u.nickname LIKE ?2 ORDER
$qb->select(array('u')) // string 'u' is converted to array internally
   ->from('User', 'u')
   ->where($qb->expr()->orX(
       $qb->expr()->eq('u.id', '?1'),
       $qb->expr()->like('u.nickname', '?2')
   ))
   ->orderBy('u.surname', 'ASC'));
```

Aquí está una lista completa de los métodos ayudantes disponibles en el generador de consultas:

```
<?php
class QueryBuilder
{
    // Example - $qb->select('u')
    // Example - $qb->select(array('u', 'p'))
    // Example - $qb->select($qb->expr()->select('u', 'p'))
    public function select($select = null);

    // Example - $qb->delete('User', 'u')
    public function delete($delete = null, $alias = null);

    // Example - $qb->update('Group', 'g')
    public function update($update = null, $alias = null);

    // Example - $qb->set('u.firstName', $qb->expr()->literal('Arnold'))
    // Example - $qb->set('u.numChilds', 'u.numChilds + ?1')
    // Example - $qb->set('u.numChilds', $qb->expr()->sum('u.numChilds', '?1'))
    public function set($key, $value);

    // Example - $qb->from('Phonenumber', 'p')
    public function from($from, $alias = null);

    // Example - $qb->innerJoin('u.Group', 'g', Expr\Join::ON, $qb->expr()->and($qb->expr()->eq('u.g
```

```
    // Example - $qb->innerJoin('u.Group', 'g', 'ON', 'u.group_id = g.id AND g.name = ?1')
    public function innerJoin($join, $alias = null, $conditionType = null, $condition = null);

    // Example - $qb->leftJoin('u.Phonenumbers', 'p', Expr\Join::WITH, $qb->expr()->eq('p.area_code',
    // Example - $qb->leftJoin('u.Phonenumbers', 'p', 'WITH', 'p.area_code = 55')
    public function leftJoin($join, $alias = null, $conditionType = null, $condition = null);

    // NOTE: ->where() overrides all previously set conditions
    //
    // Example - $qb->where('u.firstName = ?1', $qb->expr()->eq('u.surname', '?2'))
    // Example - $qb->where($qb->expr()->andX($qb->expr()->eq('u.firstName', '?1'), $qb->expr()->eq('
    // Example - $qb->where('u.firstName = ?1 AND u.surname = ?2')
    public function where($where);

    // Example - $qb->andWhere($qb->expr()->orX($qb->expr()->lte('u.age', 40), 'u.numChild = 0'))
    public function andWhere($where);

    // Example - $qb->orWhere($qb->expr()->between('u.id', 1, 10));
    public function orWhere($where);

    // NOTE: -> groupBy() overrides all previously set grouping conditions
    //
    // Example - $qb->groupBy('u.id')
    public function groupBy($groupBy);

    // Example - $qb->addGroupBy('g.name')
    public function addGroupBy($groupBy);

    // NOTE: -> having() overrides all previously set having conditions
    //
    // Example - $qb->having('u.salary >= ?1')
    // Example - $qb->having($qb->expr()->gte('u.salary', '?1'))
    public function having($having);

    // Example - $qb->andHaving($qb->expr()->gt($qb->expr()->count('u.numChild'), 0))
    public function andHaving($having);

    // Example - $qb->orHaving($qb->expr()->lte('g.managerLevel', '100'))
    public function orHaving($having);

    // NOTE: -> orderBy() overrides all previously set ordering conditions
    //
    // Example - $qb->orderBy('u.surname', 'DESC')
    public function orderBy($sort, $order = null);

    // Example - $qb->addOrderBy('u.firstName')
    public function addOrderBy($sort, $order = null); // Default $order = 'ASC'
}
```

1.15 SQL nativo

Un `NativeQuery` te permite ejecutar declaraciones `SQL` nativas, asignando el resultado de acuerdo a tus especificaciones. Tal especificación describe cómo se asigna el conjunto de resultado `SQL` a cómo *Doctrine* representa el resultado por un `ResultSetMapping`. Este describe cómo debe asignar *Doctrine* el resultado de cada columna de la base de datos en términos del objeto gráfico. Esto te permite asignar código arbitrario `SQL` a objetos, tal como proveedor `SQL` altamente optimizado o procedimientos almacenados.

Nota: Si deseas ejecutar declaraciones `DELETE`, `UPDATE` o `INSERT` nativas de la *API* `SQL` no se pueden utilizar y probablemente arrojarán errores. Usa `EntityManager#getConnection()` para acceder a la conexión de base de datos nativa e invoca al método `executeUpdate()` para estas consultas.

1.15.1 La clase `NativeQuery`

Para crear un `NativeQuery` utiliza el método `EntityManager#createNativeQuery($sql, $resultSetMapping)`. Como puedes ver en la firma de este método, este espera dos ingredientes: la declaración `SQL` que deseas ejecutar y el `ResultSetMapping` que describe cómo se van a asignar los resultados.

Una vez que obtienes una instancia de un `NativeQuery`, puede enlazar los parámetros a la misma y finalmente su ejecución.

1.15.2 El `ResultSetMapping`

La clave para utilizar un `NativeQuery` es entender el `ResultSetMapping`. Un resultado *Doctrine* puede contener los siguientes componentes:

- Entidades resultantes. Estas representan los elementos raíz del resultado.
- Resultado de entidades unidas. Estos representan las entidades unidas en asociaciones resultantes de la entidad raíz.
- Resultados de campo. Estos representan una columna en el conjunto de resultados que se asigna a un campo de una entidad. El resultado de un campo siempre pertenece a una entidad o es resultado de una entidad unida.
- Resultados escalares. Estos representan los valores escalares en el conjunto de resultados que aparecen en cada fila del resultado. Añadir los resultados escalares a un `ResultSetMapping` también puede hacer que el resultado global se vuelva **mixto** (consulta `DQL` - El lenguaje de consulta de *Doctrine*) si el mismo `ResultSetMapping` también contiene los resultados de la entidad.
- Metaresultados. Estos representan las columnas que contienen metainformación, como claves externas y columnas discriminadoras. Al consultar objetos (con `getResult()`), todos los metadatos de las columnas de las entidades raíz o entidades unidas deben estar presentes en la consulta `SQL` y se asignan de acuerdo con `ResultSetMapping#addMetaResult`.

Nota: Posiblemente no te sorprenda que internamente *Doctrine* utiliza `ResultSetMapping` cuando creas consultas `DQL`. A medida que se analiza la consulta y se transforma en `SQL`, *Doctrine* llena un `ResultSetMapping` que describe cómo debe procesar los resultados la rutina de hidratación.

Ahora vamos a ver en detalle cada uno de los tipos de resultados que pueden aparecer en un `ResultSetMapping`.

Resultados entidad

Un resultado entidad describe un tipo de entidad que aparece como elemento raíz en el resultado transformado. Agregas un resultado entidad a través de `ResultSetMapping#addEntityResult()`. Vamos a echar un vistazo en detalle a la firma del método:

```
<?php
/**
 * Agrega un resultado entidad a este ''ResultSetMapping''.
 *
```

```
     * @param string $class El nombre de la clase de la entidad.
     * @param string $alias El alias para la clase. El alias debe ser único entre todas las entidades
     *                      resultantes o entidades unidas resultantes en este ResultSetMapping.
     */
    public function addEntityResult($class, $alias)
```

El primer parámetro es el nombre completamente cualificado de la clase entidad. El segundo parámetro es un alias arbitrario para esta entidad resultante el cual debe ser único dentro del `ResultSetMapping`. Puedes utilizar este alias para asignar los resultados de la entidad resultante. Es muy similar a una variable de identificación que utilizas en las clases de alias o relaciones `DQL`.

Una entidad resultante aislada no es suficiente para formar un `ResultSetMapping` válido. Una entidad resultante o resultado de la unión de entidades siempre necesita un conjunto de campo resultante, lo cual vamos a ver en breve.

Resultado de entidades unidas

El resultado una entidad unida describe un tipo de entidad que aparece como un elemento unido a la relación en el resultado transformado, unido a un resultado entidad (raíz). Tú agregas un resultado entidad unido a través de `ResultSetMapping#addJoinedEntityResult()`. Vamos a echar un vistazo en detalle a la firma del método:

```
<?php
/**
 * Añade una entidad unida al resultado.
 *
 * @param string $class El nombre de clase de la entidad unida.
 * @param string $alias El alias único a usar por la entidad unida.
 * @param string $parentAlias The alias of the entity result that is the parent of this joined result
 * @param object $relation The association field that connects the parent entity result with the join
 */
public function addJoinedEntityResult($class, $alias, $parentAlias, $relation)
```

El primer parámetro es el nombre de clase de la entidad unida. El segundo parámetro es un alias arbitrario para la entidad unida, el cual debe ser único dentro del `ResultSetMapping`. Puedes utilizar este alias para asignar los resultados a la entidad resultante. El tercer parámetro es el alias de la entidad resultante que es el tipo del padre de la relación unida. The fourth and last parameter is the name of the field on the parent entity result that should contain the joined entity result.

Field results

A field result describes the mapping of a single column in an SQL result set to a field in an entity. As such, field results are inherently bound to entity results. You add a field result through `ResultSetMapping#addFieldResult()`. Again, let's examine the method signature in detail:

```
<?php
/**
 * Adds a field result that is part of an entity result or joined entity result.
 *
 * @param string $alias The alias of the entity result or joined entity result.
 * @param string $columnName The name of the column in the SQL result set.
 * @param string $fieldName The name of the field on the (joined) entity.
 */
public function addFieldResult($alias, $columnName, $fieldName)
```

The first parameter is the alias of the entity result to which the field result will belong. The second parameter is the name of the column in the SQL result set. Note that this name is case sensitive, i.e. if you use a native query against

Oracle it must be all uppercase. The third parameter is the name of the field on the entity result identified by `$alias` into which the value of the column should be set.

Scalar results

A scalar result describes the mapping of a single column in an SQL result set to a scalar value in the Doctrine result. Scalar results are typically used for aggregate values but any column in the SQL result set can be mapped as a scalar value. To add a scalar result use `ResultSetMapping#addScalarResult()`. The method signature in detail:

```
<?php
/**
 * Adds a scalar result mapping.
 *
 * @param string $columnName The name of the column in the SQL result set.
 * @param string $alias The result alias with which the scalar result should be placed in the result
 */
public function addScalarResult($columnName, $alias)
```

The first parameter is the name of the column in the SQL result set and the second parameter is the result alias under which the value of the column will be placed in the transformed Doctrine result.

Meta results

A meta result describes a single column in an SQL result set that is either a foreign key or a discriminator column. These columns are essential for Doctrine to properly construct objects out of SQL result sets. To add a column as a meta result use `ResultSetMapping#addMetaResult()`. The method signature in detail:

```
<?php
/**
 * Agrega una metacolumna (clave externa o columna discriminadora) al conjunto resultante.
 *
 * @param string $alias
 * @param string $columnAlias
 * @param string $columnName
 */
public function addMetaResult($alias, $columnAlias, $columnName)
```

The first parameter is the alias of the entity result to which the meta column belongs. A meta result column (foreign key or discriminator column) always belongs to to an entity result. The second parameter is the column alias/name of the column in the SQL result set and the third parameter is the column name used in the mapping.

Discriminator Column

When joining an inheritance tree you have to give Doctrine a hint which meta-column is the discriminator column of this tree.

```
<?php
/**
 * Sets a discriminator column for an entity result or joined entity result.
 * The discriminator column will be used to determine the concrete class name to
 * instantiate.
 *
 * @param string $alias The alias of the entity result or joined entity result the discriminator
 *                      column should be used for.
 * @param string $discrColumn The name of the discriminator column in the SQL result set.
```

```
 */
public function setDiscriminatorColumn($alias, $discrColumn)
```

Ejemplos

Understanding a ResultSetMapping is probably easiest through looking at some examples.

First a basic example that describes the mapping of a single entity.

```
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User owns no associations.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');

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

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

El resultado sería el siguiente:

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

Note that this would be a partial object if the entity has more fields than just id and name. In the example above the column and field names are identical but that is not necessary, of course. Also note that the query string passed to createNativeQuery is **real native SQL**. Doctrine does not touch this SQL in any way.

In the previous basic example, a User had no relations and the table the class is mapped to owns no foreign keys. The next example assumes User has a unidirectional or bidirectional one-to-one association to a CmsAddress, where the User is the owning side and thus owns the foreign key.

```
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User owns an association to an Address but the Address is not loaded in the query.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addMetaResult('u', 'address_id', 'address_id');

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

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

Foreign keys are used by Doctrine for lazy-loading purposes when querying for objects. In the previous example, each user object in the result will have a proxy (a "ghost") in place of the address that contains the address_id. When the ghost proxy is accessed, it loads itself based on this key.

Consequently, associations that are *fetch-joined* do not require the foreign keys to be present in the SQL result set, only associations that are lazy.

```
<?php
// Equivalent DQL query: "select u from User u join u.address a WHERE u.name = ?1"
// User owns association to an Address and the Address is loaded in the query.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addJoinedEntityResult('Address' , 'a', 'u', 'address');
$rsm->addFieldResult('a', 'address_id', 'id');
$rsm->addFieldResult('a', 'street', 'street');
$rsm->addFieldResult('a', 'city', 'city');

$sql = 'SELECT u.id, u.name, a.id AS address_id, a.street, a.city FROM users u ' .
       'INNER JOIN address a ON u.address_id = a.id WHERE u.name = ?';
$query = $this->_em->createNativeQuery($sql, $rsm);
$query->setParameter(1, 'romanb');

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

In this case the nested entity `Address` is registered with the `ResultSetMapping#addJoinedEntityResult` method, which notifies Doctrine that this entity is not hydrated at the root level, but as a joined entity somewhere inside the object graph. In this case we specify the alias 'u' as third parameter and `address` as fourth parameter, which means the `Address` is hydrated into the `User::$address` property.

If a fetched entity is part of a mapped hierarchy that requires a discriminator column, this column must be present in the result set as a meta column so that Doctrine can create the appropriate concrete type. This is shown in the following example where we assume that there are one or more subclasses that extend User and either Class Table Inheritance or Single Table Inheritance is used to map the hierarchy (both use a discriminator column).

```
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User is a mapped base class for other classes. User owns no associations.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addMetaResult('u', 'discr', 'discr'); // discriminator column
$rsm->setDiscriminatorColumn('u', 'discr');

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

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

Note that in the case of Class Table Inheritance, an example as above would result in partial objects if any objects in the result are actually a subtype of User. When using DQL, Doctrine automatically includes the necessary joins for this mapping strategy but with native SQL it is your responsibility.

1.15.3 ResultSetMappingBuilder

There are some downsides with Native SQL queries. The primary one is that you have to adjust all result set mapping definitions if names of columns change. In DQL this is detected dynamically when the Query is regenerated with the current metadata.

To avoid this hassle you can use the `ResultSetMappingBuilder` class. It allows to add all columns of an entity to a result set mapping. To avoid clashes you can optionally rename specific columns when you are doing the same in your sQL statement:

```php
<?php

$sql = "SELECT u.id, u.name, a.id AS address_id, a.street, a.city " .
       "FROM users u INNER JOIN address a ON u.address_id = a.id";

$rsm = new ResultSetMappingBuilder;
$rsm->addRootEntityFromClassMetadata('MyProject\User', 'u');
$rsm->addJoinedEntityFromClassMetadata('MyProject\Address', 'a', array('id' => 'address_id'));
```

For entites with more columns the builder is very convenient to use. It extends the `ResultSetMapping` class and as such has all the functionality of it as well. Actualmente, el `ResultSetMappingBuilder` no cuenta con apoyo para entidades con herencia.

1.16 Change Tracking Policies

Change tracking is the process of determining what has changed in managed entities since the last time they were synchronized with the database.

Doctrine provides 3 different change tracking policies, each having its particular advantages and disadvantages. The change tracking policy can be defined on a per-class basis (or more precisely, per-hierarchy).

1.16.1 Deferred Implicit

The deferred implicit policy is the default change tracking policy and the most convenient one. With this policy, Doctrine detects the changes by a property-by-property comparison at commit time and also detects changes to entities or new entities that are referenced by other managed entities ("persistence by reachability"). Although the most convenient policy, it can have negative effects on performance if you are dealing with large units of work (see "Understanding the Unit of Work"). Since Doctrine can't know what has changed, it needs to check all managed entities for changes every time you invoke EntityManager#flush(), making this operation rather costly.

1.16.2 Deferred Explicit

The deferred explicit policy is similar to the deferred implicit policy in that it detects changes through a property-by-property comparison at commit time. The difference is that Doctrine 2 only considers entities that have been explicitly marked for change detection through a call to EntityManager#persist(entity) or through a save cascade. All other entities are skipped. This policy therefore gives improved performance for larger units of work while sacrificing the behavior of "automatic dirty checking".

Therefore, flush() operations are potentially cheaper with this policy. The negative aspect this has is that if you have a rather large application and you pass your objects through several layers for processing purposes and business tasks you may need to track yourself which entities have changed on the way so you can pass them to EntityManager#persist().

This policy can be configured as follows:

```php
<?php
/**
 * @Entity
 * @ChangeTrackingPolicy("DEFERRED_EXPLICIT")
 */
class User
{
    // ...
}
```

1.16.3 Notify

This policy is based on the assumption that the entities notify interested listeners of changes to their properties. For that purpose, a class that wants to use this policy needs to implement the `NotifyPropertyChanged` interface from the Doctrine namespace. As a guideline, such an implementation can look as follows:

```
<?php
use Doctrine\Common\NotifyPropertyChanged,
    Doctrine\Common\PropertyChangedListener;

/**
 * @Entity
 * @ChangeTrackingPolicy("NOTIFY")
 */
class MyEntity implements NotifyPropertyChanged
{
    // ...

    private $_listeners = array();

    public function addPropertyChangedListener(PropertyChangedListener $listener)
    {
        $this->_listeners[] = $listener;
    }
}
```

Then, in each property setter of this class or derived classes, you need to notify all the `PropertyChangedListener` instances. As an example we add a convenience method on `MyEntity` that shows this behaviour:

```
<?php
// ...

class MyEntity implements NotifyPropertyChanged
{
    // ...

    protected function _onPropertyChanged($propName, $oldValue, $newValue)
    {
        if ($this->_listeners) {
            foreach ($this->_listeners as $listener) {
                $listener->propertyChanged($this, $propName, $oldValue, $newValue);
            }
        }
    }

    public function setData($data)
    {
        if ($data != $this->data) {
            $this->_onPropertyChanged('data', $this->data, $data);
            $this->data = $data;
        }
    }
}
```

You have to invoke `_onPropertyChanged` inside every method that changes the persistent state of `MyEntity`.

The check whether the new value is different from the old one is not mandatory but recommended. That way you also have full control over when you consider a property changed.

The negative point of this policy is obvious: You need implement an interface and write some plumbing code. But also note that we tried hard to keep this notification functionality abstract. Strictly speaking, it has nothing to do with the persistence layer and the Doctrine ORM or DBAL. You may find that property notification events come in handy in many other scenarios as well. As mentioned earlier, the `Doctrine\Common` namespace is not that evil and consists solely of very small classes and interfaces that have almost no external dependencies (none to the DBAL and none to the ORM) and that you can easily take with you should you want to swap out the persistence layer. This change tracking policy does not introduce a dependency on the Doctrine DBAL/ORM or the persistence layer.

The positive point and main advantage of this policy is its effectiveness. It has the best performance characteristics of the 3 policies with larger units of work and a flush() operation is very cheap when nothing has changed.

1.17 Partial Objects

A partial object is an object whose state is not fully initialized after being reconstituted from the database and that is disconnected from the rest of its data. The following section will describe why partial objects are problematic and what the approach of Doctrine2 to this problem is.

Nota: The partial object problem in general does not apply to methods or queries where you do not retrieve the query result as objects. Examples are: `Query#getArrayResult()`, `Query#getScalarResult()`, `Query#getSingleScalarResult()`, etc.

1.17.1 What is the problem?

In short, partial objects are problematic because they are usually objects with broken invariants. As such, code that uses these partial objects tends to be very fragile and either needs to "know" which fields or methods can be safely accessed or add checks around every field access or method invocation. The same holds true for the internals, i.e. the method implementations, of such objects. You usually simply assume the state you need in the method is available, after all you properly constructed this object before you pushed it into the database, right? These blind assumptions can quickly lead to null reference errors when working with such partial objects.

It gets worse with the scenario of an optional association (0..1 to 1). When the associated field is NULL, you don't know whether this object does not have an associated object or whether it was simply not loaded when the owning object was loaded from the database.

These are reasons why many ORMs do not allow partial objects at all and instead you always have to load an object with all its fields (associations being proxied). One secure way to allow partial objects is if the programming language/platform allows the ORM tool to hook deeply into the object and instrument it in such a way that individual fields (not only associations) can be loaded lazily on first access. This is possible in Java, for example, through bytecode instrumentation. In PHP though this is not possible, so there is no way to have "secure" partial objects in an ORM with transparent persistence.

Doctrine, by default, does not allow partial objects. That means, any query that only selects partial object data and wants to retrieve the result as objects (i.e. `Query#getResult()`) will raise an exception telling you that partial objects are dangerous. If you want to force a query to return you partial objects, possibly as a performance tweak, you can use the `partial` keyword as follows:

```
<?php
$q = $em->createQuery("select partial u.{id,name} from MyApp\Domain\User u");
```

1.17.2 When should I force partial objects?

Mainly for optimization purposes, but be careful of premature optimization as partial objects lead to potentially more fragile code.

1.18 Asignación XML

El controlador de asignación XML te permite proveer metadatos al ORM en forma de documentos XML.

El controlador XML está respaldado por un esquema documento XML que describe la estructura de un documento de asignación. La versión más reciente del documento del esquema XML está disponible en línea en http://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd. In order to point to the latest version of the document of a particular stable release branch, just append the release number, i.e.: doctrine-mapping-2.0.xsd The most convenient way to work with XML mapping files is to use an IDE/editor that can provide code-completion based on such an XML Schema document. The following is an outline of a XML mapping document with the proper xmlns/xsi setup for the latest code in trunk.

```
<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">

    ...

</doctrine-mapping>
```

The XML mapping document of a class is loaded on-demand the first time it is requested and subsequently stored in the metadata cache. Para poder trabajar, esto requiere de ciertas convenciones:

- Each entity/mapped superclass must get its own dedicated XML mapping document.
- El nombre del documento de asignación debe consistir con el nombre completo de la clase, donde los separadores del espacio de nombres se sustituyen por puntos (.). For example an Entity with the fully qualified class-name "MyProject" would require a mapping file "MyProject.Entities.User.dcm.xml" unless the extension is changed.
- All mapping documents should get the extension ".dcm.xml" to identify it as a Doctrine mapping file. Esto no es más que una convención y no estás obligado a hacerlo. Puedes cambiar la extensión de archivo muy facilmente.
-

```
<?php
$driver->setFileExtension('.xml');
```

It is recommended to put all XML mapping documents in a single folder but you can spread the documents over several folders if you want to. In order to tell the XmlDriver where to look for your mapping documents, supply an array of paths as the first argument of the constructor, like this:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$driver = new \Doctrine\ORM\Mapping\Driver\XmlDriver(array('/path/to/files1', '/path/to/files2'));
$config->setMetadataDriverImpl($driver);
```

1.18.1 Ejemplo

Como una guía de inicio rápido, aquí está un pequeño documento de ejemplo que usa varios elementos comunes:

```
// Doctrine.Tests.ORM.Mapping.User.dcm.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://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">

        <indexes>
            <index name="name_idx" columns="name"/>
            <index columns="user_email"/>
        </indexes>

        <unique-constraints>
            <unique-constraint columns="name,user_email" name="search_idx" />
        </unique-constraints>

        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist" method="doStuffOnPrePersist"/>
            <lifecycle-callback type="prePersist" method="doOtherStuffOnPrePersistToo"/>
            <lifecycle-callback type="postPersist" method="doStuffOnPostPersist"/>
        </lifecycle-callbacks>

        <id name="id" type="integer" column="id">
            <generator strategy="AUTO"/>
            <sequence-generator sequence-name="tablename_seq" allocation-size="100" initial-value="1"
        </id>

        <field name="name" column="name" type="string" length="50" nullable="true" unique="true" />
        <field name="email" column="user_email" type="string" column-definition="CHAR(32) NOT NULL" /

        <one-to-one field="address" target-entity="Address" inversed-by="user">
            <cascade><cascade-remove /></cascade>
            <join-column name="address_id" referenced-column-name="id" on-delete="CASCADE" on-update=
        </one-to-one>

        <one-to-many field="phonenumbers" target-entity="Phonenumber" mapped-by="user">
            <cascade>
                <cascade-persist/>
            </cascade>
            <order-by>
                <order-by-field name="number" direction="ASC" />
            </order-by>
        </one-to-many>

        <many-to-many field="groups" target-entity="Group">
            <cascade>
                <cascade-all/>
            </cascade>
            <join-table name="cms_users_groups">
                <join-columns>
                    <join-column name="user_id" referenced-column-name="id" nullable="false" unique="
                </join-columns>
                <inverse-join-columns>
                    <join-column name="group_id" referenced-column-name="id" column-definition="INT N
                </inverse-join-columns>
            </join-table>
```

```
        </many-to-many>

    </entity>

</doctrine-mapping>
```

Be aware that class-names specified in the XML files should be fully qualified.

1.18.2 XML-Element Reference

The XML-Element reference explains all the tags and attributes that the Doctrine Mapping XSD Schema defines. You should read the Basic-, Association- and Inheritance Mapping chapters to understand what each of this definitions means in detail.

Defining an Entity

Each XML Mapping File contains the definition of one entity, specified as the `<entity />` element as a direct child of the `<doctrine-mapping />` element:

```
<doctrine-mapping>
    <entity name="MiProyecto\User" table="cms_users" repository-class="MiProyecto\UserRepository">
        <!-- definition here -->
    </entity>
</doctrine-mapping>
```

Atributos requeridos:

- name - The fully qualified class-name of the entity.

Atributos opcionales:

- **table** - The Table-Name to be used for this entity. Otherwise the Unqualified Class-Name is used by default.
- **repository-class** - The fully qualified class-name of an alternative `Doctrine\ORM\EntityRepository` implementation to be used with this entity.
- **inheritance-type** - El tipo de herencia, por omisión a `none`. A more detailed description follows in the *Defining Inheritance Mappings* section.
- **read-only** - (>= 2.1) Specifies that this entity is marked as read only and not considered for change-tracking. Entities of this type can be persisted and removed though.

Definiendo campos

Each entity class can contain zero to infinite fields that are managed by Doctrine. You can define them using the `<field />` element as a children to the `<entity />` element. The field element is only used for primitive types that are not the ID of the entity. For the ID mapping you have to use the `<id />` element.

```
<entity name="MiProyecto\User">

    <field name="name" type="string" length="50" />
    <field name="username" type="string" unique="true" />
    <field name="age" type="integer" nullable="true" />
    <field name="isActive" column="is_active" type="boolean" />
    <field name="weight" type="decimal" scale="5" precision="2" />
</entity>
```

Atributos requeridos:

- name - The name of the Property/Field on the given Entity PHP class.

Atributos opcionales:

- type - The `Doctrine\DBAL\Types\Type` name, defaults to "string"
- column - Name of the column in the database, defaults to the field name.
- length - The length of the given type, for use with strings only.
- unique - Should this field contain a unique value across the table? El valor predeterminado es `false`.
- nullable - Should this field allow NULL as a value? El valor predeterminado es falso.
- version - Should this field be used for optimistic locking? Only works on fields with type integer or datetime.
- scale - Scale of a decimal type.
- precision - Precision of a decimal type.
- column-definition - Optional alternative SQL representation for this column. This definition begin after the field-name and has to specify the complete column definition. Using this feature will turn this field dirty for Schema-Tool update commands at all times.

Defining Identity and Generator Strategies

An entity has to have at least one `<id />` element. For composite keys you can specify more than one id-element, however surrogate keys are recommended for use with Doctrine 2. The Id field allows to define properties of the identifier and allows a subset of the `<field />` element attributes:

```
<entity name="MiProyecto\User">
    <id name="id" type="integer" column="user_id" />
</entity>
```

Atributos requeridos:

- name - The name of the Property/Field on the given Entity PHP class.
- type - The `Doctrine\DBAL\Types\Type` name, preferably "string" or "integer".

Atributos opcionales:

- column - Name of the column in the database, defaults to the field name.

Using the simplified definition above Doctrine will use no identifier strategy for this entity. That means you have to manually set the identifier before calling `EntityManager#persist($entity)`. This is the so called `ASSIGNED` strategy.

If you want to switch the identifier generation strategy you have to nest a `<generator />` element inside the id-element. This of course only works for surrogate keys. For composite keys you always have to use the `ASSIGNED` strategy.

```
<entity name="MiProyecto\User">
    <id name="id" type="integer" column="user_id">
        <generator strategy="AUTO" />
    </id>
</entity>
```

The following values are allowed for the `<generator />` strategy attribute:

- AUTO - Automatic detection of the identifier strategy based on the preferred solution of the database vendor.

- IDENTITY - Use of a IDENTIFY strategy such as Auto-Increment IDs available to Doctrine AFTER the INSERT statement has been executed.
- SEQUENCE - Use of a database sequence to retrieve the entity-ids. This is possible before the INSERT statement is executed.

If you are using the SEQUENCE strategy you can define an additional element to describe the sequence:

```
<entity name="MiProyecto\User">
    <id name="id" type="integer" column="user_id">
        <generator strategy="SEQUENCE" />
        <sequence-generator sequence-name="user_seq" allocation-size="5" initial-value="1" />
    </id>
</entity>
```

Required attributes for `<sequence-generator />`:

- sequence-name - The name of the sequence

Optional attributes for `<sequence-generator />`:

- allocation-size - By how much steps should the sequence be incremented when a value is retrieved. Defaults to 1
- initial-value - What should the initial value of the sequence be.

  NOTA

  If you want to implement a cross-vendor compatible application you have to specify and additionally define the <sequence-generator /> element, if Doctrine chooses the sequence strategy for a platform.

Defining a Mapped Superclass

Sometimes you want to define a class that multiple entities inherit from, which itself is not an entity however. The chapter on *Inheritance Mapping* describes a Mapped Superclass in detail. You can define it in XML using the `<mapped-superclass />` tag.

```
<doctrine-mapping>
    <mapped-superclass name="MiProyecto\BaseClass">
        <field name="created" type="datetime" />
        <field name="updated" type="datetime" />
    </mapped-superclass>
</doctrine-mapping>
```

Atributos requeridos:

- name - Class name of the mapped superclass.

You can nest any number of `<field />` and unidirectional `<many-to-one />` or `<one-to-one />` associations inside a mapped superclass.

Defining Inheritance Mappings

There are currently two inheritance persistence strategies that you can choose from when defining entities that inherit from each other. Single Table inheritance saves the fields of the complete inheritance hierarchy in a single table, joined table inheritance creates a table for each entity combining the fields using join conditions.

You can specify the inheritance type in the `<entity />` element and then use the `<discriminator-column />` and `<discriminator-mapping />` attributes.

```
<entity name="MiProyecto\Animal" inheritance-type="JOINED">
    <discriminator-column name="discr" type="string" />
    <discriminator-map>
        <discriminator-mapping value="cat" class="MiProyecto\Cat" />
        <discriminator-mapping value="dog" class="MiProyecto\Dog" />
        <discriminator-mapping value="mouse" class="MiProyecto\Mouse" />
    </discriminator-map>
</entity>
```

The allowed values for inheritance-type attribute are `JOINED` or `SINGLE_TABLE`.

Nota: All inheritance related definitions have to be defined on the root entity of the hierarchy.

Defining Lifecycle Callbacks

You can define the lifecycle callback methods on your entities using the `<lifecycle-callbacks />` element:

```
<entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">

    <lifecycle-callbacks>
        <lifecycle-callback type="prePersist" method="onPrePersist" />
    </lifecycle-callbacks>
</entity>
```

Definiendo relaciones Uno-A-Uno

You can define One-To-One Relations/Associations using the `<one-to-one />` element. The required and optional attributes depend on the associations being on the inverse or owning side.

For the inverse side the mapping is as simple as:

```
<entity class="MiProyecto\User">
    <one-to-one field="address" target-entity="Address" mapped-by="user" />
</entity>
```

Required attributes for inverse One-To-One:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!
- mapped-by - Name of the field on the owning side (here Address entity) that contains the owning side association.

For the owning side this mapping would look like:

```
<entity class="MiProyecto\Address">
    <one-to-one field="user" target-entity="User" inversed-by="address" />
</entity>
```

Required attributes for owning One-to-One:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!

Optional attributes for owning One-to-One:

- inversed-by - If the association is bidirectional the inversed-by attribute has to be specified with the name of the field on the inverse entity that contains the back-reference.
- orphan-removal - If true, the inverse side entity is always deleted when the owning side entity is. El valor predeterminado es `false`.
- fetch - Either LAZY or EAGER, defaults to LAZY. This attribute makes only sense on the owning side, the inverse side *ALWAYS* has to use the `FETCH` strategy.

The definition for the owning side relies on a bunch of mapping defaults for the join column names. Without the nested `<join-column />` element Doctrine assumes to foreign key to be called `user_id` on the Address Entities table. This is because the `MyProject\Address` entity is the owning side of this association, which means it contains the foreign key.

The completed explicitly defined mapping is:

```
<entity class="MiProyecto\Address">
    <one-to-one field="user" target-entity="User" inversed-by="address">
        <join-column name="user_id" referenced-column-name="id" />
    </one-to-one>
</entity>
```

Defining Many-To-One Associations

The many-to-one association is *ALWAYS* the owning side of any bidirectional association. This simplifies the mapping compared to the one-to-one case. The minimal mapping for this association looks like:

```
<entity class="MiProyecto\Article">
    <many-to-one field="author" target-entity="User" />
</entity>
```

Atributos requeridos:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!

Atributos opcionales:

- inversed-by - If the association is bidirectional the inversed-by attribute has to be specified with the name of the field on the inverse entity that contains the back-reference.
- orphan-removal - If true the entity on the inverse side is always deleted when the owning side entity is and it is not connected to any other owning side entity anymore. El valor predeterminado es falso.
- fetch - Either LAZY or EAGER, defaults to LAZY.

This definition relies on a bunch of mapping defaults with regards to the naming of the join-column/foreign key. The explicitly defined mapping includes a `<join-column />` tag nested inside the many-to-one association tag:

```
<entity class="MiProyecto\Article">
    <many-to-one field="author" target-entity="User">
        <join-column name="author_id" referenced-column-name="id" />
    </many-to-one>
</entity>
```

The join-column attribute `name` specifies the column name of the foreign key and the `referenced-column-name` attribute specifies the name of the primary key column on the User entity.

Defining One-To-Many Associations

The one-to-many association is *ALWAYS* the inverse side of any association. There exists no such thing as a uni-directional one-to-many association, which means this association only ever exists for bi-directional associations.

```
<entity class="MiProyecto\User">
    <one-to-many field="phonenumbers" target-entity="Phonenumber" mapped-by="user" />
</entity>
```

Atributos requeridos:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!
- mapped-by - Name of the field on the owning side (here Phonenumber entity) that contains the owning side association.

Atributos opcionales:

- fetch - Either LAZY, EXTRA_LAZY or EAGER, defaults to LAZY.
- index-by: Index the collection by a field on the target entity.

Defining Many-To-Many Associations

From all the associations the many-to-many has the most complex definition. When you rely on the mapping defaults you can omit many definitions and rely on their implicit values.

```
<entity class="MiProyecto\User">
    <many-to-many field="groups" target-entity="Group" />
</entity>
```

Atributos requeridos:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!

Atributos opcionales:

- mapped-by - Name of the field on the owning side that contains the owning side association if the defined many-to-many association is on the inverse side.
- inversed-by - If the association is bidirectional the inversed-by attribute has to be specified with the name of the field on the inverse entity that contains the back-reference.
- fetch - Either LAZY, EXTRA_LAZY or EAGER, defaults to LAZY.
- index-by: Index the collection by a field on the target entity.

The mapping defaults would lead to a join-table with the name "User_Group" being created that contains two columns "user_id" and "group_id". The explicit definition of this mapping would be:

```
<entity class="MiProyecto\User">
    <many-to-many field="groups" target-entity="Group">
        <join-table name="cms_users_groups">
            <join-columns>
                <join-column name="user_id" referenced-column-name="id"/>
            </join-columns>
```

```
            <inverse-join-columns>
                <join-column name="group_id" referenced-column-name="id"/>
            </inverse-join-columns>
        </join-table>
    </many-to-many>
</entity>
```

Here both the <join-columns> and <inverse-join-columns> tags are necessary to tell Doctrine for which side the specified join-columns apply. These are nested inside a <join-table /> attribute which allows to specify the table name of the many-to-many join-table.

Cascade Element

Doctrine allows cascading of several UnitOfWork operations to related entities. You can specify the cascade operations in the <cascade /> element inside any of the association mapping tags.

```
<entity class="MiProyecto\User">
    <many-to-many field="groups" target-entity="Group">
        <cascade>
            <cascade-all/>
        </cascade>
    </many-to-many>
</entity>
```

Besides <cascade-all /> the following operations can be specified by their respective tags:

- <cascade-persist />
- <cascade-merge />
- <cascade-remove />
- <cascade-refresh />

Join Column Element

In any explicitly defined association mapping you will need the <join-column /> tag. It defines how the foreign key and primary key names are called that are used for joining two entities.

Atributos requeridos:

- name - The column name of the foreign key.
- referenced-column-name - The column name of the associated entities primary key

Atributos opcionales:

- unique - If the join column should contain a UNIQUE constraint. This makes sense for Many-To-Many join-columns only to simulate a one-to-many unidirectional using a join-table.
- nullable - should the join column be nullable, defaults to true.
- on-delete - Foreign Key Cascade action to perform when entity is deleted, defaults to NO ACTION/RESTRICT but can be set to “CASCADE”.

Defining Order of To-Many Associations

You can require one-to-many or many-to-many associations to be retrieved using an additional ORDER BY.

```
<entity class="MiProyecto\User">
    <many-to-many field="groups" target-entity="Group">
        <order-by>
            <order-by-field name="name" direction="ASC" />
        </order-by>
    </many-to-many>
</entity>
```

Defining Indexes or Unique Constraints

To define additional indexes or unique constraints on the entities table you can use the `<indexes />` and `<unique-constraints />` elements:

```
<entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">

    <indexes>
        <index name="name_idx" columns="name"/>
        <index columns="user_email"/>
    </indexes>

    <unique-constraints>
        <unique-constraint columns="name,user_email" name="search_idx" />
    </unique-constraints>
</entity>
```

Tienes que especificar la columna y no los nombres de campo de la clase entidad en el índice y definir las restricciones de unicidad.

Derived Entities ID syntax

If the primary key of an entity contains a foreign key to another entity we speak of a derived entity relationship. You can define this in XML with the "association-key" attribute in the `<id>` tag.

```
<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="Application\Model\ArticleAttribute">
        <id name="article" association-key="true" />
        <id name="attribute" type="string" />

        <field name="value" type="string" />

        <many-to-one field="article" target-entity="Article" inversed-by="attributes" />
     <entity>

</doctrine-mapping>
```

1.19 Asignación YAML

El controlador de asignación YAML te permite proporcionar metadatos al ORM en forma de documentos YAML.

La asignación de documentos YAML de una clase se carga bajo demanda la primera vez que se solicita y posteriormente se almacena en la caché de metadatos. Para poder trabajar, esto requiere de ciertas convenciones:

- Cada entidad/asignada como superclase debe tener su propia asignación a documento *YAML* dedicado.
- El nombre del documento de asignación debe consistir con el nombre completo de la clase, donde los separadores del espacio de nombres se sustituyen por puntos (.).
- Todas las asignaciones a documentos deben tener la extensión "dcm.yml." para identificarlos como archivos de asignación *Doctrine*. Esto no es más que una convención y no estás obligado a hacerlo. Puedes cambiar la extensión de archivo muy facilmente.
-

```
<?php
$driver->setFileExtension('.yml');
```

Te recomendamos poner todos los documentos de asignación YAML en un sólo directorio, pero puedes distribuir los documentos en varios directorios, si así lo deseas. A fin de informar a `YamlDriver` dónde buscar los documentos de asignación, suministra una matriz de rutas como primer argumento del constructor, de esta manera:

```
<?php
// $configura una instancia de Doctrine\ORM\Configuration
$driver = new YamlDriver(array('/ruta/a/tus/archivos'));
$config->setMetadataDriverImpl($driver);
```

1.19.1 Ejemplo

Como una guía de inicio rápido, aquí está un pequeño documento de ejemplo que usa varios elementos comunes:

```
# Doctrine.Tests.ORM.Mapping.User.dcm.yml
Doctrine\Tests\ORM\Mapping\User:
  type: entity
  table: cms_users
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
      length: 50
  oneToOne:
    address:
      targetEntity: Address
      joinColumn:
        name: address_id
        referencedColumnName: id
  oneToMany:
    phonenumbers:
      targetEntity: Phonenumber
      mappedBy: user
      cascade: ["persist", "merge"]
  manyToMany:
    groups:
      targetEntity: Group
      joinTable:
        name: cms_users_groups
```

```
        joinColumns:
          user_id:
            referencedColumnName: id
        inverseJoinColumns:
          group_id:
            referencedColumnName: id
  lifecycleCallbacks:
    prePersist: [ doStuffOnPrePersist, doOtherStuffOnPrePersistToo ]
    postPersist: [ doStuffOnPostPersist ]
```

Ten en cuenta que los nombres de clase especificado en los archivos YAML deben ser completamente cualificados.

1.20 Referencia de anotaciones

En este capítulo se da una referencia de cada anotación de *Doctrine* 2 con una breve explicación de su contexto y uso.

1.20.1 Índice

- *@Column*
- *@ChangeTrackingPolicy*
- *@DiscriminatorColumn*
- *@DiscriminatorMap*
- *@Entity*
- *@GeneratedValue*
- *@HasLifecycleCallbacks*
- *@Index*
- *@Id*
- *@InheritanceType*
- *@JoinColumn*
- *@JoinTable*
- *@ManyToOne*
- *@ManyToMany*
- *@MappedSuperclass*
- *@OneToOne*
- *@OneToMany*
- *@OrderBy*
- *@PostLoad*
- *@PostPersist*
- *@PostRemove*
- *@PostUpdate*
- *@PrePersist*

- *@PreRemove*
- *@PreUpdate*
- *@SequenceGenerator*
- *@Table*
- *@UniqueConstraint*
- *@Version*

1.20.2 Referencia

@Column

Marca una variable de instancia anotada como "persistente". Tiene que ser dentro del comentario *DocBlock* PHP de la instancia de la variable. Cualquier valor que tenga esta variable se guardará y cargará desde la base de datos como parte del ciclo de vida de las instancias de variables de la clase entidad.

Atributos requeridos:

- **type**: nombre del tipo *Doctrine* el cual se convierte entre PHP y la representación de la base de datos.

Atributos opcionales:

- **name**: By default the property name is used for the database column name also, however the 'name' attribute allows you to determine the column name.
- **length**: Used by the "string" type to determine its maximum length in the database. Doctrine does not validate the length of a string values for you.
- **precision**: The precision for a decimal (exact numeric) column (Applies only for decimal column)
- **scale**: The scale for a decimal (exact numeric) column (Applies only for decimal column)
- **unique**: Boolean value to determine if the value of the column should be unique across all rows of the underlying entities table.
- **nullable**: Determines if NULL values allowed for this column.
- **columnDefinition**: DDL SQL snippet that starts after the column name and specifies the complete (non-portable!) column definition. This attribute allows to make use of advanced RMDBS features. However you should make careful use of this feature and the consequences. SchemaTool will not detect changes on the column correctly anymore if you use "columnDefinition".

  Additionally you should remember that the "type" attribute still handles the conversion between PHP and Database values. If you use this attribute on a column that is used for joins between tables you should also take a look at *@JoinColumn*.

Ejemplos:

```
<?php
/**
 * @Column(type="string", length=32, unique=true, nullable=false)
 */
protected $username;

/**
 * @Column(type="string", columnDefinition="CHAR(2) NOT NULL")
 */
protected $country;
```

```
/**
 * @Column(type="decimal", precision=2, scale=1)
 */
protected $height;
```

@ChangeTrackingPolicy

The Change Tracking Policy annotation allows to specify how the Doctrine 2 UnitOfWork should detect changes in properties of entities during flush. By default each entity is checked according to a deferred implicit strategy, which means upon flush UnitOfWork compares all the properties of an entity to a previously stored snapshot. This works out of the box, however you might want to tweak the flush performance where using another change tracking policy is an interesting option.

The *details on all the available change tracking policies* can be found in the configuration section.

Ejemplo:

```
<?php
/**
 * @Entity
 * @ChangeTrackingPolicy("DEFERRED_IMPLICIT")
 * @ChangeTrackingPolicy("DEFERRED_EXPLICIT")
 * @ChangeTrackingPolicy("NOTIFY")
 */
class User {}
```

@DiscrimnatorColumn

This annotation is a required annotation for the topmost/super class of an inheritance hierarchy. It specifies the details of the column which saves the name of the class, which the entity is actually instantiated as.

Atributos requeridos:

- **name**: The column name of the discriminator. This name is also used during Array hydration as key to specify the class-name.

Atributos opcionales:

- **type**: Por omisión es una cadena.
- **length**: Por omisión esta es 255.

@DiscriminatorMap

The discriminator map is a required annotation on the top-most/super class in an inheritance hierarchy. It takes an array as only argument which defines which class should be saved under which name in the database. Keys are the database value and values are the classes, either as fully- or as unqualified class names depending if the classes are in the namespace or not.

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

```
class Person
{
    // ...
}
```

@Entity

Required annotation to mark a PHP class as Entity. Doctrine manages the persistence of all classes marked as entity.

Atributos opcionales:

- **repositoryClass**: Specifies the FQCN of a subclass of the Doctrine. Use of repositories for entities is encouraged to keep specialized DQL and SQL operations separated from the Model/Domain Layer.
- **readOnly**: (>= 2.1) Specifies that this entity is marked as read only and not considered for change-tracking. Entities of this type can be persisted and removed though.

Ejemplo:

```
<?php
/**
 * @Entity(repositoryClass="MiProyecto\UserRepository")
 */
class User
{
    //...
}
```

@GeneratedValue

Specifies which strategy is used for identifier generation for an instance variable which is annotated by *@Id*. This annotation is optional and only has meaning when used in conjunction with @Id.

If this annotation is not specified with @Id the NONE strategy is used as default.

Atributos requeridos:

- **strategy**: Set the name of the identifier generation strategy. Valid values are AUTO, SEQUENCE, TABLE, IDENTITY and NONE.

Ejemplo:

```
<?php
/**
 * @Id
 * @Column(type="integer")
 * @generatedValue(strategy="IDENTITY")
 */
protected $id = null;
```

@HasLifecycleCallbacks

Annotation which has to be set on the entity-class PHP DocBlock to notify Doctrine that this entity has entity life-cycle callback annotations set on at least one of its methods. Using @PostLoad, @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate or @PostUpdate without this marker annotation will make Doctrine ignore the callbacks.

Ejemplo:

```php
<?php
/**
 * @Entity
 * @HasLifecycleCallbacks
 */
class User
{
    /**
     * @PostPersist
     */
    public function sendOptinMail() {}
}
```

@Index

Anotación utilizada dentro de la anotación *@Table* a nivel de la clase entidad. It allows to hint the SchemaTool to generate a database index on the specified table columns. It only has meaning in the SchemaTool schema generation context.

Atributos requeridos:

- **name**: Name of the Index
- **columns**: Array of columns.

Ejemplo:

```php
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",indexes={@index(name="search_idx", columns={"name", "email"})})
 */
class ECommerceProduct
{
}
```

@Id

The annotated instance variable will be marked as entity identifier, the primary key in the database. This annotation is a marker only and has no required or optional attributes. For entities that have multiple identifier columns each column has to be marked with @Id.

Ejemplo:

```php
<?php
/**
 * @Id
 * @Column(type="integer")
 */
protected $id = null;
```

@InheritanceType

In an inheritance hierarchy you have to use this annotation on the topmost/super class to define which strategy should be used for inheritance. Currently Single Table and Class Table Inheritance are supported.

This annotation has always been used in conjunction with the *@DiscriminatorMap* and *@DiscriminatorColumn* annotations.

Ejemplos:

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

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

@JoinColumn

This annotation is used in the context of relations in *@ManyToOne*, *@OneToOne* fields and in the Context of *@JoinTable* nested inside a @ManyToMany. This annotation is not required. If its not specified the attributes *name* and *referencedColumnName* are inferred from the table and primary key names.

Atributos requeridos:

- **name**: Column name that holds the foreign key identifier for this relation. In the context of @JoinTable it specifies the column name in the join table.
- **referencedColumnName**: Name of the primary key identifier that is used for joining of this relation.

Atributos opcionales:

- **unique**: Determines if this relation exclusive between the affected entities and should be enforced so on the database constraint level. El valor predeterminado es `false`.
- **nullable**: Determine if the related entity is required, or if null is an allowed state for the relation. Defaults to true.
- **onDelete**: Cascade Action (Database-level)
- **columnDefinition**: DDL SQL snippet that starts after the column name and specifies the complete (non-portable!) column definition. This attribute allows to make use of advanced RMDBS features. Using this attribute on @JoinColumn is necessary if you need slightly different column definitions for joining columns, for example regarding NULL/NOT NULL defaults. However by default a "columnDefinition" attribute on *@Column* also sets the related @JoinColumn's columnDefinition. This is necessary to make foreign keys work.

Ejemplo:

```php
<?php
/**
 * @OneToOne(targetEntity="Customer")
 * @JoinColumn(name="customer_id", referencedColumnName="id")
 */
private $customer;
```

@JoinColumns

An array of @JoinColumn annotations for a *@ManyToOne* or *@OneToOne* relation with an entity that has multiple identifiers.

@JoinTable

Using *@OneToMany* or *@ManyToMany* on the owning side of the relation requires to specify the @JoinTable annotation which describes the details of the database join table. If you do not specify @JoinTable on these relations reasonable mapping defaults apply using the affected table and the column names.

Atributos requeridos:

- **name**: Database name of the join-table
- **joinColumns**: An array of @JoinColumn annotations describing the join-relation between the owning entities table and the join table.
- **inverseJoinColumns**: An array of @JoinColumn annotations describing the join-relation between the inverse entities table and the join table.

Ejemplo:

```php
<?php
/**
 * @ManyToMany(targetEntity="Phonenumber")
 * @JoinTable(name="users_phonenumbers",
 *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
 *      inverseJoinColumns={@JoinColumn(name="phonenumber_id", referencedColumnName="id", unique=true
 * )
 */
public $phonenumbers;
```

@ManyToOne

Defines that the annotated instance variable holds a reference that describes a many-to-one relationship between two entities.

Atributos requeridos:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!

Atributos opcionales:

- **cascade**: Cascade Option
- **fetch**: One of LAZY or EAGER
- inversedBy - The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.

Ejemplo:

```
<?php
/**
 * @ManyToOne(targetEntity="Cart", cascade={"all"}, fetch="EAGER")
 */
private $cart;
```

@ManyToMany

Defines an instance variable holds a many-to-many relationship between two entities. *@JoinTable* is an additional, optional annotation that has reasonable default configuration values using the table and names of the two related entities.

Atributos requeridos:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!

Atributos opcionales:

- **mappedBy**: This option specifies the property name on the targetEntity that is the owning side of this relation. Its a required attribute for the inverse side of a relationship.
- **inversedBy**: The inversedBy attribute designates the eld in the entity that is the inverse side of the relationship.
- **cascade**: Cascade Option
- **fetch**: One of LAZY, EXTRA_LAZY or EAGER
- **indexBy**: Index the collection by a field on the target entity.

Nota: For ManyToMany bidirectional relationships either side may be the owning side (the side that defines the @JoinTable and/or does not make use of the mappedBy attribute, thus using a default join table).

Ejemplo:

```
<?php
/**
 * Owning Side
 *
 * @ManyToMany(targetEntity="Group", inversedBy="features")
 * @JoinTable(name="user_groups",
 *       joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
 *       inverseJoinColumns={@JoinColumn(name="group_id", referencedColumnName="id")}
 *       )
 */
private $groups;

/**
 * Inverse Side
 *
 * @ManyToMany(targetEntity="User", mappedBy="groups")
 */
private $features;
```

@MappedSuperclass

Una asignación de superclase es una clase abstracta o concreta que proporciona estado persistente a la entidad y la información de asignación de sus subclases, pero que en sí misma no es una entidad. This annotation is specified on the Class docblock and has no additional attributes.

The @MappedSuperclass annotation cannot be used in conjunction with @Entity. See the Inheritance Mapping section for *more details on the restrictions of mapped superclasses*.

@OneToOne

The @OneToOne annotation works almost exactly as the *@ManyToOne* with one additional option that can be specified. The configuration defaults for *@JoinColumn* using the target entity table and primary key column names apply here too.

Atributos requeridos:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!

Atributos opcionales:

- **cascade**: Cascade Option
- **fetch**: One of LAZY or EAGER
- **orphanRemoval**: Boolean that specifies if orphans, inverse OneToOne entities that are not connected to any owning instance, should be removed by Doctrine. El valor predeterminado es `false`.
- **inversedBy**: The inversedBy attribute designates the eld in the entity that is the inverse side of the relationship.

Ejemplo:

```
<?php
/**
 * @OneToOne(targetEntity="Customer")
 * @JoinColumn(name="customer_id", referencedColumnName="id")
 */
private $customer;
```

@OneToMany

Atributos requeridos:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANTE:* ¡Sin barra diagonal inversa inicial!

Atributos opcionales:

- **cascade**: Cascade Option
- **orphanRemoval**: Boolean that specifies if orphans, inverse OneToOne entities that are not connected to any owning instance, should be removed by Doctrine. El valor predeterminado es `false`.
- **mappedBy**: This option specifies the property name on the targetEntity that is the owning side of this relation. Its a required attribute for the inverse side of a relationship.
- **fetch**: One of LAZY, EXTRA_LAZY or EAGER.
- **indexBy**: Index the collection by a field on the target entity.

Ejemplo:

```
<?php
/**
 * @OneToMany(targetEntity="Phonenumber", mappedBy="user", cascade={"persist", "remove", "merge"}, o
 */
public $phonenumbers;
```

@OrderBy

Optional annotation that can be specified with a *@ManyToMany* or *@OneToMany* annotation to specify by which criteria the collection should be retrieved from the database by using an ORDER BY clause.

This annotation requires a single non-attributed value with an DQL snippet:

Ejemplo:

```
<?php
/**
 * @ManyToMany(targetEntity="Group")
 * @OrderBy({"name" = "ASC"})
 */
private $groups;
```

El fragmento `DQL` en `OrderBy` sólo puede consistir de nombres de campo no calificados, sin comillas y de una opcional declaración `ASC/DESC`. Múltiples campos van separados por una coma (,). Los nombres de los campos referidos existentes en la clase `targetEntity` de la anotación `@ManyToMany` o `@OneToMany`.

@PostLoad

Marks a method on the entity to be called as a @PostLoad event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostPersist

Marks a method on the entity to be called as a @PostPersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostRemove

Marks a method on the entity to be called as a @PostRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostUpdate

Marks a method on the entity to be called as a @PostUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PrePersist

Marks a method on the entity to be called as a @PrePersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PreRemove

Marks a method on the entity to be called as a @PreRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PreUpdate

Marks a method on the entity to be called as a @PreUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@SequenceGenerator

For the use with @generatedValue(strategy="SEQUENCE") this annotation allows to specify details about the sequence, such as the increment size and initial values of the sequence.

Atributos requeridos:

- **sequenceName**: Name of the sequence

Atributos opcionales:

- **allocationSize**: Increment the sequence by the allocation size when its fetched. A value larger than 1 allows to optimize for scenarios where you create more than one new entity per request. Defaults to 10
- **initialValue**: Where does the sequence start, defaults to 1.

Ejemplo:

```
<?php
/**
 * @Id
 * @GeneratedValue(strategy="SEQUENCE")
 * @Column(type="integer")
 * @SequenceGenerator(sequenceName="tablename_seq", initialValue=1, allocationSize=100)
 */
protected $id = null;
```

@Table

Annotation describes the table an entity is persisted in. It is placed on the entity-class PHP DocBlock and is optional. If it is not specified the table name will default to the entities unqualified classname.

Atributos requeridos:

- **name**: Name of the table

Atributos opcionales:

- **indexes**: Array of @Index annotations
- **uniqueConstraints**: Array of @UniqueConstraint annotations.

Ejemplo:

```
<?php
/**
 * @Entity
 * @Table(name="user",
 *      uniqueConstraints={@UniqueConstraint(name="user_unique",columns={"username"})},
```

```
 *      indexes={@Index(name="user_idx", columns={"email"})}
 * )
 */
class User { }
```

@UniqueConstraint

Anotación utilizada dentro de la anotación *@Table* a nivel de la clase entidad. It allows to hint the SchemaTool to generate a database unique constraint on the specified table columns. It only has meaning in the SchemaTool schema generation context.

Atributos requeridos:

- **name**: Name of the Index
- **columns**: Array of columns.

Ejemplo:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx", columns=
 */
class ECommerceProduct
{
}
```

@Version

Marker annotation that defines a specified column as version attribute used in an optimistic locking scenario. It only works on *@Column* annotations that have the type integer or datetime. Combining @Version with *@Id* is not supported.

Ejemplo:

```
<?php
/**
 * @column(type="integer")
 * @version
 */
protected $version;
```

1.21 PHP Mapping

Doctrine 2 also allows you to provide the ORM metadata in the form of plain PHP code using the `ClassMetadata` API. You can write the code in PHP files or inside of a static function named `loadMetadata($class)` on the entity class itself.

1.21.1 PHP Files

If you wish to write your mapping information inside PHP files that are named after the entity and included to populate the metadata for an entity you can do so by using the `PHPDriver`:

```
<?php
$driver = new PHPDriver('/path/to/php/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

Now imagine we had an entity named `Entities\User` and we wanted to write a mapping file for it using the above configured `PHPDriver` instance:

```
<?php
namespace Entities;

class User
{
    private $id;
    private $username;
}
```

To write the mapping information you just need to create a file named `Entities.User.php` inside of the `/path/to/php/mapping/files` folder:

```
<?php
// /path/to/php/mapping/files/Entities.User.php

$metadata->mapField(array(
   'id' => true,
   'fieldName' => 'id',
   'type' => 'integer'
));

$metadata->mapField(array(
   'fieldName' => 'username',
   'type' => 'string'
));
```

Now we can easily retrieve the populated `ClassMetadata` instance where the `PHPDriver` includes the file and the `ClassMetadataFactory` caches it for later retrieval:

```
<?php
$class = $em->getClassMetadata('Entities\User');
// or
$class = $em->getMetadataFactory()->getMetadataFor('Entities\User');
```

1.21.2 Static Function

In addition to the PHP files you can also specify your mapping information inside of a static function defined on the entity class itself. This is useful for cases where you want to keep your entity and mapping information together but don't want to use annotations. For this you just need to use the `StaticPHPDriver`:

```
<?php
$driver = new StaticPHPDriver('/path/to/entities');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

Now you just need to define a static function named `loadMetadata($metadata)` on your entity:

```
<?php
namespace Entities;

use Doctrine\ORM\Mapping\ClassMetadata;
```

```
class User
{
    // ...

    public static function loadMetadata(ClassMetadata $metadata)
    {
        $metadata->mapField(array(
           'id' => true,
           'fieldName' => 'id',
           'type' => 'integer'
        ));

        $metadata->mapField(array(
           'fieldName' => 'username',
           'type' => 'string'
        ));
    }
}
```

1.21.3 ClassMetadataInfo API

The `ClassMetadataInfo` class is the base data object for storing the mapping metadata for a single entity. It contains all the getters and setters you need populate and retrieve information for an entity.

General Setters

- `setTableName($tableName)`
- `setPrimaryTable(array $primaryTableDefinition)`
- `setCustomRepositoryClass($repositoryClassName)`
- `setIdGeneratorType($generatorType)`
- `setIdGenerator($generator)`
- `setSequenceGeneratorDefinition(array $definition)`
- `setChangeTrackingPolicy($policy)`
- `setIdentifier(array $identifier)`

Inheritance Setters

- `setInheritanceType($type)`
- `setSubclasses(array $subclasses)`
- `setParentClasses(array $classNames)`
- `setDiscriminatorColumn($columnDef)`
- `setDiscriminatorMap(array $map)`

Field Mapping Setters

- `mapField(array $mapping)`
- `mapOneToOne(array $mapping)`
- `mapOneToMany(array $mapping)`
- `mapManyToOne(array $mapping)`
- `mapManyToMany(array $mapping)`

Lifecycle Callback Setters

- `addLifecycleCallback($callback, $event)`
- `setLifecycleCallbacks(array $callbacks)`

Versioning Setters

- `setVersionMapping(array &$mapping)`
- `setVersioned($bool)`
- `setVersionField()`

General Getters

- `getTableName()`
- `getTemporaryIdTableName()`

Identifier Getters

- `getIdentifierColumnNames()`
- `usesIdGenerator()`
- `isIdentifier($fieldName)`
- `isIdGeneratorIdentity()`
- `isIdGeneratorSequence()`
- `isIdGeneratorTable()`
- `isIdentifierNatural()`
- `getIdentifierFieldNames()`
- `getSingleIdentifierFieldName()`
- `getSingleIdentifierColumnName()`

Inheritance Getters

- `isInheritanceTypeNone()`
- `isInheritanceTypeJoined()`
- `isInheritanceTypeSingleTable()`
- `isInheritanceTypeTablePerClass()`
- `isInheritedField($fieldName)`
- `isInheritedAssociation($fieldName)`

Change Tracking Getters

- `isChangeTrackingDeferredExplicit()`
- `isChangeTrackingDeferredImplicit()`
- `isChangeTrackingNotify()`

Field & Association Getters

- `isUniqueField($fieldName)`
- `isNullable($fieldName)`
- `getColumnName($fieldName)`
- `getFieldMapping($fieldName)`
- `getAssociationMapping($fieldName)`
- `getAssociationMappings()`
- `getFieldName($columnName)`
- `hasField($fieldName)`
- `getColumnNames(array $fieldNames = null)`
- `getTypeOfField($fieldName)`
- `getTypeOfColumn($columnName)`
- `hasAssociation($fieldName)`
- `isSingleValuedAssociation($fieldName)`
- `isCollectionValuedAssociation($fieldName)`

Lifecycle Callback Getters

- `hasLifecycleCallbacks($lifecycleEvent)`
- `getLifecycleCallbacks($event)`

1.21.4 ClassMetadata API

The `ClassMetadata` class extends `ClassMetadataInfo` and adds the runtime functionality required by Doctrine. It adds a few extra methods related to runtime reflection for working with the entities themselves.

- `getReflectionClass()`
- `getReflectionProperties()`
- `getReflectionProperty($name)`
- `getSingleIdReflectionProperty()`
- `getIdentifierValues($entity)`
- `setIdentifierValues($entity, $id)`
- `setFieldValue($entity, $field, $value)`
- `getFieldValue($entity, $field)`

1.22 Memoria caché

Doctrine provides cache drivers in the `Common` package for some of the most popular caching implementations such as APC, Memcache and Xcache. We also provide an `ArrayCache` driver which stores the data in a PHP array. Obviously, the cache does not live between requests but this is useful for testing in a development environment.

1.22.1 Cache Drivers

The cache drivers follow a simple interface that is defined in `Doctrine\Common\Cache\Cache`. All the cache drivers extend a base class `Doctrine\Common\Cache\AbstractCache` which implements the before mentioned interface.

The interface defines the following methods for you to publicly use.

- 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.

Each driver extends the `AbstractCache` class which defines a few abstract protected methods that each of the drivers must implement.

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

The public methods `fetch()`, `contains()`, etc. utilize the above protected methods that are implemented by the drivers. The code is organized this way so that the protected methods in the drivers do the raw interaction with the cache implementation and the `AbstractCache` can build custom functionality on top of these methods.

APC

In order to use the APC cache driver you must have it compiled and enabled in your php.ini. You can read about APC in the PHP Documentation. It will give you a little background information about what it is and how you can use it as well as how to install it.

Below is a simple example of how you could use the APC cache driver by itself.

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

Memcache

In order to use the Memcache cache driver you must have it compiled and enabled in your php.ini. You can read about Memcache ‘ on the PHP website <http://us2.php.net/memcache>‘_. It will give you a little background information about what it is and how you can use it as well as how to install it.

Below is a simple example of how you could use the Memcache cache driver by itself.

```
<?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

In order to use the Xcache cache driver you must have it compiled and enabled in your php.ini. You can read about Xcache here. It will give you a little background information about what it is and how you can use it as well as how to install it.

Below is a simple example of how you could use the Xcache cache driver by itself.

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

1.22.2 Using Cache Drivers

In this section we’ll describe how you can fully utilize the API of the cache drivers to save cache, check if some cache exists, fetch the cached data and delete the cached data. We’ll use the `ArrayCache` implementation as our example here.

```
<?php
$cacheDriver = new \Doctrine\Common\Cache\ArrayCache();
```

Saving

To save some data to the cache driver it is as simple as using the `save()` method.

```
<?php
$cacheDriver->save('cache_id', 'my_data');
```

The `save()` method accepts three arguments which are described below.

- `$id` - The cache id
- `$data` - The cache entry/data.
- `$lifeTime` - The lifetime. If != false, sets a specific lifetime for this cache entry (null => infinite lifeTime).

You can save any type of data whether it be a string, array, object, etc.

```
<?php
$array = array(
    'key1' => 'value1',
    'key2' => 'value2'
);
$cacheDriver->save('my_array', $array);
```

Checking

Checking whether some cache exists is very simple, just use the `contains()` method. It accepts a single argument which is the ID of the cache entry.

```
<?php
if ($cacheDriver->contains('cache_id')) {
    echo 'cache exists';
} else {
    echo 'cache does not exist';
}
```

Fetching

Now if you want to retrieve some cache entry you can use the `fetch()` method. It also accepts a single argument just like `contains()` which is the ID of the cache entry.

```
<?php
$array = $cacheDriver->fetch('my_array');
```

Deleting

As you might guess, deleting is just as easy as saving, checking and fetching. We have a few ways to delete cache entries. You can delete by an individual ID, regular expression, prefix, suffix or you can delete all entries.

By Cache ID

```
<?php
$cacheDriver->delete('my_array');
```

You can also pass wild cards to the `delete()` method and it will return an array of IDs that were matched and deleted.

```
<?php
$deleted = $cacheDriver->delete('users_*');
```

By Regular Expression

If you need a little more control than wild cards you can use a PHP regular expression to delete cache entries.

```
<?php
$deleted = $cacheDriver->deleteByRegex('/users_.*/');
```

By Prefix

Because regular expressions are kind of slow, if simply deleting by a prefix or suffix is sufficient, it is recommended that you do that instead of using a regular expression because it will be much faster if you have many cache entries.

```
<?php
$deleted = $cacheDriver->deleteByPrefix('users_');
```

By Suffix

Just like we did above with the prefix you can do the same with a suffix.

```
<?php
$deleted = $cacheDriver->deleteBySuffix('_my_account');
```

All

If you simply want to delete all cache entries you can do so with the `deleteAll()` method.

```
<?php
$deleted = $cacheDriver->deleteAll();
```

Counting

If you want to count how many entries are stored in the cache driver instance you can use the `count()` method.

```
<?php
echo $cacheDriver->count();
```

Nota: In order to use `deleteByRegex()`, `deleteByPrefix()`, `deleteBySuffix()`, `deleteAll()`, `count()` or `getIds()` you must enable an option for the cache driver to manage your cache IDs internally. This is necessary because APC, Memcache, etc. don't have any advanced functionality for fetching and deleting. We add some functionality on top of the cache drivers to maintain an index of all the IDs stored in the cache driver so that we can allow more granular deleting operations.

```
<?php
$cacheDriver->setManageCacheIds(true);
```

Namespaces

If you heavily use caching in your application and utilize it in multiple parts of your application, or use it in different applications on the same server you may have issues with cache naming collisions. This can be worked around by using namespaces. You can set the namespace a cache driver should use by using the `setNamespace()` method.

```
<?php
$cacheDriver->setNamespace('my_namespace_');
```

1.22.3 Integrating with the ORM

The Doctrine ORM package is tightly integrated with the cache drivers to allow you to improve performance of various aspects of Doctrine by just simply making some additional configurations and method calls.

Query Cache

It is highly recommended that in a production environment you cache the transformation of a DQL query to its SQL counterpart. It doesn't make sense to do this parsing multiple times as it doesn't change unless you alter the DQL query.

This can be done by configuring the query cache implementation to use on your ORM configuration.

```
<?php
$config = new \Doctrine\ORM\Configuration();
$config->setQueryCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Result Cache

The result cache can be used to cache the results of your queries so that we don't have to query the database or hydrate the data again after the first time. You just need to configure the result cache implementation.

```
<?php
$config->setResultCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Now when you're executing DQL queries you can configure them to use the result cache.

```
<?php
$query = $em->createQuery('select u from \Entities\User u');
$query->useResultCache(true);
```

You can also configure an individual query to use a different result cache driver.

```
<?php
$query->setResultCacheDriver(new \Doctrine\Common\Cache\ApcCache());
```

Nota: Setting the result cache driver on the query will automatically enable the result cache for the query. If you want to disable it pass false to `useResultCache()`.

```
<?php
$query->useResultCache(false);
```

If you want to set the time the cache has to live you can use the `setResultCacheLifetime()` method.

```
<?php
$query->setResultCacheLifetime(3600);
```

The ID used to store the result set cache is a hash which is automatically generated for you if you don't set a custom ID yourself with the `setResultCacheId()` method.

```
<?php
$query->setResultCacheId('my_custom_id');
```

You can also set the lifetime and cache ID by passing the values as the second and third argument to `useResultCache()`.

```
<?php
$query->useResultCache(true, 3600, 'my_custom_id');
```

Metadata Cache

Your class metadata can be parsed from a few different sources like YAML, XML, Annotations, etc. Instead of parsing this information on each request we should cache it using one of the cache drivers.

Just like the query and result cache we need to configure it first.

```
<?php
$config->setMetadataCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Now the metadata information will only be parsed once and stored in the cache driver.

1.22.4 Clearing the Cache

We've already shown you previously how you can use the API of the cache drivers to manually delete cache entries. For your convenience we offer a command line task for you to help you with clearing the query, result and metadata cache.

From the Doctrine command line you can run the following command.

```
$ ./doctrine clear-cache
```

Running this task with no arguments will clear all the cache for all the configured drivers. If you want to be more specific about what you clear you can use the following options.

To clear the query cache use the `--query` option.

```
$ ./doctrine clear-cache --query
```

To clear the metadata cache use the `--metadata` option.

```
$ ./doctrine clear-cache --metadata
```

To clear the result cache use the `--result` option.

```
$ ./doctrine clear-cache --result
```

When you use the `--result` option you can use some other options to be more specific about what queries result sets you want to clear.

Just like the API of the cache drivers you can clear based on an ID, regular expression, prefix or suffix.

```
$ ./doctrine clear-cache --result --id=cache_id
```

Or if you want to clear based on a regular expressions.

```
$ ./doctrine clear-cache --result --regex=users_.*
```

Or with a prefix.

```
$ ./doctrine clear-cache --result --prefix=users_
```

And finally with a suffix.

```
$ ./doctrine clear-cache --result --suffix=_my_account
```

Nota: Using the `--id`, `--regex`, etc. options with the `--query` and `--metadata` are not allowed as it is not necessary to be specific about what you clear. You only ever need to completely clear the cache to remove stale entries.

1.22.5 Cache Slams

Something to be careful of when utilizing the cache drivers is cache slams. If you have a heavily trafficked website with some code that checks for the existence of a cache record and if it does not exist it generates the information and saves it to the cache. Now if 100 requests were issued all at the same time and each one sees the cache does not exist and they all try and insert the same cache entry it could lock up APC, Xcache, etc. and cause problems. Ways exist to work around this, like pre-populating your cache and not letting your users requests populate the cache.

You can read more about cache slams in this blog post.

1.23 Mejorando el rendimiento

1.23.1 Memorizando el código de bytes

Es muy recomendable usar una caché de código de bytes como APC. Una caché de código de bytes elimina la necesidad de analizar el código *PHP* en cada petición y puede mejorar el rendimiento.

> "Si te preocupa el rendimiento y no utilizas una caché de código de bytes, entonces, realmente no te preocupas por el rendimiento. Por favor, consigue una y empieza a usarla".
>
> - Stas Malyshev, Colaborador del núcleo de *PHP* y Empleado en Zend*

1.23.2 Metadatos y consulta de caché

Como ya mencionamos anteriormente en este capítulo sobre la configuración de *Doctrine*, no se recomienda utilizar *Doctrine* sin una caché de metadatos y consultas (de preferencia con APC o Memcache como el controlador de la caché). Al operar *Doctrine* sin esta caché, tendría que cargar su información de asignación en cada petición y tendría que analizar cada consulta `DQL` en cada petición. Esto es un desperdicio de recursos.

1.23.3 Formatos alternativos para resultados de consulta

Make effective use of the available alternative query result formats like nested array graphs or pure scalar results, especially in scenarios where data is loaded for read-only purposes.

1.23.4 Read-Only Entities

Starting with Doctrine 2.1 you can mark entities as read only (See metadata mapping references for details). This means that the entity marked as read only is never considered for updates, which means when you call flush on the EntityManager these entities are skipped even if properties changed. Read-Only allows to persist new entities of a kind and remove existing ones, they are just not considered for updates.

1.23.5 Extra-Lazy Collections

If entities hold references to large collections you will get performance and memory problems initializing them. To solve this issue you can use the EXTRA_LAZY fetch-mode feature for collections. See the *tutorial* for more information on how this fetch mode works.

1.23.6 Temporarily change fetch mode in DQL

See *Doctrine Query Language chapter*

1.23.7 Aplicando buenas prácticas

Muchos de los puntos mencionados en el capítulo de las buenas prácticas también afectan positivamente al rendimiento de *Doctrine*.

1.24 Herramientas

1.24.1 Consola de *Doctrine*

La consola de *Doctrine* es una herramienta de interfaz de línea de ordenes para simplificar las tareas más comunes durante el desarrollo de un proyecto que utiliza *Doctrine* 2.

Instalando

Si instalaste *Doctrine* 2 a través de PEAR, la orden `doctrine` de la herramienta de línea de ordenes ya debería estar disponible para ti.

If you use Doctrine through SVN or a release package you need to copy the `doctrine` and `doctrine.php` files from the `tools/sandbox` or `bin` folder, respectively, to a location of your choice, for example a `tools` folder of your project. You probably need to edit `doctrine.php` to adjust some paths to the new environment, most importantly the first line that includes the `Doctrine\Common\ClassLoader`.

Consiguiendo ayuda

Type `doctrine` on the command line and you should see an overview of the available commands or use the –help flag to get information on the available commands. If you want to know more about the use of generate entities for example, you can call:

```
doctrine orm:generate-entities --help
```

Configurando

Whenever the `doctrine` command line tool is invoked, it can access alls Commands that were registered by developer. There is no auto-detection mechanism at work. The `bin\doctrine.php` file already registers all the commands that currently ship with Doctrine DBAL and ORM. If you want to use additional commands you have to register them yourself.

All the commands of the Doctrine Console require either the `db` or the `em` helpers to be defined in order to work correctly. Doctrine Console requires the definition of a HelperSet that is the DI tool to be injected in the Console. In case of a project that is dealing exclusively with DBAL, the ConnectionHelper is required:

```
<?php
$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($conn)
));
$cli->setHelperSet($helperSet);
```

When dealing with the ORM package, the EntityManagerHelper is required:

```
<?php
$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em)
));
$cli->setHelperSet($helperSet);
```

The HelperSet instance has to be generated in a separate file (i.e. `cli-config.php`) that contains typical Doctrine bootstrap code and predefines the needed HelperSet attributes mentioned above. A typical `cli-config.php` file looks as follows:

```
<?php
require_once __DIR__ . '/../../lib/Doctrine/Common/ClassLoader.php';

$classLoader = new \Doctrine\Common\ClassLoader('Entities', __DIR__);
$classLoader->register();

$classLoader = new \Doctrine\Common\ClassLoader('Proxies', __DIR__);
$classLoader->register();

$config = new \Doctrine\ORM\Configuration();
$config->setMetadataCacheImpl(new \Doctrine\Common\Cache\ArrayCache);
$config->setProxyDir(__DIR__ . '/Proxies');
$config->setProxyNamespace('Proxies');

$connectionOptions = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite'
);

$em = \Doctrine\ORM\EntityManager::create($connectionOptions, $config);

$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($em->getConnection()),
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em)
));
```

It is important to define a correct HelperSet that doctrine.php script will ultimately use. The Doctrine Binary will automatically find the first instance of HelperSet in the global variable namespace and use this.

You can also add your own commands on-top of the Doctrine supported tools. To include a new command on Doctrine Console, you need to do:

```
<?php
$cli->addCommand(new \MiProyecto\Tools\Console\Commands\MyCustomCommand());
```

Additionally, include multiple commands (and overriding previously defined ones) is possible through the command:

```
<?php
$cli->addCommands(array(
    new \MiProyecto\Tools\Console\Commands\MyCustomCommand(),
    new \MiProyecto\Tools\Console\Commands\SomethingCommand(),
    new \MiProyecto\Tools\Console\Commands\AnotherCommand(),
    new \MiProyecto\Tools\Console\Commands\OneMoreCommand(),
));
```

Command Overview

The following Commands are currently available:

- `help` Displays help for a command (?)
- `list` Lists commands
- `dbal:import` Import SQL file(s) directly to Database.
- `dbal:run-sql` Executes arbitrary SQL directly from the command line.
- `orm:clear-cache:metadata` Clear all metadata cache of the various cache drivers.
- `orm:clear-cache:query` Clear all query cache of the various cache drivers.
- `orm:clear-cache:result` Clear result cache of the various cache drivers.
- `orm:convert-d1-schema` Converts Doctrine 1.X schema into a Doctrine 2.X schema.
- `orm:convert-mapping` Convert mapping information between supported formats.
- `orm:ensure-production-settings` Verify that Doctrine is properly configured for a production environment.
- `orm:generate-entities` Generate entity classes and method stubs from your mapping information.
- `orm:generate-proxies` Generates proxy classes for entity classes.
- `orm:generate-repositories` Generate repository classes from your mapping information.
- `orm:run-dql` Executes arbitrary DQL directly from the command line.
- `orm:schema-tool:create` Processes the schema and either create it directly on EntityManager Storage Connection or generate the SQL output.
- `orm:schema-tool:drop` Processes the schema and either drop the database schema of EntityManager Storage Connection or generate the SQL output.
- `orm:schema-tool:update` Processes the schema and either update the database schema of EntityManager Storage Connection or generate the SQL output.

1.24.2 Database Schema Generation

Nota: SchemaTool can do harm to your database. It will drop or alter tables, indexes, sequences and such. Please use this tool with caution in development and not on a production server. It is meant for helping you develop your Database Schema, but NOT with migrating schema from A to B in production. A safe approach would be generating

the SQL on development server and saving it into SQL Migration files that are executed manually on the production server.

SchemaTool assumes your Doctrine Project uses the given database on its own. Update and Drop commands will mess with other tables if they are not related to the current project that is using Doctrine. Please be careful!

To generate your database schema from your Doctrine mapping files you can use the `SchemaTool` class or the `schema-tool` Console Command.

When using the SchemaTool class directly, create your schema using the `createSchema()` method. First create an instance of the `SchemaTool` and pass it an instance of the `EntityManager` that you want to use to create the schema. This method receives an array of `ClassMetadataInfo` instances.

```
<?php
$tool = new \Doctrine\ORM\Tools\SchemaTool($em);
$classes = array(
  $em->getClassMetadata('Entities\User'),
  $em->getClassMetadata('Entities\Profile')
);
$tool->createSchema($classes);
```

To drop the schema you can use the `dropSchema()` method.

```
<?php
$tool->dropSchema($classes);
```

This drops all the tables that are currently used by your metadata model. When you are changing your metadata a lot during development you might want to drop the complete database instead of only the tables of the current model to clean up with orphaned tables.

```
<?php
$tool->dropSchema($classes, \Doctrine\ORM\Tools\SchemaTool::DROP_DATABASE);
```

You can also use database introspection to update your schema easily with the `updateSchema()` method. It will compare your existing database schema to the passed array of `ClassMetdataInfo` instances.

```
<?php
$tool->updateSchema($classes);
```

If you want to use this functionality from the command line you can use the `schema-tool` command.

To create the schema use the `create` command:

```
$ php doctrine orm:schema-tool:create
```

To drop the schema use the `drop` command:

```
$ php doctrine orm:schema-tool:drop
```

If you want to drop and then recreate the schema then use both options:

```
$ php doctrine orm:schema-tool:drop
$ php doctrine orm:schema-tool:create
```

As you would think, if you want to update your schema use the `update` command:

```
$ php doctrine orm:schema-tool:update
```

All of the above commands also accept a `--dump-sql` option that will output the SQL for the ran operation.

```
$ php doctrine orm:schema-tool:create --dump-sql
```

Before using the orm:schema-tool commands, remember to configure your cli-config.php properly.

Nota: When using the Annotation Mapping Driver you have to either setup your autoloader in the cli-config.php correctly to find all the entities, or you can use the second argument of the `EntityManagerHelper` to specify all the paths of your entities (or mapping files), i.e. `new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em, $mappingPaths);`

1.24.3 Entity Generation

Generate entity classes and method stubs from your mapping information.

```
$ php doctrine orm:generate-entities
$ php doctrine orm:generate-entities --update-entities
$ php doctrine orm:generate-entities --regenerate-entities
```

This command is not suited for constant usage. It is a little helper and does not support all the mapping edge cases very well. You still have to put work in your entities after using this command.

It is possible to use the EntityGenerator on code that you have already written. It will not be lost. The EntityGenerator will only append new code to your file and will not delete the old code. However this approach may still be prone to error and we suggest you use code repositories such as GIT or SVN to make backups of your code.

It makes sense to generate the entity code if you are using entities as Data Access Objects only and dont put much additional logic on them. If you are however putting much more logic on the entities you should refrain from using the entity-generator and code your entities manually.

Nota: Even if you specified Inheritance options in your XML or YAML Mapping files the generator cannot generate the base and child classes for you correctly, because it doesn't know which class is supposed to extend which. You have to adjust the entity code manually for inheritance to work!

1.24.4 Convert Mapping Information

Convert mapping information between supported formats.

This is an **execute one-time** command. It should not be necessary for you to call this method multiple times, escpecially when using the `--from-database` flag.

Converting an existing databsae schema into mapping files only solves about 70-80 % of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

Nota: There is no need to convert YAML or XML mapping files to annotations every time you make changes. All mapping drivers are first class citizens in Doctrine 2 and can be used as runtime mapping for the ORM. See the docs on XML and YAML Mapping for an example how to register this metadata drivers as primary mapping source.

To convert some mapping information between the various supported formats you can use the `ClassMetadataExporter` to get exporter instances for the different formats:

```
<?php
$cme = new \Doctrine\ORM\Tools\Export\ClassMetadataExporter();
```

Once you have a instance you can use it to get an exporter. For example, the yml exporter:

```
<?php
$exporter = $cme->getExporter('yml', '/path/to/export/yml');
```

Now you can export some `ClassMetadata` instances:

```
<?php
$classes = array(
  $em->getClassMetadata('Entities\User'),
  $em->getClassMetadata('Entities\Profile')
);
$exporter->setMetadata($classes);
$exporter->export();
```

This functionality is also available from the command line to convert your loaded mapping information to another format. The `orm:convert-mapping` command accepts two arguments, the type to convert to and the path to generate it:

```
$ php doctrine orm:convert-mapping xml /path/to/mapping-path-converted-to-xml
```

1.24.5 Ingeniería inversa

You can use the `DatabaseDriver` to reverse engineer a database to an array of `ClassMetadataInfo` instances and generate YAML, XML, etc. from them.

Nota: Reverse Engineering is a **one-time** process that can get you started with a project. Converting an existing database schema into mapping files only detects about 70-80 % of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

First you need to retrieve the metadata instances with the `DatabaseDriver`:

```
<?php
$em->getConfiguration()->setMetadataDriverImpl(
    new \Doctrine\ORM\Mapping\Driver\DatabaseDriver(
        $em->getConnection()->getSchemaManager()
    )
);

$cmf = new DisconnectedClassMetadataFactory();
$cmf->setEntityManager($em);
$metadata = $cmf->getAllMetadata();
```

Now you can get an exporter instance and export the loaded metadata to yml:

```
<?php
$exporter = $cme->getExporter('yml', '/ruta/a/export/yml');
$exporter->setMetadata($metadata);
$exporter->export();
```

You can also reverse engineer a database using the `orm:convert-mapping` command:

```
$ php doctrine orm:convert-mapping --from-database yml /ruta/a/mapping-path-converted-to-yml
```

Nota: Reverse Engineering is not always working perfectly depending on special cases. It will only detect Many-To-One relations (even if they are One-To-One) and will try to create entities from Many-To-Many tables. It also has

problems with naming of foreign keys that have multiple column names. Cualquier ingeniería inversa de esquemas de bases de datos necesita de considerable trabajo manual para convertirla en un modelo de dominio útil.

1.25 Metadata Drivers

The heart of an object relational mapper is the mapping information that glues everything together. It instructs the EntityManager how it should behave when dealing with the different entities.

1.25.1 Core Metadata Drivers

Doctrine provides a few different ways for you to specify your metadata:

- **XML files** (XmlDriver)
- **Class DocBlock Annotations** (AnnotationDriver)
- **YAML files** (YamlDriver)
- **PHP Code in files or static functions** (PhpDriver)

Something important to note about the above drivers is they are all an intermediate step to the same end result. The mapping information is populated to `Doctrine\ORM\Mapping\ClassMetadata` instances. So in the end, Doctrine only ever has to work with the API of the `ClassMetadata` class to get mapping information for an entity.

Nota: The populated `ClassMetadata` instances are also cached so in a production environment the parsing and populating only ever happens once. You can configure the metadata cache implementation using the `setMetadataCacheImpl()` method on the `Doctrine\ORM\Configuration` class:

```
<?php
$em->getConfiguration()->setMetadataCacheImpl(new ApcCache());
```

If you want to use one of the included core metadata drivers you just need to configure it. All the drivers are in the `Doctrine\ORM\Mapping\Driver` namespace:

```
<?php
$driver = new \Doctrine\ORM\Mapping\Driver\XmlDriver('/path/to/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

1.25.2 Implementing Metadata Drivers

In addition to the included metadata drivers you can very easily implement your own. All you need to do is define a class which implements the `Driver` interface:

```
<?php
namespace Doctrine\ORM\Mapping\Driver;

use Doctrine\ORM\Mapping\ClassMetadataInfo;

interface Driver
{
    /**
     * Loads the metadata for the specified class into the provided container.
     *
```

```
     * @param string $className
     * @param ClassMetadataInfo $metadata
     */
    function loadMetadataForClass($className, ClassMetadataInfo $metadata);

    /**
     * Gets the names of all mapped classes known to this driver.
     *
     * @return array The names of all mapped classes known to this driver.
     */
    function getAllClassNames();

    /**
     * Whether the class with the specified name should have its metadata loaded.
     * This is only the case if it is either mapped as an Entity or a
     * MappedSuperclass.
     *
     * @param string $className
     * @return boolean
     */
    function isTransient($className);
}
```

If you want to write a metadata driver to parse information from some file format we've made your life a little easier by providing the `AbstractFileDriver` implementation for you to extend from:

```
<?php
class MyMetadataDriver extends AbstractFileDriver
{
    /**
     * {@inheritdoc}
     */
    protected $_fileExtension = '.dcm.ext';

    /**
     * {@inheritdoc}
     */
    public function loadMetadataForClass($className, ClassMetadataInfo $metadata)
    {
        $data = $this->_loadMappingFile($file);

        // populate ClassMetadataInfo instance from $data
    }

    /**
     * {@inheritdoc}
     */
    protected function _loadMappingFile($file)
    {
        // parse contents of $file and return php data structure
    }
}
```

Nota: When using the `AbstractFileDriver` it requires that you only have one entity defined per file and the file named after the class described inside where namespace separators are replaced by periods. So if you have an entity named `Entities\User` and you wanted to write a mapping file for your driver above you would need to name the file `Entities.User.dcm.ext` for it to be recognized.

Now you can use your `MyMetadataDriver` implementation by setting it with the `setMetadataDriverImpl()` method:

```
<?php
$driver = new MyMetadataDriver('/path/to/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

1.25.3 ClassMetadata

The last piece you need to know and understand about metadata in Doctrine 2 is the API of the `ClassMetadata` classes. You need to be familiar with them in order to implement your own drivers but more importantly to retrieve mapping information for a certain entity when needed.

You have all the methods you need to manually specify the mapping information instead of using some mapping file to populate it from. The base `ClassMetadataInfo` class is responsible for only data storage and is not meant for runtime use. It does not require that the class actually exists yet so it is useful for describing some entity before it exists and using that information to generate for example the entities themselves. The class `ClassMetadata` extends `ClassMetadataInfo` and adds some functionality required for runtime usage and requires that the PHP class is present and can be autoloaded.

You can read more about the API of the `ClassMetadata` classes in the PHP Mapping chapter.

1.25.4 Getting ClassMetadata Instances

If you want to get the `ClassMetadata` instance for an entity in your project to programatically use some mapping information to generate some HTML or something similar you can retrieve it through the `ClassMetadataFactory`:

```
<?php
$cmf = $em->getMetadataFactory();
$class = $cmf->getMetadataFor('MyEntityName');
```

Now you can learn about the entity and use the data stored in the `ClassMetadata` instance to get all mapped fields for example and iterate over them:

```
<?php
foreach ($class->fieldMappings as $fieldMapping) {
    echo $fieldMapping['fieldName'] . "\n";
}
```

1.26 Buenas prácticas

Las buenas prácticas mencionadas aquí que generalmente afectan al diseño de bases de datos se refieren a las buenas prácticas cuando se trabaja con *Doctrine* y no necesariamente reflejan las mejores prácticas para el diseño de bases de datos en general.

1.26.1 No utilices propiedades públicas en las entidades

Es muy importante que no asignes propiedades públicas en las entidades, sino únicamente protegidas o privadas. La razón de esto es simple, cada vez que accedes a una propiedad pública de un objeto delegado que no se ha iniciado aún el valor devuelto será nulo. *Doctrine* no se puede conectar a este proceso y mágicamente cargar la entidad de manera diferida.

Esto puede crear situaciones en las que es muy difícil depurar el error actual. Por lo tanto, te instamos a asignar sólo propiedades privadas y protegidas en las entidades y usar métodos captadores o `\_\_get()` mágicos para acceder a ellos.

1.26.2 Limita las relaciones tanto como sea posible

Es importante limitar las relaciones tanto como sea posible. Esto significa que:

- Impose a traversal direction (avoid bidirectional associations if possible)
- Eliminate nonessential associations

This has several benefits:

- Reduced coupling in your domain model
- Simpler code in your domain model (no need to maintain bidirectionality properly)
- Less work for Doctrine

1.26.3 Avoid composite keys

Even though Doctrine fully supports composite keys it is best not to use them if possible. Composite keys require additional work by Doctrine and thus have a higher probability of errors.

1.26.4 Use events judiciously

The event system of Doctrine is great and fast. Even though making heavy use of events, especially lifecycle events, can have a negative impact on the performance of your application. Thus you should use events judiciously.

1.26.5 Use cascades judiciously

Automatic cascades of the persist/remove/merge/etc. operations are very handy but should be used wisely. Do NOT simply add all cascades to all associations. Think about which cascades actually do make sense for you for a particular association, given the scenarios it is most likely used in.

1.26.6 Don't use special characters

Avoid using any non-ASCII characters in class, field, table or column names. Doctrine itself is not unicode-safe in many places and will not be until PHP itself is fully unicode-aware (PHP6).

1.26.7 Don't use identifier quoting

Identifier quoting is a workaround for using reserved words that often causes problems in edge cases. Do not use identifier quoting and avoid using reserved words as table or column names.

1.26.8 Initialize collections in the constructor

It is recommended best practice to initialize any business collections in entities in the constructor. Ejemplo:

```
<?php
namespace MiProyecto\Model;
use Doctrine\Common\Collections\ArrayCollection;

class User {
    private $addresses;
    private $articles;

    public function __construct() {
        $this->addresses = new ArrayCollection;
        $this->articles = new ArrayCollection;
    }
}
```

1.26.9 No asignes claves externas a los campos en una entidad

Foreign keys have no meaning whatsoever in an object model. Foreign keys are how a relational database establishes relationships. Your object model establishes relationships through object references. Thus mapping foreign keys to object fields heavily leaks details of the relational model into the object model, something you really should not do.

1.26.10 Use explicit transaction demarcation

While Doctrine will automatically wrap all DML operations in a transaction on flush(), it is considered best practice to explicitly set the transaction boundaries yourself. Otherwise every single query is wrapped in a small transaction (Yes, SELECT queries, too) since you can not talk to your database outside of a transaction. While such short transactions for read-only (SELECT) queries generally don't have any noticeable performance impact, it is still preferable to use fewer, well-defined transactions that are established through explicit transaction boundaries.

1.27 Limitations and Known Issues

We try to make using Doctrine2 a very pleasant experience. Therefore we think it is very important to be honest about the current limitations to our users. Much like every other piece of software Doctrine2 is not perfect and far from feature complete. This section should give you an overview of current limitations of Doctrine 2 as well as critical known issues that you should know about.

1.27.1 Current Limitations

There is a set of limitations that exist currently which might be solved in the future. Any of this limitations now stated has at least one ticket in the Tracker and is discussed for future releases.

Join-Columns with non-primary keys

It is not possible to use join columns pointing to non-primary keys. Doctrine will think these are the primary keys and create lazy-loading proxies with the data, which can lead to unexpected results. Doctrine can for performance reasons not validate the correctness of this settings at runtime but only through the Validate Schema command.

Mapping Arrays to a Join Table

Related to the previous limitation with "Foreign Keys as Identifier" you might be interested in mapping the same table structure as given above to an array. However this is not yet possible either. Ve el siguiente ejemplo:

```
CREATE TABLE product (
    id INTEGER,
    name VARCHAR,
    PRIMARY KEY(id)
);

CREATE TABLE product_attributes (
    product_id INTEGER,
    attribute_name VARCHAR,
    attribute_value VARCHAR,
    PRIMARY KEY (product_id, attribute_name)
);
```

This schema should be mapped to a Product Entity as follows:

```
class Product
{
    private $id;
    private $name;
    private $attributes = array();
}
```

Where the `attribute_name` column contains the key and `attribute_value` contains the value of each array element in `$attributes`.

The feature request for persistence of primitive value arrays is described in the DDC-298 ticket.

Value Objects

There is currently no native support value objects in Doctrine other than for `DateTime` instances or if you serialize the objects using `serialize()/deserialize()` which the DBAL Type "object" supports.

The feature request for full value-object support is described in the DDC-93 ticket.

Applying Filter Rules to any Query

There are scenarios in many applications where you want to apply additional filter rules to each query implicitly. Examples include:

- En aplicaciones I18N restringen los resultados a las entidades anotadas con un lugar específico
- Para una gran colección siempre devuelven sólo los objetos en un rango de fechas/donde la condición fue aplicada.
- Soft-Delete

There is currently no way to achieve this consistently across both DQL and Repository/Persister generated queries, but as this is a pretty important feature we plan to add support for it in the future.

Restricing Associations

There is currently no way to restrict associations to a subset of entities matching a given condition. You should use a DQL query to retrieve a subset of associated entities. For example if you need all visible articles in a certain category

you could have the following code in an entity repository:

```
<?php
class ArticleRepository extends EntityRepository
{
    public function getVisibleByCategory(Category $category)
    {
        $dql = "SELECT a FROM Article a WHERE a.category = ?1 and a.visible = true";
        return $this->getEntityManager()
                    ->createQuery($dql)
                    ->setParameter(1, $category)
                    ->getResult();
    }
}
```

Cascade Merge with Bi-directional Associations

There are two bugs now that concern the use of cascade merge in combination with bi-directional associations. Make sure to study the behavior of cascade merge if you are using it:

- DDC-875 Merge can sometimes add the same entity twice into a collection
- DDC-763 Cascade merge on associated entities can insert too many rows through "Persistence by Reachability"

Custom Persisters

A Persister in Doctrine is an object that is responsible for the hydration and write operations of an entity against the database. Currently there is no way to overwrite the persister implementation for a given entity, however there are several use-cases that can benefit from custom persister implementations:

- Add Upsert Support
- Evaluate possible ways in which stored-procedures can be used
- The previous Filter Rules Feature Request

Persist Keys of Collections

PHP Arrays are ordered hash-maps and so should be the `Doctrine\Common\Collections\Collection` interface. We plan to evaluate a feature that optionally persists and hydrates the keys of a Collection instance.

Ticket DDC-213

Mapping many tables to one entity

It is not possible to map several equally looking tables onto one entity. For example if you have a production and an archive table of a certain business concept then you cannot have both tables map to the same entity.

Behaviors

Doctrine 2 *will never* include a behavior system like Doctrine 1 in the core library. We don't think behaviors add more value than they cost pain and debugging hell. Please see the many different blog posts we have written on this topics:

- Doctrine2 "Behaviors" in a Nutshell
- A re-usable Versionable behavior for Doctrine2

- Write your own ORM on top of Doctrine2
- Doctrine 2 Behavioral Extensions
- *Doctrator <https://github.com/pablodip/doctrator>_*

Doctrine 2 has enough hooks and extension points so that *you* can add whatever you want on top of it. None of this will ever become core functionality of Doctrine2 however, you will have to rely on third party extensions for magical behaviors.

Nested Set

NestedSet was offered as a behavior in Doctrine 1 and will not be included in the core of Doctrine 2. However there are already two extensions out there that offer support for Nested Set with Doctrine 2:

- Doctrine2 Hierachical-Structural Behavior
- Doctrine2 NestedSet

1.27.2 Known Issues

The Known Issues section describes critical/blocker bugs and other issues that are either complicated to fix, not fixable due to backwards compatibility issues or where no simple fix exists (yet). We don't plan to add every bug in the tracker there, just those issues that can potentially cause nightmares or pain of any sort.

Identifier Quoting and Legacy Databases

For compatibility reasons between all the supported vendors and edge case problems Doctrine 2 does *NOT* do automatic identifier quoting. This can lead to problems when trying to get legacy-databases to work with Doctrine 2.

- You can quote column-names as described in the *Basic-Mapping* section.
- You cannot quote join column names.
- You cannot use non [a-zA-Z0-9_]+ characters, they will break several SQL statements.

Having problems with these kind of column names? Many databases support all CRUD operations on views that semantically map to certain tables. You can create views for all your problematic tables and column names to avoid the legacy quoting nightmare.

Microsoft SQL Server and Doctrine "datetime"

Doctrine assumes that you use DateTime2 data-types. If your legacy database contains DateTime datatypes then you have to add your own data-type (see Basic Mapping for an example).

Guías iniciales

2.1 Primeros pasos

Doctrine 2 es un asignador objetorelacional (ORM) para *PHP* 5.3.0+ que proporciona persistencia transparente de objetos PHP. Utiliza el patrón asignador de datos en el corazón de este proyecto, a fin de conseguir una completa separación del dominio/lógica del negocio de la persistencia de un sistema de bases de datos relacional. El beneficio de *Doctrine* para el programador es la capacidad de centrarse exclusivamente en la lógica del negocio orientado a objetos y preocuparse de la persistencia sólo como una tarea secundaria. Esto no significa que la persistencia no sea importante para *Doctrine* 2, no obstante, creemos que hay beneficios considerables para la programación orientada a objetos, si la persistencia y las entidades se mantienen perfectamente separadas.

2.1.1 ¿Qué son las entidades?

Las entidades son objetos PHP ligeros que no necesitan extender ninguna clase abstracta base o interfaz. Una clase entidad no debe ser final o contener métodos final. Además, no debe implementar **clone** ni **wakeup** o *hazlo con seguridad*.

Consulta el *capítulo arquitectura* para ver una lista completa de las restricciones que tus entidades deben cumplir.

Una entidad contiene propiedades persistibles. Una propiedad persistible es una variable de la instancia de Entidad que se guarda y recupera de la base de datos por medio de las capacidades de asignación de datos de *Doctrine*.

2.1.2 Un modelo de ejemplo: Rastreador de fallos

Para esta guía de introducción a *Doctrine* vamos a implementar el modelo del dominio rastreador de fallos de documentación de Zend_Db_Table. Leyendo la documentación podemos extraer los requisitos que son:

- Un fallo tiene una descripción, fecha de creación, estado, informante e ingeniero
- Un fallo puede ocurrir en diferentes productos (sistemas operativos)
- Los productos tienen un nombre.
- Ambos informantes de fallos e ingenieros son Usuarios del Sistema.
- Un usuario puede crear nuevos fallos.
- El ingeniero asignado puede cerrar un fallo.

- Un usuario puede ver todos sus fallos reportados o asignados.
- Los fallos se pueden paginar a través de una vista de lista.

Advertencia: Esta guía va construyendo gradualmente tu conocimiento de *Doctrine* 2 e incluso te permite cometer algunos errores, para mostrar algunos errores comunes en la asignación de entidades a una base de datos. No copies y pegues ciegamente estos ejemplos, no están listos para producción sin comentarios adicionales y el conocimiento que esta guía enseña.

2.1.3 Un primer prototipo

Un primer diseño simplificado para este modelo del dominio podría tener el siguiente conjunto de clases:

```
<?php
class Bug
{
    public $id;
    public $description;
    public $created;
    public $status;
    public $products = array();
    public $reporter;
    public $engineer;
}
class Product
{
    public $id;
    public $nombre;
}
class User
{
    public $id;
    public $nombre;
    public $reportedBugs = array();
    public $assignedBugs = array();
}
```

Advertencia: Este es sólo un prototipo, por favor no utilices propiedades públicas en todo *Doctrine* 2, la sección "Consultas para casos de uso de la aplicación" muestra por qué. En combinación con los delegados de las propiedades públicas puedes compensar fallos bastante desagradables.

Debido a que nos vamos a centrar en el aspecto de la asignación, no estamos tratando de encapsular la lógica del negocio en este ejemplo. Todas las propiedades persistibles tienen visibilidad pública. Pronto veremos que esta no es la mejor solución en combinación con *Doctrine* 2, una restricción que en realidad te obliga a encapsular tus propiedades. En realidad *Doctrine* 2 para la persistencia utiliza Reflexión para acceder a los valores de todas las propiedades de tus entidades.

Muchos de los campos son simples valores escalares, por ejemplo, los tres campos ID de las entidades, sus nombres, descripción, estado y fechas de cambio. *Doctrine* 2 puede manejar fácilmente estos valores individuales al igual que cualquier otro ORM. Desde el punto de vista de nuestro modelo del dominio están listos para utilizarse en este momento y, en una etapa posterior, vamos a ver la forma en que se asignan a la base de datos.

También hay varias referencias entre objetos en este modelo del dominio, cuya semántica se analiza caso por caso en este punto para explicar cómo los maneja *Doctrine*. En general, cada relación unoAuno o muchosAuno en la base de datos se sustituye por una instancia del objeto relacionado en el modelo del dominio. Cada relación unoAmuchos o muchosAmuchos se sustituye por una colección de instancias en el modelo del dominio.

Si piensas detenidamente en esto, te darás cuenta de que *Doctrine* 2 debería cargar en memoria la base de datos completa si accedes a un objeto. Sin embargo, de manera predeterminada *Doctrine* genera delegados diferidos para cargar las entidades o colecciones de todas las relaciones que todavía no se han recuperado explícitamente de la base de datos.

To be able to use lazyload with collections, simple PHP arrays have to be replaced by a generic collection interface for Doctrine which tries to act as as much like an array as possible by using ArrayAccess, IteratorAggregate and Countable interfaces. La clase es la implementación más simple de esta interfaz.

Ahora que sabemos esto, tenemos que aclarar nuestro modelo del dominio para hacer frente a los supuestos acerca de las colecciones relacionadas:

```
<?php
use Doctrine\Common\Collections\ArrayCollection;

class Bug
{
    public $products = null;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
}

class User
{
    public $reportedBugs = null;
    public $assignedBugs = null;

    public function __construct()
    {
        $this->reportedBugs = new ArrayCollection();
        $this->assignedBugs = new ArrayCollection();
    }
}
```

Cada vez que se recrea una entidad desde la base de datos, una implementación de tipo colección de *Doctrine* se inyecta en la entidad en lugar de una matriz. En comparación con el ArrayCollection esta implementación ayuda a que el ORM de *Doctrine* entienda los cambios que han sucedido a la colección que se destaca por la persistencia.

Advertencia: La carga diferida de delegados siempre contiene una instancia del `EntityManager` de *Doctrine* y todas sus dependencias. Por lo tanto, un `var\_dump()` posiblemente volcará una estructura recursiva muy grande de la cual es imposible de reproducir y leer. Tienes que usar `Doctrine\Common\Util\Debug::dump()` para restringir el volcado a un nivel humanamente legible. Además debes tener en cuenta que el volcado del `EntityManager` a un navegador puede tardar varios minutos, y el método `Debug::dump()` sólo omite cualquier aparición de esta en instancias delegadas.

Debido a que sólo trabajamos con colecciones para las referencias debemos tener cuidado de implementar una referencia bidireccional en el modelo del dominio. El concepto de propiedad o lado inverso de una relación es fundamental para esta noción y siempre se debe tener en cuenta. Se hacen los siguientes supuestos sobre las relaciones y se deben seguir para poder trabajar con *Doctrine* 2. Estos supuestos no son exclusivos de *Doctrine* 2, pero son las buenas prácticas en el manejo de las relaciones de bases de datos y asignador objetorelacional.

- Los cambios a las Colecciones se guardan o actualizan, cuando la entidad en el lado del *dueño* de la colección se guarda o actualiza.
- Guardar una Entidad en el lado inverso de una relación nunca provoca una operación de persistencia para cam-

biar la colección.

- En una relación uno-a-uno la entidad que tiene en su propia tabla la clave externa de la entidad relacionada en la base de datos *siempre* es la parte propietaria de la relación.
- En una relación muchos a muchos, las dos partes pueden ser la parte propietaria de la relación. Sin embargo, en una relación bidireccional de muchos a muchos sólo se permite sea uno.
- En una relación de muchos a uno, el lado muchos es el lado propietario predeterminado, ya que posee la clave externa.
- El lado UnoAmuchos de una relación es inverso por omisión, ya que la clave externa se guarda en el lado de muchos. Una relación UnoAMuchos sólo puede ser el lado propietario, si se implementa usando una relación MuchosAMuchos uniendo tablas y restringiendo el lado uno para permitir que sólo los valores únicos restrinjan la base de datos.

Nota: La consistencia de las referencias bidireccionales en el lado inverso de la relación, el código de la aplicación las tiene que gestionar en modo usuario. *Doctrine* no puede actualizar mágicamente sus colecciones para ser consistente.

En el caso de los usuarios y fallos en que tenemos referencias de ida y vuelta a los fallos asignados e informados por un usuario, haciendo de esta una relación bidireccional. Tenemos que cambiar el código para garantizar la coherencia de la referencia bidireccional:

```
<?php
class Bug
{
    private $engineer;
    private $reporter;

    public function setEngineer($engineer)
    {
        $engineer->assignedToBug($this);
        $this->engineer = $engineer;
    }

    public function setReporter($reporter)
    {
        $reporter->addReportedBug($this);
        $this->reporter = $reporter;
    }

    public function getEngineer()
    {
        return $this->engineer;
    }

    public function getReporter()
    {
        return $this->reporter;
    }
}
class User
{
    private $reportedBugs = null;
    private $assignedBugs = null;

    public function addReportedBug($bug)
    {
```

```
        $this->reportedBugs[] = $bug;
    }

    public function assignedToBug($bug)
    {
        $this->assignedBugs[] = $bug;
    }
}
```

Elegí nombrar los métodos inversos en tiempo pasado, lo cual debe indicar que la asignación actual ya ha tenido lugar y los métodos se utilizan únicamente para garantizar la coherencia de las referencias. Puedes ver desde `User::addReportedBug()` y `User::assignedToBug()` que usar este método en el espacio de usuario por sí solo no agrega el Fallo a la colección de la parte propietaria de `Bug::$reporter` o `Bug::$engineer`. Usando estos métodos e invocando a *Doctrine* para que los persistiera no actualizaba la representación de las colecciones en la base de datos.

Sólo usando `Bug::setEngineer()` o `Bug::setReporter()` guarda correctamente la información de la relación. También fijamos como protegidas ambas variables de instancias de colección, sin embargo con las nuevas características de *PHP* 5.3 *Doctrine* aún es capaz de utilizar la reflexión para establecer y obtener valores de las propiedades protegidas y privadas.

Las propiedades `Bug::$reporter` y `Bug::$engineer` son relaciones muchos a uno, que apuntan a un usuario. En un modelo relacional normalizado la clave externa se guarda en la tabla de Fallos, por lo tanto, en nuestro modelo objeto→relación el Fallo está en la parte propietaria de la relación. Siempre debes asegurarte de que los casos de uso de tu modelo del dominio deben conducir a qué parte corresponde la inversión o poseer una en tu asignación a *Doctrine*. En nuestro ejemplo, cada vez que se guarda un nuevo Fallo o se asigna un fallo al ingeniero, no deseamos actualizar el usuario para conservar la referencia, sino el Fallo. Este es el caso con los Fallos en la parte propietaria de la relación.

La referencia producida por una relación unidireccional muchos a muchos en la base de datos que apunta desde Fallos a Productos.

```
<?php
class Bug
{
    private $products = null;

    public function assignToProduct($product)
    {
        $this->products[] = $product;
    }

    public function getProducts()
    {
        return $this->products;
    }
}
```

Ahora hemos terminado el modelo del dominio con los requisitos proporcionados. A partir del modelo simple con propiedades públicas sólo tuvimos que hacer un poco de trabajo para llegar a un modelo que encapsula las referencias entre los objetos para asegurarnos de que no se rompa su estado consistente al utilizar *Doctrine*.

Sin embargo, hasta ahora los supuestos de *Doctrine* impuestos sobre los objetos de nuestro negocio no nos restringen mucho en nuestra capacidad de modelar el dominio. En realidad, tendríamos que encapsular el acceso a todos los modos usando las buenas prácticas de las propiedades orientadas a objetos.

2.1.4 Asignando metadatos a nuestras `Entidades`

Hasta ahora sólo hemos implementado nuestras `Entidades` como estructuras de datos sin decirle a *Doctrine* cómo se persisten en la base de datos. Si existiera la perfecta base de datos en memoria, ahora podríamos terminar la aplicación usando estas entidades para implementar el código que cumpla con todos los requisitos. Sin embargo, el mundo no es perfecto y tenemos que persistir nuestras entidades en algún almacenamiento para asegurarnos de que no pierden su estado. Actualmente *Doctrine* sirve como un sistema de gestión de bases de datos. En el futuro estamos pensando en apoyar proveedores distintos a SQL como CouchDB o MongoDB, sin embargo esto todavía está lejos en el futuro.

El siguiente paso para la persistencia con *Doctrine* es describir la estructura de nuestras entidades del modelo de dominio a *Doctrine* usando un lenguaje de metadatos. El lenguaje de metadatos describe cómo se deben persistir las entidades, sus propiedades y referencias, y qué restricciones se deben aplicar a las mismas.

Los metadatos de las entidades se cargan utilizando una implementación de `Doctrine\ORM\Mapping\Driver\Driver` y *Doctrine* 2 ya viene con XML, YAML y controladores de Anotaciones. Esta guía deberá mostrarte las asignaciones de todos los controladores de asignación. Las referencias en el texto se harán para la asignación XML.

Puesto que no hemos encapsulado nuestras tres entidades en un espacio de nombres, tenemos que implementar tres archivos de asignación llamados Bug.dcm.xml, Product.dcm.xml y User.dcm.xml (o .yml) y ponerlos en un directorio distinto para asignar configuraciones. Para el controlador de anotaciones tenemos que usar comentarios `doc-block` sobre las clases entidad en sí mismas.

La primera definición descrita será Producto, ya que es la más simple:

- *PHP*

```php
<?php
/**
 * @Entity @Table(name="products")
 */
class Product
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;
    /** @Column(type="string") */
    public $nombre;
}
```

- *XML*

```xml
<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="Product" table="products">
          <id name="id" type="integer" column="product_id">
              <generator strategy="AUTO" />
          </id>

          <field name="name" column="product_name" type="string" />
      </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Product:
  type: entity
  table: products
```

```
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
```

La definición de la etiqueta de nivel superior `entidad` especifica información sobre la clase y el nombre de tabla. El tipo primitivo `Product::$nombre` se define como atributos de `campo`. La propiedad Id se define con la etiqueta `id`. El identificador tiene una etiqueta `generador` anidada la cual define que el mecanismo de generación de claves principal automáticamente utiliza la estrategia nativa de generación de Identificación de las plataformas de base de datos, por ejemplo AUTO INCREMENTO en el caso de MySQL o Secuencias en el caso de PostgreSQL y Oracle.

Entonces vamos a especificar la definición de un Fallo:

- *PHP*

```
<?php
/**
 * @Entity @Table(name="bugs")
 */
class Bug
{
    /**
     * @Id @Column(type="integer") @GeneratedValue
     */
    public $id;
    /**
     * @Column(type="string")
     */
    public $description;
    /**
     * @Column(type="datetime")
     */
    public $created;
    /**
     * @Column(type="string")
     */
    public $status;

    /**
     * @ManyToOne(targetEntity="User", inversedBy="assignedBugs")
     */
    private $engineer;

    /**
     * @ManyToOne(targetEntity="User", inversedBy="reportedBugs")
     */
    private $reporter;

    /**
     * @ManyToMany(targetEntity="Product")
     */
    private $products;
}
```

- *XML*

```
<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="Bug" table="bugs">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>

        <field name="description" type="text" />
        <field name="created" type="datetime" />
        <field name="status" type="string" />

        <many-to-one target-entity="User" field="reporter" inversed-by="reportedBugs" />
        <many-to-one target-entity="User" field="engineer" inversed-by="assignedBugs" />

        <many-to-many target-entity="Product" field="products" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
Bug:
  type: entity
  table: bugs
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    description:
      type: text
    created:
      type: datetime
    status:
      type: string
  manyToOne:
    reporter:
      targetEntity: User
      inversedBy: reportedBugs
    engineer:
      targetEntity: User
      inversedBy: assignedBugs
  manyToMany:
    products:
      targetEntity: Product
```

Una vez más tenemos la entidad, id y definiciones de tipo primitivos. Los nombres de columna se utilizan desde los ejemplos de Zend_Db_Table y tienen nombres diferentes a las propiedades de la clase Fallo. Además para la "creación" del campo, especifica que es del tipo "DATETIME", que se traduce al formato YYYY-mm-dd HH:mm:ss de la base de datos en una instancia DateTime de *PHP* y de vuelta.

Después de las definiciones del campo, se definen las dos referencias calificadas a la entidad Usuario. Ellas son creadas por la etiqueta `muchos-a-uno`. El nombre de clase de la entidad relacionada se tiene que especificar con atributo `entidad-atributo`, el cual es información suficiente para que el asignador de base de datos acceda a la tabla externa. Puesto que `informante` e `ingeniero` están en la parte propietaria de una relación bidireccional también tienes que especificar el atributo `invertido-por`. Ellos tienen que apuntar a los nombres de campo en el lado

inverso de la relación. En el siguiente ejemplo vamos a ver que el atributo `invertido-por` tiene una contraparte `asignado-por` la cual hace de este el lado inverso.

La última propiedad que falta es la colección `Bug::$products`. Esta tiene todos los productos donde el fallo específico está ocurriendo. Una vez más hay que definir los atributos `target-entity` y `field` en la etiqueta `many-to-many`. Además tienes que especificar los detalles de la unión de tablas muchos a muchos y sus columnas clave externas. La definición es bastante compleja, sin embargo, confía en el autocompletado XML el cual trabaja fácilmente, aunque no recuerdes los detalles del esquema todo el tiempo.

La última definición faltante es la de la entidad Usuario:

- *PHP*

```php
<?php
/**
 * @Entity @Table(name="users")
 */
class User
{
    /**
     * @Id @GeneratedValue @Column(type="integer")
     * @var string
     */
    public $id;

    /**
     * @Column(type="string")
     * @var string
     */
    public $nombre;

    /**
     * @OneToMany(targetEntity="Bug", mappedBy="reporter")
     * @var Bug[]
     */
    protected $reportedBugs = null;

    /**
     * @OneToMany(targetEntity="Bug", mappedBy="engineer")
     * @var Bug[]
     */
    protected $assignedBugs = null;
```

- *XML*

```xml
<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="User" name="users">
         <id name="id" type="integer">
             <generator strategy="AUTO" />
         </id>

         <field name="name" type="string" />

         <one-to-many target-entity="Bug" field="reportedBugs" mapped-by="reporter" />
         <one-to-many target-entity="Bug" field="assignedBugs" mapped-by="engineer" />
     </entity>
```

```
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  table: users
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
  oneToMany:
    reportedBugs:
      targetEntity: Bug
      mappedBy: reporter
    assignedBugs:
      targetEntity: Bug
      mappedBy: engineer
```

Aquí hay algunas cosas nuevas a mencionar sobre las etiquetas `uno-a-muchos`. Recuerda qué hemos explicado sobre el lado inverso y propietario. Ahora, tanto reportedBugs y assignedBugs son relaciones inversas, es decir, unen los detalles que ya se han definido en la parte propietaria. Por lo tanto, sólo tienes que especificar la propiedad en la clase Fallo que mantiene la parte propietaria.

En este ejemplo tienes una visión justa de las características más básicas del lenguaje de definición de metadatos.

2.1.5 Obteniendo el `EntityManager`

La interfaz pública de *Doctrine* es el `EntityManager`, el cual proporciona el punto de acceso a la gestión del ciclo de vida completo de tus entidades y transforma las entidades de ida y vuelta a la persistencia. Lo debes crear y configurar para usar tus entidades con *Doctrine* 2. Voy a mostrarte los pasos de configuración y luego lo diseccionaremos paso a paso:

```
<?php
// Configura el autocargador (1)
// consulta :doc:`Configuración <../reference/configuration>` para detalles actualizados del autocarg

$config = new Doctrine\ORM\Configuration(); // (2)

// Configura el delegado (3)
$config->setProxyDir(__DIR__.'/lib/MiProyecto/Proxies');
$config->setProxyNamespace('MiProyecto\Proxies');
$config->setAutoGenerateProxyClasses((APPLICATION_ENV == "development"));

// Configura la asignación (4)
$driverImpl = new Doctrine\ORM\Mapping\Driver\XmlDriver(__DIR__."/config/mappings/xml");
//$driverImpl = new Doctrine\ORM\Mapping\Driver\XmlDriver(__DIR__."/config/mappings/yml");
//$driverImpl = $config->newDefaultAnnotationDriver(__DIR__."/entities");
$config->setMetadataDriverImpl($driverImpl);

// Configura la memorización en caché (5)
if (APPLICATION_ENV == "development") {
    $cache = new \Doctrine\Common\Cache\ArrayCache();
```

```
} else {
    $cache = new \Doctrine\Common\Cache\ApcCache();
}
$config->setMetadataCacheImpl($cache);
$config->setQueryCacheImpl($cache);

// parámetros de configuración de la base de datos (6)
$conn = array(
    'driver' => 'pdo_sqlite',
    'path' => __DIR__ . '/db.sqlite',
);

// obtiene el gestor de entidades (7)
$evm = new Doctrine\Common\EventManager()
$entityManager = \Doctrine\ORM\EntityManager::create($conn, $config, $evm);
```

El primer bloque configura la capacidad de carga automática de *Doctrine*. Registramos el espacio de nombres *Doctrine* a la ruta especificada. Para añadir tu propio espacio de nombres puedes crear una instancia de otro `Cargador de clases` con diferentes argumentos para el espacio de nombres y la ruta. No hay ningún requisito para utilizar el `cargador de clases` de *Doctrine* para tus necesidades de carga automática, puedes utilizar el que más te convenga.

El segundo bloque contiene la instancia del objeto de configuración del ORM. Además de la configuración que muestran los siguientes bloques hay muchos otros, todos explicados en la *sección de configuración del manual*.

La configuración del delegado es un bloque necesario para tu aplicación, tienes que especificar el lugar donde *Doctrine* escribirá el código *PHP* para generar el delegado. Los delegados son hijos de las entidades generadas por *Doctrine* para permitir la seguridad de tipos en la carga diferida. En un capítulo posterior, veremos exactamente cómo funciona esto. Además de la ruta a los delegados también especificamos bajo qué espacio de nombres deben residir así como un indicador `autoGenerateProxyClasses` indicando cuales delegados se deberían regenerar en cada petición, lo cual se recomienda para cuando estás desarrollando. En producción esto se debe evitar a toda costa, la generación de la clase delegada puede ser bastante costosa.

El cuarto bloque contiene los detalles del controlador de asignación. Vamos a utilizar la asignación XML en este ejemplo, por lo tanto configura la instancia `XmlDriver` con una ruta al directorio de configuración de asignaciones, dónde hemos colocado el `Bug.dcm.xml`, `Product.dcm.xml` y `User.dcm.xml`.

El 5° bloque establece la configuración del almacenamiento en caché. En producción utilizamos el almacenamiento en caché sólo basándonos en la petición usando el `ArrayCache`. En producción literalmente es obligatorio utilizar Apc, Memcache o XCache para conseguir la máxima velocidad de *Doctrine*. Internamente *Doctrine* utiliza el almacenamiento en caché en gran medida para los metadatos y el lenguaje de consulta `DQL` así que asegúrate de usar un mecanismo de almacenamiento en caché.

El 6° bloque muestra las opciones de configuración necesarias para conectarse a una base de datos, en mi caso una base de datos basada en archivos SQLite. Todas las opciones de configuración para todos los controladores incluidos se presentan en la sección de configuración DBAL del manual.

El último bloque muestra cómo obtener el `EntityManager` a partir de un método fábrica, aquí también pasamos una instancia de `EventManager` la cual es opcional. However using the EventManager you can hook in to the lifecycle of entities, which is a common use-case, so you know how to configure it already.

2.1.6 Generando el esquema de base de datos

Ahora que hemos definido la asignación de metadatos y arrancamos el `EntityManager` queremos generar el esquema de la base de datos relacional desde ahí. *Doctrine* tiene una interfaz de línea de ordenes que te permite acceder al `SchemaTool`, un componente que genera las tablas necesarias para trabajar con los metadatos.

Para trabajar con la herramienta de la línea de ordenes tiene que estar presente un archivo `cli-config.php` en el directorio raíz del proyecto, donde ejecutarás la orden *doctrine*. Es un archivo bastante simple:

```
<?php
$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($entityManager)
));
```

Entonces puedes cambiarte al directorio del proyecto e invocar la línea de ordenes de *Doctrine*:

```
doctrine@my-desktop> cd myproject/
doctrine@my-desktop> doctrine orm:schema-tool:create
```

Nota: La orden `doctrine` sólo estará presente si instalaste *Doctrine* desde PEAR. De lo contrario, tendrás que escarbar en el código de `bin/doctrine.php` en tu directorio de *Doctrine* 2 para configurar la línea de ordenes cliente de *Doctrine*.

Consulta la sección `Herramientas del manual` sobre la correcta configuración de la consola de *Doctrine*.

Durante el desarrollo probablemente tengas que volver a crear la base de datos varias veces al cambiar los metadatos de la Entidad. Entonces, puedes volver a crear la base de datos:

```
doctrine@my-desktop> doctrine orm:schema-tool:drop --force
doctrine@my-desktop> doctrine orm:schema-tool:create
```

O utilizar la funcionalidad de actualización:

```
doctrine@my-desktop> doctrine orm:schema-tool:update --force
```

La actualización de la base de datos utiliza un algoritmo de diferencias para un esquema de base de datos dado, uno de los pilares del paquete `Doctrine\DBAL`, que incluso puedes utilizar sin el paquete ORM de *Doctrine*. Sin embargo, no está disponible en SQLite, ya que no es compatible con `ALTER TABLE`.

2.1.7 Escribiendo entidades en la base de datos

Una vez creado el esquema podemos empezar a trabajar y guardar entidades en la base de datos. Para empezar tenemos que crear un usuario de caso de uso:

```
<?php
$newUsername = "beberlei";

$user = new User();
$user->name = $newUsername;

$entityManager->persist($user);
$entityManager->flush();
```

También puedes crear los productos:

```
<?php
$newProductName = "My Product";

$product = new Product();
$product->name = $newProductName;

$entityManager->persist($product);
$entityManager->flush();
```

¿Qué es lo que está sucediendo en estos dos fragmentos? En ambos ejemplos, la creación de clases es bastante estándar, las piezas interesantes son la comunicación con el `EntityManager`. Para notificar al `EntityManager` que se debe insertar una nueva entidad en la base de datos, para lo cual tienes que llamar a `persist()`. Sin embargo, el `EntityManager` no actúa en esta, es una mera notificación. Tienes que llamar explícitamente a `flush()` para que el `EntityManager` escriba esas dos entidades a la base de datos.

Podrías preguntarte ¿por qué existe esta distinción entre notificación y persistencia? *Doctrine* 2 utiliza el patrón 'Unidad de trabajo' (`UnitOfWork`) para agregar todas las escrituras (`INSERT`, `UDPATE`, `DELETE`) en una sola operación rápida, que se ejecuta al invocar a `flush`. Al usar este enfoque de escritura el rendimiento es significativamente más rápido que en un escenario en el que las actualizaciones para cada entidad se realizan por separado. En escenarios más complejos que los dos anteriores, eres libre de solicitar actualizaciones en muchas entidades diferentes y escribir todas ellas a la vez con `flush`.

La unidad de trabajo de *Doctrine* automáticamente detecta las entidades que han cambiado después de haber recuperado la base de datos, cuando se llama a la operación de vaciado, de modo que sólo tiene que llevar un registro de aquellas entidades que son nuevas o se han removido y pasarlas a `EntityManager#persist()` y `EntityManager#remove()`, respectivamente. Esta comparación para encontrar entidades sucias que es necesario actualizar utiliza un algoritmo muy eficiente, que casi no tiene sobrecarga adicional de memoria y hasta puede ahorrar energía del equipo al sólo actualizar las columnas de la base de datos que realmente han cambiado.

Ahora estamos llegando al requisito "Crea un nuevo fallo" y el código para este escenario podría tener el siguiente aspecto:

```
<?php
$reporter = $entityManager->find("User", $theReporterId);
$engineer = $entityManager->find("User", $theDefaultEngineerId);

$bug = new Bug();
$bug->description = "Something does not work!";
$bug->created = new DateTime("now");
$bug->status = "NEW";

foreach ($productIds AS $productId) {
    $product = $entityManager->find("Product", $productId);
    $bug->assignToProduct($product);
}

$bug->setReporter($reporter);
$bug->setEngineer($engineer);

$entityManager->persist($bug);
$entityManager->flush();

echo "Your new Bug Id: ".$bug->id."\n";
```

Este es el primer contacto con la lectura de la *API* del `EntityManager`, que muestra una llamada a `EntityManager#find($nombre, $id)` la cual devuelve una sola instancia de una entidad consultada por la clave principal. Además de esto vemos que el patrón persiste + vacía de nuevo para guardar el Fallo en la base de datos.

Ve lo simple que es relacionar Fallo, Informante, Ingeniero y Productos y se lleva a cabo usando los métodos descritos en la sección "Un primer prototipo". La unidad de trabajo detectará estas relaciones cuando se invoque a `flush` y los relaciona en la base de datos apropiadamente.

2.1.8 Consultas para casos de uso de la aplicación

Lista de fallos

Usando los ejemplos anteriores podemos rellenar un poco la base de datos, sin embargo, ahora necesitamos explicar la forma de consultar el asignador subyacente para las representaciones de vista necesarias. Al abrir la aplicación, los errores se pueden paginar a través de una lista de vistas, el cual es el primer casos de uso de sólo lectura:

```
<?php
$dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ORDER BY b.created DESC";

$query = $entityManager->createQuery($dql);
$query->setMaxResults(30);
$bugs = $query->getResult();

foreach($bugs AS $bug) {
    echo $bug->description." - ".$bug->created->format('d.m.Y')."\n";
    echo "    Reported by: ".$bug->getReporter()->name."\n";
    echo "    Assigned to: ".$bug->getEngineer()->name."\n";
    foreach($bug->getProducts() AS $product) {
        echo "    Platform: ".$product->name."\n";
    }
    echo "\n";
}
```

La consulta DQL en este ejemplo recupera los 30 errores más recientes con sus respectivos ingeniero e informante en una sola declaración SQL. Entonces, la salida de la consola de este guión es:

```
¡Algo no funciona! - 02.04.2010
    Reported by: beberlei
    Assigned to: beberlei
    Platform: My Product
```

Nota: DQL no es SQL

Puedes preguntarte por qué empezamos a escribir SQL al principio de este caso de uso. ¿No utilizamos un ORM para deshacernos al fin de toda la escritura a mano de SQL? *Doctrine* introduce DQL el cual se describe mejor como **lenguaje de objeto consulta** y es un dialecto del OQL y similar a HQL o JPQL. No conoce el concepto de tablas y columnas, sino sólo los de clase Entidad y propiedad. Usando los metadatos que definimos anteriormente nos permite realizar consultas distintivas y poderosas muy cortas.

Una razón importante de por qué DQL es favorable a la *API* de consulta de la mayoría de los ORM es su similitud con SQL. El lenguaje DQL permite construcciones de consulta que la mayoría de los ORM no, GROUP BY, incluso con HAVING, Subselecciones, Uniones de recuperación con clases anidadas, resultados mezclados con entidades y datos escalares como resultados COUNT() y mucho más. Utilizando DQL rara vez llegarás al punto en el que desees botar tu ORM a la basura, porque no es compatible con algunos conceptos SQL más poderosos.

Más allá de escribir DQL a mano, no obstante, también puedes utilizar el Generador de consultas recuperado llamando a $entityManager->createQueryBuilder() el cual es un objeto Query en torno al lenguaje DQL.

Como último recurso, sin embargo, también puedes utilizar SQL nativo y una descripción del conjunto de resultados para recuperar entidades desde la base de datos. DQL se reduce hasta a una declaración SQL nativa y una instancia de ResultSetMapping en sí mismo. Usando SQL nativo incluso podrías utilizar procedimientos almacenados para recuperar datos, o usar consultas avanzadas no portables de la base de datos, tal como las consultas recurrentes de PostgreSQL.

Hidratando un arreglo con la lista de fallos

En el caso de uso anterior recuperamos el resultado de las respectivas instancias de objetos. Sin embargo, no nos limitamos a recuperar únicamente objetos desde *Doctrine*. Para una vista de lista simple como la anterior, sólo necesitamos acceso de lectura a nuestras entidades y, en su lugar, puedes cambiar la hidratación de objetos a simples arreglos PHP. Esto, obviamente, puede generar beneficios considerables para el rendimiento de las peticiones de sólo lectura.

Implementando la misma vista de lista que antes usar hidratación de arreglos podemos reescribir nuestro código:

```
<?php
$dql = "SELECT b, e, r, p FROM Bug b JOIN b.engineer e ".
       "JOIN b.reporter r JOIN b.products p ORDER BY b.created DESC";
$query = $entityManager->createQuery($dql);
$bugs = $query->getArrayResult();

foreach ($bugs AS $bug) {
    echo $bug['description'] . " - " . $bug['created']->format('d.m.Y')."\n";
    echo "    Reported by: ".$bug['reporter']['name']."\n";
    echo "    Assigned to: ".$bug['engineer']['name']."\n";
    foreach($bug['products'] AS $product) {
        echo "    Platform: ".$product['name']."\n";
    }
    echo "\n";
}
```

No obstante, hay una diferencia significativa en la consulta `DQL`, tenemos que agregar una unión de recuperación adicional a los productos conectados a un fallo. La consulta `SQL` resultante de esta simple instrucción de selección es bastante grande, sin embargo, todavía más eficaz para recuperar comparada con la hidratación de objetos.

Buscando por clave primaria

El siguiente caso de uso es visualizar un fallo por clave primaria. Esto se podría hacer usando `DQL` como en el ejemplo anterior, con una cláusula `where`, sin embargo, hay un conveniente método en el administrador de Entidad que maneja la carga por la clave primaria, que ya hemos visto en los escenarios descritos:

```
<?php
$bug = $entityManager->find("Bug", (int)$theBugId);
```

Sin embargo, pronto veremos otro problema con nuestras entidades que utilizan este enfoque. Trata de mostrar el nombre del ingeniero:

```
<?php
echo "Bug: ".$bug->description."\n";
echo "Engineer: ".$bug->getEngineer()->name."\n";
```

¡Será nulo! ¿Qué está pasando? Si trabajó en el ejemplo anterior, por lo tanto no puede ser un problema con el código de persistencia de *Doctrine*. ¿Qué es entonces? Caíste en la trampa de la propiedad pública.

Ya que sólo recuperamos el fallo por la clave primaria, tanto el ingeniero como el informante no se cargan inmediatamente desde la base de datos, sino que son reemplazados por cargadores delegados diferidos. El código de ejemplo de este código delegado generado lo puedes encontrar en el directorio delegado especificado, se muestra como:

```
<?php
namespace MiProyecto\Proxies;

/**
 * ESTA CLASE FUE GENERADA POR EL ORM DE DOCTRINE. NO EDITES ESTE ARCHIVO.
 */
```

```
class UserProxy extends \User implements \Doctrine\ORM\Proxy\Proxy
{
    // .. código de carga diferida aquí

    public function addReportedBug($bug)
    {
        $this->_load();
        return parent::addReportedBug($bug);
    }

    public function assignedToBug($bug)
    {
        $this->_load();
        return parent::assignedToBug($bug);
    }
}
```

¿Ves cómo cada llamada al método delegado se carga de manera diferida desde la base de datos? Usando las propiedades públicas sin embargo, nunca llamamos a un método y *Doctrine* no tiene forma de engancharse en el motor de *PHP* para detectar un acceso directo a una propiedad pública y desencadenar la carga diferida. Es necesario volver a escribir nuestras entidades, haciendo todas las propiedades privadas o protegidas y agregando captadores y definidores para conseguir que el ejemplo trabaje:

```
<?php
echo "Bug: ".$bug->getDescription()."\n";
echo "Engineer: ".$bug->getEngineer()->getName()."\n";

/**
Bug: Something does not work!
Engineer: beberlei
*/
```

Es necesario utilizar propiedades privadas o protegidas en *Doctrine* 2 para forzarte a encapsular tus objetos de acuerdo a las buenas prácticas orientadas a objetos.

2.1.9 Panel del usuario

Para el siguiente caso de uso queremos recuperar la vista del panel, una lista de todos los fallos abiertos por el usuario informante o a quién se le asignaron. Esto se logrará con `DQL` de nuevo, esta vez con algunas cláusulas `WHERE` y usando parámetros vinculados:

```
<?php
$dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ".
       "WHERE b.status = 'OPEN' AND e.id = ?1 OR r.id = ?1 ORDER BY b.created DESC";

$myBugs = $entityManager->createQuery($dql)
                        ->setParameter(1, $theUserId)
                        ->setMaxResults(15)
                        ->getResult();

foreach ($myBugs AS $bug) {
    echo $bug->getDescription()."\n";
}
```

Este es para los escenarios de lectura de este ejemplo, vamos a continuar con la última parte faltante, los ingenieros tienen posibilidad de cerrar un fallo.

2.1.10 Número de fallos

Hasta ahora sólo hemos recuperado entidades o su representación en arreglo. *Doctrine* también es compatible con la recuperación de fragmentos de las entidades a través de `DQL`. Estos valores se denominan "valores de resultado escalar", e incluso pueden ser valores agregados usando `COUNT`, `SUM` o las funciones `MIN`, `MAX` o `AVG`.

Vamos a necesitar este conocimiento para recuperar el número de fallos abiertos agrupados por producto:

```
<?php
$dql = "SELECT p.id, p.name, count(b.id) AS openBugs FROM Bug b ".
       "JOIN b.products p WHERE b.status = 'OPEN' GROUP BY p.id";
$productBugs = $entityManager->createQuery($dql)->getScalarResult();

foreach($productBugs as $productBug) {
    echo $productBug['name']." has " . $productBug['openBugs'] . " ¡fallos abiertos!\n";
}
```

2.1.11 Actualizando entidades

No hay un solo caso de uso carente de los requisitos, los ingenieros deben poder cerrar un fallo. Esto se ve así:

```
<?php
$bug = $entityManager->find("Bug", (int)$theBugId);
$bug->close();

$entityManager->flush();
```

Al recuperar el Fallo de la base de datos este se inserta en el `IdentityMap` de la unidad de trabajo de *Doctrine*. Esto significa que el fallo con exactamente este identificador sólo puede existir una vez durante toda la petición sin importar cuántas veces se llame a `EntityManager#find()`. Este detecta incluso las entidades hidratadas con `DQL` y que ya están presentes en el mapa de identidad.

Cuando llames a `flush` del `EntityManager` actúa sobre todas entidades en el mapa de identidad y realiza una comparación entre los valores recuperados originalmente de la base de datos y los valores de tiene la entidad actualmente. Si por lo menos una de estas propiedades es diferente en la entidad se programa para un `UPDATE` en la base de datos. Sólo se actualizan las columnas modificadas, lo cual ofrece una mejora bastante buena en el rendimiento en comparación con la actualización de todas las propiedades.

2.1.12 Repositorios de entidad

Por ahora no hemos hablado de cómo separar la lógica de la consulta *Doctrine* de tu modelo. En *Doctrine* 1 el concepto fueron instancias de `Doctrine_Table` para esta separación. El concepto similar en Doctrine2 se llama `Repositorios de entidad`, integrando el patrón de repositorio en el corazón de *Doctrine*.

Cada entidad utiliza un repositorio predeterminado por omisión y ofrece un montón de métodos útiles que puedes utilizar para consultar las instancias de la Entidad. Tomemos por ejemplo nuestra entidad Producto. Si quisiéramos consultar por su nombre, podemos utilizar:

```
<?php
$product = $entityManager->getRepository('Product')
                         ->findOneBy(array('name' => $productName));
```

El método `findOneBy()` tiene una gran variedad de campos o claves asociadas y valores equivalentes.

Si deseas encontrar todas las entidades que coincidan con una condición puedes utilizar `findBy()`, por ejemplo, para consulta por todos los fallos cerrados:

```php
<?php
$bugs = $entityManager->getRepository('Bug')
                      ->findBy(array('status' => 'CLOSED'));

foreach ($bugs AS $bug) {
    // hace cosas
}
```

Comparado con estos métodos de consulta `DQL` están por debajo de una funcionalidad muy rápido. *Doctrine* ofrece una manera conveniente de ampliar las funcionalidades del `EntityRepository` predeterminado y pone toda la lógica de consulta especializada `DQL` en ella. Para ello, debes crear una subclase de `Doctrine\ORM\EntityRepository`, en nuestro caso un `BugRepository` y agrupar en ella toda la funcionalidad de consulta explicada previamente:

```php
<?php

use Doctrine\ORM\EntityRepository;

class BugRepository extends EntityRepository
{
    public function getRecentBugs($number = 30)
    {
        $dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ORDER BY b.created DESC

        $query = $this->getEntityManager()->createQuery($dql);
        $query->setMaxResults($number);
        return $query->getResult();
    }

    public function getRecentBugsArray($number = 30)
    {
        $dql = "SELECT b, e, r, p FROM Bug b JOIN b.engineer e ".
               "JOIN b.reporter r JOIN b.products p ORDER BY b.created DESC";
        $query = $this->getEntityManager()->createQuery($dql);
        $query->setMaxResults($number);
        return $query->getArrayResult();
    }

    public function getUsersBugs($userId, $number = 15)
    {
        $dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ".
               "WHERE b.status = 'OPEN' AND e.id = ?1 OR r.id = ?1 ORDER BY b.created DESC";

        return $this->getEntityManager()->createQuery($dql)
                    ->setParameter(1, $userId)
                    ->setMaxResults($number)
                    ->getResult();
    }

    public function getOpenBugsByProduct()
    {
        $dql = "SELECT p.id, p.name, count(b.id) AS openBugs FROM Bug b ".
               "JOIN b.products p WHERE b.status = 'OPEN' GROUP BY p.id";
        return $this->getEntityManager()->createQuery($dql)->getScalarResult();
    }
}
```

Para poder utilizar esta lógica de consulta a través de `$this->getEntityManager()->getRepository('Fallo')`

tenemos que ajustar un poco los metadatos.

- *PHP*

```php
<?php
/**
 * @Entity(repositoryClass="BugRepository")
 * @Table(name="bugs")
 */
class Bug
{
    //...
}
```

- *XML*

```xml
<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="Bug" table="bugs" entity-repository="BugRepository">

      </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Product:
  type: entity
  repositoryClass: BugRepository
```

Ahora podemos eliminar nuestra lógica de consulta en todos los lugares y en su lugar utilizarla a través del `EntityRepository`. Como ejemplo aquí está el código del primer caso el uso "Lista de fallos":

```php
<?php
$bugs = $entityManager->getRepository('Bug')->getRecentBugs();

foreach($bugs AS $bug) {
    echo $bug->description." - ".$bug->created->format('d.m.Y')."\n";
    echo "    Reported by: ".$bug->getReporter()->name."\n";
    echo "    Assigned to: ".$bug->getEngineer()->name."\n";
    foreach($bug->getProducts() AS $product) {
        echo "    Platform: ".$product->name."\n";
    }
    echo "\n";
}
```

Utilizando `EntityRepositories` puedes evitar el acoplamiento con la lógica de tu modelo de consulta específico. También puedes reutilizar la lógica de consulta fácilmente a través de tu aplicación.

2.1.13 Conclusión

Hasta aquí llegamos en esta guía, espero te hayas divertido. Incrementalmente añadiremos contenido adicional a esta guía, los temas deben incluir:

- Más información sobre la asociación de asignaciones
- Eventos del ciclo de vida activo en la unidad de trabajo

- Ordenamiento de colecciones

Puedes encontrar detalles adicionales sobre todos los temas tratados aquí en los respectivos capítulos del manual.

2.2 Trabajando con asociaciones indexadas

Nota: Esta característica está programada para la versión 2.1 de *Doctrine* y no se incluye en la serie 2.0.x.

Las colecciones de *Doctrine* 2 están basadas en los arreglos nativos de *PHP*. Los arreglos de *PHP* son un *HashMap* ordenado, pero en la primera versión de *Doctrine* las claves numéricas menos utilizadas siempre fueron recuperadas de la base de datos con `INDEX BY`. A partir de *Doctrine* 2.1 puedes indexar tus colecciones por un valor en la entidad relacionada. Este es un primer paso hacia el pleno apoyo para ordenar *HashMap* a través del ORM de *Doctrine*. La característica funciona como un `INDEX BY` implícito para la asociación seleccionada, pero también tiene varias desventajas:

- Tú tienes que gestionar la clave y el campo si quieres cambiar el valor del campo índice.
- En cada petición, las claves se regeneran a partir del valor del campo clave no desde la colección anterior.
- Los valores de las claves `INDEX BY` no se consideran durante la persistencia, existen sólo para propósitos de acceso.
- Los campos que se utilizan para la característica `index by` **TIENEN** que ser únicos en la base de datos. El comportamiento de múltiples entidades con el mismo valor del campo `index by` no está definido.

Como ejemplo vamos a diseñar un simple intercambio de valores de la vista lista. El dominio se compone de la entidad `Acciones` y `Mercado` en el que cada `Acción` tiene un símbolo y se negocia en un único mercado. En lugar de tener una lista numérica de las acciones negociadas en un mercado esta será indexada por su símbolo, el cual es único en todos los mercados.

2.2.1 Asignando las asociaciones indexadas

Puedes asignar asociaciones indexadas añadiendo:

- El atributo `indexBy` a cualquier anotación `@OneToMany` o `@ManyToMany`.
- El atributo `index-by` a cualquier elemento XML `<one-to-many />` o `<many-to-many />`.
- El par clave/valor `indexBy:` a cualquier asociación `manyToMany:` o `oneToMany:` definida en los archivos de asignación YML.

El código y las asignaciones para la entidad Mercado es la siguiente:

- *PHP*

```
<?php
namespace Doctrine\Tests\Models\StockExchange;

use Doctrine\Common\Collections\ArrayCollection;

/**
 * @Entity
 * @Table(name="exchange_markets")
 */
class Market
{
    /**
```

```
     * @Id @Column(type="integer") @GeneratedValue
     * @var int
     */
    private $id;

    /**
     * @Column(type="string")
     * @var string
     */
    private $nombre;

    /**
     * @OneToMany(targetEntity="Stock", mappedBy="market", indexBy="symbol")
     * @var Stock[]
     */
    private $stocks;

    public function __construct($nombre)
    {
        $this->name = $nombre;
        $this->stocks = new ArrayCollection();
    }

    public function getId()
    {
        return $this->id;
    }

    public function getName()
    {
        return $this->name;
    }

    public function addStock(Stock $stock)
    {
        $this->stocks[$stock->getSymbol()] = $stock;
    }

    public function getStock($symbol)
    {
        if (!isset($this->stocks[$symbol])) {
            throw new \InvalidArgumentException("Symbol is not traded on this market.");
        }

        return $this->stocks[$symbol];
    }

    public function getStocks()
    {
        return $this->stocks->toArray();
    }
}
```

- *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://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Doctrine\Tests\Models\StockExchange\Market">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>

        <field name="name" type="string"/>

        <one-to-many target-entity="Stock" mapped-by="market" field="stocks" index-by="symbol" /
    </entity>
</doctrine-mapping>
```

- *YAML*

```
Doctrine\Tests\Models\StockExchange\Market:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type:string
  oneToMany:
    stocks:
      targetEntity: Stock
      mappedBy: market
      indexBy: symbol
```

Dentro del método `addStock()` puedes ver cómo establecer directamente la clave de la asociación para el símbolo, de modo que podamos trabajar directamente con la asociación indizada después de invocar `addStock()`. Dentro de `getStock($symbol)` tomamos una acción negociada en un mercado en particular por el símbolo. Si esta acción no existe desencadenamos una excepción.

La entidad `Acción` no contiene todas las instrucciones especiales que son nuevas, pero está completa aquí está el código y las asignaciones para la misma:

- *PHP*

```
<?php
namespace Doctrine\Tests\Models\StockExchange;

/**
 * @Entity
 * @Table(name="exchange_stocks")
 */
class Stock
{
    /**
     * @Id @GeneratedValue @Column(type="integer")
     * @var int
     */
    private $id;

    /**
     * En realidad esta columna tendría que se unique=true. Pero deseo probar el comportamiento
     *
```

```
     * @Column(type="string", unique=true)
     */
    private $symbol;

    /**
     * @ManyToOne(targetEntity="Market", inversedBy="stocks")
     * @var Market
     */
    private $market;

    public function __construct($symbol, Market $market)
    {
        $this->symbol = $symbol;
        $this->market = $market;
        $market->addStock($this);
    }

    public function getSymbol()
    {
        return $this->symbol;
    }
}
```

- *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://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Doctrine\Tests\Models\StockExchange\Stock">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>

        <field name="symbol" type="string" unique="true" />
        <many-to-one target-entity="Market" field="market" inversed-by="stocks" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
Doctrine\Tests\Models\StockExchange\Stock:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    symbol:
      type: string
  manyToOne:
    market:
      targetEntity: Market
      inversedBy: stocks
```

2.2.2 Consultando asociaciones indexadas

Ahora que hemos definido la colección de acciones para que sean indexadas por los símbolos podemos echar un vistazo a algo de código, que usaremos en la indexación.

En primer lugar vamos a poblar nuestra base de datos con dos acciones de ejemplo negociadas en un único mercado:

```
<?php
// $em es el EntityManager

$market = new Market("Some Exchange");
$stock1 = new Stock("AAPL", $market);
$stock2 = new Stock("GOOG", $market);

$em->persist($market);
$em->persist($stock1);
$em->persist($stock2);
$em->flush();
```

Este código no es especialmente interesante, ya que la característica de indexación todavía no se utiliza. En una nueva petición ahora podemos consultar por el mercado:

```
<?php
// $em is the EntityManager
$marketId = 1;
$symbol = "AAPL";

$market = $em->find("Doctrine\Tests\Models\StockExchange\Market", $marketId);

// Ahora accede a la acción por símbolo:
$stock = $market->getStock($symbol);

echo $stock->getSymbol(); // imprimirá "AAPL"
```

La implementación de `Mercado::addStock()` en combinación con `indexBy` permite acceder a la colección consistentemente por el símbolo acción. No importa si las acciones son gestionadas por *Doctrine* o no.

Lo mismo se aplica a las consultas `DQL`: La configuración `indexBy` actúa como "INDEX BY" implícita para unir la asociación.

```
<?php
// $em es el EntityManager
$marketId = 1;
$symbol = "AAPL";

$dql = "SELECT m, s FROM Doctrine\Tests\Models\StockExchange\Market m JOIN m.stocks s WHERE m.id = ?1
$market = $em->createQuery($dql)
             ->setParameter(1, $marketId)
             ->getSingleResult();

// Ahora accede a la acción por símbolo:
$stock = $market->getStock($symbol);

echo $stock->getSymbol(); // imprimirá "AAPL"
```

Si deseas utilizar el `INDEX BY` explícitamente en una asociación indexada estás en libertad de hacerlo. Además, las asociaciones indexadas también trabajan con la funcionalidad `Collection::slice()`, sin importar si están marcadas como `LAZY` o `EXTRA_LAZY`.

2.2.3 Perspectiva a futuro

Por el lado inverso de una asociación de muchos a muchos, habrá una manera de mantener las claves y el orden como un tercer y cuarto parámetro en la unión de tablas. Esta característica se explica en DDC-213 esta función no se puede implementar en algunas de las asociaciones uno-a-Muchos, porque no son la parte propietaria.

2.3 Extra Lazy Associations

In many cases associations between entities can get pretty large. Even in a simple scenario like a blog. where posts can be commented, you always have to assume that a post draws hundrets of comments. In Doctrine 2.0 if you accessed an association it would always get loaded completly into memory. This can lead to pretty serious performance problems, if your associations contain several hundrets or thousands of entities.

With Doctrine 2.1 a feature called **Extra Lazy** is introduced for associations. Associations are marked as **Lazy** by default, which means the whole collection object for an association is populated the first time its accessed. If you mark an association as extra lazy the following methods on collections can be called without triggering a full load of the collection:

- `Collection#contains($entity)`
- `Collection#count()`
- `Collection#slice($offset, $length = null)`

For each of this three methods the following semantics apply:

- For each call, if the Collection is not yet loaded, issue a straight SELECT statement against the database.
- For each call, if the collection is already loaded, fallback to the default functionality for lazy collections. No additional SELECT statements are executed.

Additionally even with Doctrine 2.0 the following methods do not trigger the collection load:

- `Collection#add($entity)`
- `Collection#offsetSet($key, $entity)` - ArrayAccess with no specific key `$coll[] = $entity`, it does not work when setting specific keys like `$coll[0] = $entity`.

With extra lazy collections you can now not only add entities to large collections but also paginate them easily using a combination of `count` and `slice`.

2.3.1 Enabling Extra-Lazy Associations

The mapping configuration is simple. Instead of using the default value of `fetch="LAZY"` you have to switch to extra lazy as shown in these examples:

- *PHP*

```php
<?php
namespace Doctrine\Tests\Models\CMS;

/**
 * @Entity
 */
class CmsGroup
{
    /**
     * @ManyToMany(targetEntity="CmsUser", mappedBy="groups", fetch="EXTRA_LAZY")
```

```
     */
    public $users;
}
```

- *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://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Doctrine\Tests\Models\CMS\CmsGroup">
        <!-- ... -->
        <many-to-many field="users" target-entity="CmsUser" mapped-by="groups" fetch="EXTRA_LAZY
    </entity>
</doctrine-mapping>
```

- *YAML*

```
Doctrine\Tests\Models\CMS\CmsGroup:
  type: entity
  # ...
  manyToMany:
    users:
      targetEntity: CmsUser
      mappedBy: groups
      fetch: EXTRA_LAZY
```

2.4 Composite and Foreign Keys as Primary Key

Doctrine 2 nativamente es compatible con claves primarias compuestas. Las claves compuestas son un muy potente concepto de bases de datos relacional y debemos tener buen cuidado para que la *Doctrine* 2 sea compatible con el mayor número de casos de uso de claves primarias. For Doctrine 2.0 composite keys of primitive data-types are supported, for Doctrine 2.1 even foreign keys as primary keys are supported.

This tutorial shows how the semantics of composite primary keys work and how they map to the database.

2.4.1 Consideraciones generales

Every entity with a composite key cannot use an id generator other than "ASSIGNED". That means the ID fields have to have their values set before you call `EntityManager#persist($entity)`.

2.4.2 Primitive Types only

Even in version 2.0 you can have composite keys as long as they only consist of the primative types `integer` and `string`. Suppose you want to create a database of cars and use the model-name and year of production as primary keys:

- *PHP*

```
<?php
namespace VehicleCatalogue\Model;
```

```php
/**
 * @Entity
 */
class Car
{
    /** @Id @Column(type="string") */
    private $nombre;
    /** @Id @Column(type="integer") */
    private $year

    public function __construct($nombre, $year)
    {
        $this->name = $nombre;
        $this->year = $year;
    }

    public function getModelName()
    {
        return $this->name;
    }

    public function getYearOfProduction()
    {
        return $this->year;
    }
}
```

- *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://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="VehicleCatalogue\Model\Car">
        <id field="name" type="string" />
        <id field="year" type="integer" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
VehicleCatalogue\Model\Car:
  type: entity
  id:
    name:
      type: string
    year:
      type: integer
```

Ahora puedes utilizar esta entidad:

```php
<?php
namespace VehicleCatalogue\Model;

// $em es el EntityManager

$car = new Car("Audi A8", 2010);
```

```
$em->persist($car);
$em->flush();
```

Y para consultar puedes utilizar matrices tanto `DQL` como EntityRepositories:

```
<?php
namespace VehicleCatalogue\Model;

// $em es el EntityManager
$audi = $em->find("VehicleCatalogue\Model\Car", array("name" => "Audi A8", "year" => 2010));

$dql = "SELECT c FROM VehicleCatalogue\Model\Car c WHERE c.id = ?1";
$audi = $em->createQuery($dql)
           ->setParameter(1, array("name" => "Audi A8", "year" => 2010))
           ->getSingleResult();
```

También puedes utilizar esta entidad en asociaciones. *Doctrine* luego generará dos claves externas una para `name` y `year` para las entidades relacionadas.

Nota: This example shows how you can nicely solve the requirement for exisiting values before `EntityManager#persist()`: By adding them as mandatory values for the constructor.

2.4.3 Identity through foreign Entities

Nota: Identity through foreign entities is only supported with Doctrine 2.1

There are tons of use-cases where the identity of an Entity should be determined by the entity of one or many parent entities.

- Dynamic Attributes of an Entity (for example Article). Each Article has many attributes with primary key "article_id" and "attribute_name".
- Address object of a Person, the primary key of the adress is "user_id". This is not a case of a composite primary key, but the identity is derived through a foreign entity and a foreign key.
- Join Tables with metadata can be modelled as Entity, for example connections between two articles with a little description and a score.

The semantics of mapping identity through foreign entities are easy:

- Only allowed on Many-To-One or One-To-One associations.
- Plug an `@Id` annotation onto every assocation.
- Set an attribute `association-key` with the field name of the association in XML.
- Set a key `associationKey:` with the field name of the association in YAML.

2.4.4 Use-Case 1: Dynamic Attributes

We keep up the example of an Article with arbitrary attributes, the mapping looks like this:

- *PHP*

```php
<?php
namespace Application\Model;

use Doctrine\Common\Collections\ArrayCollection;

/**
 * @Entity
 */
class Article
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;
    /** @Column(type="string") */
    private $title;

    /**
     * @OneToMany(targetEntity="ArticleAttribute", mappedBy="article", cascade={"ALL"}, indexBy=
     */
    private $attributes;

    public function addAttribute($name, $value)
    {
        $this->attributes[$name] = new ArticleAttribute($name, $value, $this);
    }
}

/**
 * @Entity
 */
class ArticleAttribute
{
    /** @Id @ManyToOne(targetEntity="Article", inversedBy="attributes") */
    private $article;

    /** @Id @Column(type="string") */
    private $attribute;

    /** @Column(type="string") */
    private $value;

    public function __construct($name, $value, $article)
    {
        $this->attribute = $name;
        $this->value = $value;
        $this->article = $article;
    }
}
```

- *XML*

```xml
<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="Application\Model\ArticleAttribute">
        <id name="article" association-key="true" />
        <id name="attribute" type="string" />
```

```
            <field name="value" type="string" />

            <many-to-one field="article" target-entity="Article" inversed-by="attributes" />
        <entity>

    </doctrine-mapping>
```

- *YAML*

```
Application\Model\ArticleAttribute:
  type: entity
  id:
    article:
      associationKey: true
    attribute:
      type: string
  fields:
    value:
      type: string
  manyToOne:
    article:
      targetEntity: Article
      inversedBy: attributes
```

2.4.5 Use-Case 2: Simple Derived Identity

Sometimes you have the requirement that two objects are related by a One-To-One association and that the dependent class should re-use the primary key of the class it depends on. One good example for this is a user-address relationship:

```
<?php
/**
 * @Entity
 */
class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;
}

/**
 * @Entity
 */
class Address
{
    /** @Id @OneToOne(targetEntity="User") */
    private $user;
}
```

2.4.6 Use-Case 3: Join-Table with Metadata

In the classic order product shop example there is the concept of the order item which contains references to order and product and additional data such as the amount of products purchased and maybe even the current price.

```
<?php
use Doctrine\Common\Collections\ArrayCollection;
```

```php
/** @Entity */
class Order
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;

    /** @ManyToOne(targetEntity="Customer") */
    private $customer;
    /** @OneToMany(targetEntity="OrderItem", mappedBy="order") */
    private $items;

    /** @Column(type="boolean") */
    private $payed = false;
    /** @Column(type="boolean") */
    private $shipped = false;
    /** @Column(type="datetime") */
    private $created;

    public function __construct(Customer $customer)
    {
        $this->customer = $customer;
        $this->items = new ArrayCollection();
        $this->created = new \DateTime("now");
    }
}

/** @Entity */
class Product
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;

    /** @Column(type="string")
    private $nombre;

    /** @Column(type="decimal")
    private $currentPrice;

    public function getCurrentPrice()
    {
        return $this->currentPrice;
    }
}

/** @Entity */
class OrderItem
{
    /** @Id @ManyToOne(targetEntity="Order") */
    private $order;

    /** @Id @ManyToOne(targetEntity="Product") */
    private $product;

    /** @Column(type="integer") */
    private $amount = 1;

    /** @Column(type="decimal") */
    private $offeredPrice;
```

```
    public function __construct(Order $order, Product $product, $amount = 1)
    {
        $this->order = $order;
        $this->product = $product;
        $this->offeredPrice = $product->getCurrentPrice();
    }
}
```

2.4.7 Consideraciones sobre rendimiento

Using composite keys always comes with a performance hit compared to using entities with a simple surrogate key. This performance impact is mostly due to additional PHP code that is necessary to handle this kind of keys, most notably when using derived identifiers.

Por el lado de SQL no hay mucha sobrecarga como consultas adicionales o imprevistas puesto que se tienen que ejecutar para gestionar las entidades con claves externas derivadas.

Recetario

3.1 Campos agregados

Autor de la sección: Benjamin Eberlei <kontakt@beberlei.de>

A menudo te encontrarás con la obligación de mostrar los valores de datos agregados que se pueden calcular utilizando funciones `SQL` como `MIN`, `MAX`, `COUNT` o `SUM`. Para cualquier ORM, tradicionalmente, este es un tema complicado. *Doctrine* 2 ofrece varias maneras de acceder a estos valores y este artículo describe todas ellas desde diferentes perspectivas.

Verás que los campos agregados pueden llegar a ser características muy explícitas en tu modelo del dominio y cómo, potencialmente fácil, puedes probar las complejas reglas del negocio.

3.1.1 Un modelo de ejemplo

Digamos que quieres un modelo de cuenta bancaria y todos sus movimientos. Los movimientos en la cuenta pueden ser valores monetarios positivos o negativos. Cada cuenta tiene un límite de crédito y no se permite a la cuenta tener un balance por debajo de ese valor.

Para simplificarnos la vida habitamos un mundo dónde el dinero se compone sólo de números enteros. También omitimos el nombre del receptor/emisor, la indicación de la razón para la transferencia y la fecha de ejecución. Todo ello se tendría que añadir al objeto `Movimiento`.

Nuestras entidades tienen este aspecto:

```
<?php
namespace Banco\Entidades;

/**
 * @Entity
 */
class Cuenta
{
    /** @Id @GeneratedValue @Column(type="integer") */
    private $id;

    /** @Column(type="string", unique=true) */
    private $no;
```

```
    /**
     * @OneToMany(targetEntity="Movimiento", mappedBy="cuenta", cascade={"persist"})
     */
    private $movimientos;

    /**
     * @Column(type="integer")
     */
    private $creditoMax = 0;

    public function __construct($no, $creditoMax = 0)
    {
        $this->no = $no;
        $this->creditoMax = $creditoMax;
        $this->movimientos = new \Doctrine\Common\Collections\ArrayCollection();
    }
}

/**
 * @Entity
 */
class Movimiento
{
    /** @Id @GeneratedValue @Column(type="integer") */
    private $id;

    /**
     * @ManyToOne(targetEntity="Cuenta", inversedBy="movimientos")
     */
    private $cuenta;

    /**
     * @Column(type="integer")
     */
    private $monto;

    public function __construct($cuenta, $monto)
    {
        $this->cuenta = $cuenta;
        $this->monto = $monto;
        // más información aquí, de/para quién, concepto, fecha de aplicación y tal
    }

    public function getMonto()
    {
        return $this->monto;
    }
}
```

3.1.2 Usando DQL

El *Lenguaje de Consulta Doctrine* te permite seleccionar los valores agregados calculados a partir de los campos de tu modelo del dominio. Puedes seleccionar el balance actual de tu cuenta llamando a:

```
<?php
$dql = "SELECT SUM(e.monto) AS balance FROM Banco\Entidades\Movimiento e " .
       "WHERE e.cuenta = ?1";
```

```
$balance = $em->createQuery($dql)
               ->setParameter(1, $idDeMiCuenta)
               ->getSingleScalarResult();
```

La variable `$em` en este ejemplo (y futuros) mantiene al `EntityManager` de *Doctrine*. Creamos una consulta por sumando todos los montos (las cantidades negativas son retiros) y los recuperamos como un único resultado escalar, esencialmente, devolvemos sólo la primera columna de la primera fila.

Este enfoque es simple y poderoso, sin embargo, tiene un grave inconveniente. Tenemos que ejecutar una consulta específica para el balance siempre que lo necesitemos.

Para implementar un potente modelo del dominio, en su lugar tendríamos que acceder al saldo de nuestra entidad `Cuenta` todas las veces (¡incluso si la cuenta no se hubiera persistido anteriormente en la base de datos!).

También un requisito adicional es la regla del crédito máximo por `Cuenta`.

No podemos confiablemente hacer cumplir esta regla en nuestra entidad `Cuenta` recuperando el balance con `DQL`. Hay muchas maneras de recuperar cuentas. No podemos garantizar que se pueda ejecutar la consulta agregando todos estos casos de uso, y mucho menos que un programador en el entorno de usuario compruebe el balance con nuevas entradas recién agregadas.

3.1.3 Utilizando tu modelo del dominio

La `Cuenta` y todas las instancias de `Movimiento` están conectadas a través de una colección, lo cual significa que podemos calcular este valor en tiempo de ejecución:

```
<?php
class Cuenta
{
    // .. código anterior
    public function getBalance()
    {
        $balance = 0;
        foreach ($this->movimientos AS $movimiento) {
            $balance += $movimiento->getMonto();
        }
        return $balance;
    }
}
```

Ahora, siempre podemos llamar a `Cuenta::getBalance()` para acceder al balance actual de la cuenta.

Para aplicar la regla de crédito máximo, tenemos que implementar el patrón *"Raíz agregada"* como se describe en el libro de Eric Evans en `Domain Driven Design`. Descrito con una frase, una raíz agregada controla la creación de la instancia, el acceso y manipulación de sus hijos.

En nuestro caso queremos forzar que sólo se puedan agregar nuevos movimientos a la `Cuenta` usando un método designado. La `Cuenta` es la raíz agregada de esta relación. También podemos forzar la corrección bidireccional de la relación `Cuenta Movimiento` con este método:

```
<?php
class Cuenta
{
    public function addMovimiento($monto)
    {
        $this->assertAcceptEntryAllowed($monto);

        $e = new Movimiento($this, $monto);
        $this->movimientos[] = $e;
```

```
            return $e;
        }
    }
```

Ahora mira el siguiente código de prueba para nuestras entidades:

```
<?php
class CuentaTest extends \PHPUnit_Framework_TestCase
{
    public function testAddMovimiento()
    {
        $cuenta = new Cuenta("123456", $creditoMax = 200);
        $this->assertEquals(0, $cuenta->getBalance());

        $cuenta->addMovimiento(500);
        $this->assertEquals(500, $cuenta->getBalance());

        $cuenta->addMovimiento(-700);
        $this->assertEquals(-200, $cuenta->getBalance());
    }

    public function testExcedeLimiteMax()
    {
        $cuenta = new Cuenta("123456", $creditoMax = 200);

        $this->setExpectedException("Exception");
        $cuenta->addMovimiento(-1000);
    }
}
```

Para hacer cumplir nuestras reglas, ahora podemos implementar la aserción en `Cuenta::addMovimiento`:

```
<?php
class Cuenta
{
    private function assertAcceptEntryAllowed($monto)
    {
        $balanceFuturo = $this->getBalance() + $monto;
        $balanceMinimoPermitido = ($this->creditoMax * -1);
        if ($balanceFuturo < $balanceMinimoPermitido) {
            throw new Exception("Excede el líite de crédito, ¡movimiento no permitido!");
        }
    }
}
```

Hasta ahora no hemos hablado con el gestor de la entidad para persistir nuestro ejemplar de cuenta. Puedes llamar a `EntityManager::persist($cuenta)` y `EntityManager::flush()` en cualquier momento para guardar la cuenta en la base de datos. Todos los objetos `Movimiento` anidados se vacían automáticamente a la base de datos también.

```
<?php
$cuenta = new Cuenta("123456", 200);
$cuenta->addMovimiento(500);
$cuenta->addMovimiento(-200);
$em->persist($cuenta);
$em->flush();
```

La implementación actual tiene una desventaja considerable. Para conseguir el balance, tenemos que iniciar la colección `Cuenta::$movimientos` completa, posiblemente, una muy grande. Esto puede afectar considerablemente el

rendimiento de tu aplicación.

3.1.4 Usando un campo agregado

Para superar el problema mencionado anteriormente (iniciar toda la colección de movimientos) pretendemos añadir un campo agregado a la cuenta llamado "balance" y ajustar el código en `Cuenta::getBalance()` y `Cuenta:addMovimiento()`:

```
<?php
class Cuenta
{
    /**
     * @Column(type="integer")
     */
    private $balance = 0;

    public function getBalance()
    {
        return $this->balance;
    }

    public function addMovimiento($monto)
    {
        $this->assertAcceptEntryAllowed($monto);

        $e = new Movimiento($this, $monto);
        $this->movimientos[] = $e;
        $this->balance += $monto;
        return $e;
    }
}
```

Este es un cambio muy simple, pero todavía pasan todas las pruebas. Nuestras entidades Cuenta devuelven el balance correcto. Ahora llamando al método `Cuenta::getBalance()` no se producirá más la sobrecarga de cargar todos los movimientos. Al agregar un nuevo Movimiento a `Cuenta::$movimientos` tampoco inicia la colección internamente.

Por lo tanto, agregar un nuevo movimiento es muy rápido y se engancha explícitamente en el modelo del dominio. Esto sólo actualizará la cuenta con el saldo actual e insertará el nuevo movimiento en la base de datos.

3.1.5 Enfrentando condiciones de carrera con campos agregados

Cada vez que desnormalizas el esquema de tu base de datos, hay ciertas condiciones que, potencialmente, te pueden conducir a un estado inconsistente. Ve este ejemplo:

```
<?php
// La cuenta $accId tiene un balance de 0 y un límite de crédito máximo de 200:
// petición a la cuenta 1
$cuenta1 = $em->find('Banco\Entidades\Cuenta', $accId);

// petición a la cuenta 2
$cuenta2 = $em->find('Banco\Entidades\Cuenta', $accId);

$cuenta1->addMovimiento(-200);
$cuenta2->addMovimiento(-200);
```

```
// ahora ambas peticiones 1 y 2 vuelcan sus cambios.
```

El campo agregado `Cuenta::$balance` ahora es de -200, sin embargo, la suma de todas las cantidades de las entradas resulta en -400. Una violación a nuestra regla de crédito máximo.

Puedes utilizar el bloqueo optimista o pesimista, tanto para salvar como para vigilar tus campos agregados contra este tipo de condición. Leyendo cuidadosamente a Eric Evans DDD menciona que la "Raíz agregada" (Cuenta, en nuestro ejemplo) necesita un mecanismo de bloqueo.

El bloqueo optimista es tan fácil como añadir una columna de versión:

```
<?php
class Monto
{
    /** @Column(type="integer") @Version */
    private $version;
}
```

En el ejemplo anterior entonces lanzaría una excepción a la cara de cualquier petición para guardar la última entidad (y crearía el estado inconsistente).

El bloqueo pesimista requiere establecer una bandera adicional en las llamadas a `EntityManager::find()`, la cual permita escribir bloqueando directamente la base de datos utilizando un `FOR UPDATE`.

```
<?php
use Doctrine\DBAL\LockMode;

$cuenta = $em->find('Banco\Entidades\Cuenta', $accId, LockMode::PESSIMISTIC_READ);
```

3.1.6 Manteniendo sincronizadas las actualizaciones y eliminaciones

El ejemplo mostrado en este artículo no permite cambios en el valor de `Movimiento`, lo cual simplifica considerablemente el esfuerzo de mantener sincronizada `Cuenta::$balance`. Si tu caso de uso permite que se actualicen campos o se remuevan entidades relacionadas tienes que encapsular esta lógica en la "raíz agregada" de la entidad y ajustar el campo agregado en consecuencia.

3.1.7 Conclusión

Este artículo describe cómo obtener valores agregados usando `DQL` o tu modelo del dominio. Demuestra lo fácil que es añadir un campo agregado que ofrece grandes beneficios de rendimiento, en lugar de iterar sobre todos los objetos relacionados que constituyen un valor agregado. Por último, te mostré cómo puedes garantizar que tus campos agregados no pierdan la sincronía por condiciones de carrera y acceso simultáneo.

3.2 Extendiendo `DQL` en *Doctrine 2*: Paseantes `AST` personalizados

Autor de la sección: Benjamin Eberlei <kontakt@beberlei.de>

El *Lenguaje de Consulta Doctrine* (`DQL`) es un dialecto `sql` propietario que sustituye los nombres de tablas y columnas por nombres de Entidad y sus campos. Utilizando `DQL` escribes una consulta contra la base de datos usando tus entidades. Con la ayuda de los metadatos puedes escribir consultas muy concisas, compactas y potentes que el `ORM` de *Doctrine* traduce en `SQL`.

En *Doctrine* 1 el lenguaje `DQL` no se implementó utilizando un analizador real. Esto imposibilitó las modificaciones por el usuario al `DQL`. *Doctrine* 2 en contraste tiene un analizador real para el lenguaje `DQL`, el cual transforma la

declaración DQL en un árbol de sintaxis abstracta y genera la declaración SQL correspondiente. Dado que este proceso es determinista *Doctrine* en gran medida almacena en caché el SQL que se genera a partir de cualquier consulta DQL dada, lo cual reduce a cero la sobrecarga en el rendimiento del proceso de análisis.

Puedes modificar el árbol de sintaxis abstracta conectándolo en el proceso de análisis DQL añadiendo un paseante personalizado para el árbol. Un paseante es una interfaz que camina cada nodo del árbol de sintaxis abstracta, generando la declaración SQL.

Hay dos tipos de paseantes personalizados para el árbol que puedes conectar al analizador DQL:

- Una paseante de salida. Este en realidad genera el SQL, y solo hay uno de ellos. Implementamos la aplicación predeterminada de SqlWalker para ello.
- Un paseante del árbol. No puede haber muchos paseantes del árbol, no pueden generar el código SQL, sin embargo, puedes modificar el AST antes de reproducir a sql.

Ahora todo esto es muy técnico, así que me vienen a algunos rápidos casos de uso para mantenerte motivado. Usando la implementación de paseante puedes, por ejemplo:

- Modificar el AST para generar una consulta Count para utilizarla con un paginador para cualquier consulta DQL dada.
- Modificar la salida del paseante para generar SQL específicas al proveedor (en lugar de ANSI).
- Modificar el AST para agregar más cláusulas where para entidades específicas (por ejemplo, ACL, contenido específico del país...)
- Modificar la salida del paseante para imprimir elegantemente el SQL para fines de depuración.

En esta parte del recetario voy a mostrar ejemplos de los dos primeros puntos. Probablemente hay mucho más casos de uso.

3.2.1 Consulta count genérica para paginación

Digamos que tienes un *blog* y todos los mensajes con una categoría y un autor. Una consulta para la primera página o para cualquier archivo de página podría ser algo como esta:

```
SELECT p, c, a FROM BlogPost p JOIN p.category c JOIN p.author a WHERE ...
```

Ahora, en esta consulta el *blog* es la entidad raíz, es decir, es la que se hidrata directamente desde la consulta y devuelta como una matriz de comunicados del *blog*. En cambio, el comentario y el autor se cargan desde más adentro usando el árbol de objetos.

Una paginación para esta consulta debería aproximar el número de mensajes que coinciden con la cláusula WHERE de esta consulta para ser capaces de predecir el número de páginas a mostrar al usuario. Un borrador de la consulta DQL para la paginación se vería así:

```
SELECT count(DISTINCT p.id) FROM BlogPost p JOIN p.category c JOIN p.author a WHERE ...
```

Ahora puedes ir y escribir cada una de estas consultas a mano, o puede utilizar un paseante para modificar el árbol AST por ti. Vamos a ver cómo se vería la *API* para este caso de uso:

```
<?php
$numPagina = 1;
$consulta = $em->createQuery($dql);
$consulta->setFirstResult( ($numPagina-1) * 20)->setMaxResults(20);

$totalResultados = Paginate::count($consulta);
$resultados = $consulta->getResult();
```

El Paginate::count(Query $consulta) se ve así:

```php
<?php
class Paginador
{
    static public function count(Query $consulta)
    {
        /* @var $countQuery Query */
        $countQuery = clone $consulta;

        $countQuery->setHint(Query::HINT_CUSTOM_TREE_WALKERS, array('DoctrineExtensions\Paginate\Cou
        $countQuery->setFirstResult(null)->setMaxResults(null);

        return $countQuery->getSingleScalarResult();
    }
}
```

It clones the query, resets the limit clause first and max results and registers the `CountSqlWalker` customer tree walker which will modify the AST to execute a count query. La implementación de los paseantes es la siguiente:

```php
<?php
class CountSqlWalker extends TreeWalkerAdapter
{
    /**
     * Walks down a SelectStatement AST node, thereby generating the appropriate SQL.
     *
     * @return string La SQL.
     */
    public function walkSelectStatement(SelectStatement $AST)
    {
        $parent = null;
        $parentName = null;
        foreach ($this->_getQueryComponents() AS $dqlAlias => $qComp) {
            if ($qComp['parent'] === null && $qComp['nestingLevel'] == 0) {
                $parent = $qComp;
                $parentName = $dqlAlias;
                break;
            }
        }

        $pathExpression = new PathExpression(
            PathExpression::TYPE_STATE_FIELD | PathExpression::TYPE_SINGLE_VALUED_ASSOCIATION, $parer
            $parent['metadata']->getSingleIdentifierFieldName()
        );
        $pathExpression->type = PathExpression::TYPE_STATE_FIELD;

        $AST->selectClause->selectExpressions = array(
            new SelectExpression(
                new AggregateExpression('count', $pathExpression, true), null
            )
        );
    }
}
```

This will delete any given select expressions and replace them with a distinct count query for the root entities primary key. This will only work if your entity has only one identifier field (composite keys won't work).

3.2.2 Modify the Output Walker to generate Vendor specific SQL

Most RMDBS have vendor-specific features for optimizing select query execution plans. You can write your own output walker to introduce certain keywords using the Query Hint API. A query hint can be set via `Query::setHint($name, $value)` as shown in the previous example with the `HINT_CUSTOM_TREE_WALKERS` query hint.

We will implement a custom Output Walker that allows to specify the SQL_NO_CACHE query hint.

```
<?php
$dql = "SELECT p, c, a FROM BlogPost p JOIN p.category c JOIN p.author a WHERE ...";
$query = $m->createQuery($dql);
$query->setHint(Query::HINT_CUSTOM_OUTPUT_WALKER, 'DoctrineExtensions\Query\MysqlWalker');
$query->setHint("mysqlWalker.sqlNoCache", true);
$results = $query->getResult();
```

Our `MysqlWalker` will extend the default `SqlWalker`. We will modify the generation of the SELECT clause, adding the SQL_NO_CACHE on those queries that need it:

```
<?php
class MysqlWalker extends SqlWalker
{
     /**
      * Walks down a SelectClause AST node, thereby generating the appropriate SQL.
      *
      * @param $selectClause
      * @return string The SQL.
      */
    public function walkSelectClause($selectClause)
    {
        $sql = parent::walkSelectClause($selectClause);

        if ($this->getQuery()->getHint('mysqlWalker.sqlNoCache') === true) {
            if ($selectClause->isDistinct) {
                $sql = str_replace('SELECT DISTINCT', 'SELECT DISTINCT SQL_NO_CACHE', $sql);
            } else {
                $sql = str_replace('SELECT', 'SELECT SQL_NO_CACHE', $sql);
            }
        }

        return $sql;
    }
}
```

Writing extensions to the Output Walker requires a very deep understanding of the DQL Parser and Walkers, but may offer your huge benefits with using vendor specific features. Esto te permitiría escribir consultas *DQL* en lugar de consultas nativas para usar las características específicas del proveedor.

3.3 Funciones *DQL* definidas por el usuario

Autor de la sección: Benjamin Eberlei <kontakt@beberlei.de>

Por omisión *DQL* admite un subconjunto limitado de todas las funciones específicas del proveedor *SQL* común entre todos los proveedores. Sin embargo, en muchos casos, una vez te hayas decidido por un proveedor de base de datos específico, nunca lo vas a cambiar durante la vida de tu proyecto. Esta decisión de un proveedor específico potencialmente te permite usar potentes características de *SQL* que son únicas para el proveedor.

Vale la pena mencionar que *Doctrine 2* también te permite escribir tu *SQL* a mano en lugar de extender el analizador *DQL*. Extender *DQL* es una especie de punto de extensión avanzado. You can map arbitrary SQL to your objects and gain access to vendor specific functionalities using the `EntityManager#createNativeQuery()` API as described in the *Native Query* chapter.

The DQL Parser has hooks to register functions that can then be used in your DQL queries and transformed into SQL, allowing to extend Doctrines Query capabilities to the vendors strength. This post explains the Used-Defined Functions API (UDF) of the Dql Parser and shows some examples to give you some hints how you would extend DQL.

There are three types of functions in DQL, those that return a numerical value, those that return a string and those that return a Date. Your custom method has to be registered as either one of those. The return type information is used by the DQL parser to check possible syntax errors during the parsing process, for example using a string function return value in a math expression.

3.3.1 Registering your own DQL functions

You can register your functions adding them to the ORM configuration:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($nombre, $class);
$config->addCustomNumericFunction($nombre, $class);
$config->addCustomDatetimeFunction($nombre, $class);

$em = EntityManager::create($dbParams, $config);
```

The `$name` is the name the function will be referred to in the DQL query. `$class` is a string of a class-name which has to extend `Doctrine\ORM\Query\Node\FunctionNode`. This is a class that offers all the necessary API and methods to implement a UDF.

In this post we will implement some MySql specific Date calculation methods, which are quite handy in my opinion:

3.3.2 Date Diff

Mysql's DateDiff function takes two dates as argument and calculates the difference in days with `date1-date2`.

The DQL parser is a top-down recursive descent parser to generate the Abstract-Syntax Tree (AST) and uses a Tree-Walker approach to generate the appropriate SQL from the AST. This makes reading the Parser/TreeWalker code manageable in a finite amount of time.

The `FunctionNode` class I referred to earlier requires you to implement two methods, one for the parsing process (obviously) called `parse` and one for the TreeWalker process called `getSql()`. I show you the code for the DateDiff method and discuss it step by step:

```
<?php
/**
 * DateDiffFunction ::= "DATEDIFF" "(" ArithmeticPrimary "," ArithmeticPrimary ")"
 */
class DateDiff extends FunctionNode
{
    // (1)
    public $firstDateExpression = null;
    public $secondDateExpression = null;

    public function parse(\Doctrine\ORM\Query\Parser $parser)
    {
        $parser->match(Lexer::T_IDENTIFIER); // (2)
```

```
        $parser->match(Lexer::T_OPEN_PARENTHESIS); // (3)
        $this->firstDateExpression = $parser->ArithmeticPrimary(); // (4)
        $parser->match(Lexer::T_COMMA); // (5)
        $this->secondDateExpression = $parser->ArithmeticPrimary(); // (6)
        $parser->match(Lexer::T_CLOSE_PARENTHESIS); // (3)
    }

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'DATEDIFF(' .
            $this->firstDateExpression->dispatch($sqlWalker) . ', ' .
            $this->secondDateExpression->dispatch($sqlWalker) .
        ')'; // (7)
    }
}
```

The Parsing process of the DATEDIFF function is going to find two expressions the date1 and the date2 values, whose AST Node representations will be saved in the variables of the DateDiff FunctionNode instance at (1).

The parse() method has to cut the function call "DATEDIFF" and its argument into pieces. Since the parser detects the function using a lookahead the T_IDENTIFIER of the function name has to be taken from the stack (2), followed by a detection of the arguments in (4)-(6). The opening and closing parenthesis have to be detected also. This happens during the Parsing process and leads to the generation of a DateDiff FunctionNode somewhere in the AST of the dql statement.

The `ArithmeticPrimary` method call is the most common denominator of valid EBNF tokens taken from the DQL EBNF grammar that matches our requirements for valid input into the DateDiff Dql function. Picking the right tokens for your methods is a tricky business, but the EBNF grammar is pretty helpful finding it, as is looking at the Parser source code.

Now in the TreeWalker process we have to pick up this node and generate SQL from it, which apparently is quite easy looking at the code in (7). Since we don't know which type of AST Node the first and second Date expression are we are just dispatching them back to the SQL Walker to generate SQL from and then wrap our DATEDIFF function call around this output.

Now registering this DateDiff FunctionNode with the ORM using:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction('DATEDIFF', 'DoctrineExtensions\Query\MySql\DateDiff');
```

We can do fancy stuff like:

```
SELECT p FROM DoctrineExtensions\Query\BlogPost p WHERE DATEDIFF(CURRENT_TIME(), p.created) < 7
```

3.3.3 Date Add

Often useful it the ability to do some simple date calculations in your DQL query using MySql's DATE_ADD function.

I'll skip the blah and show the code for this function:

```
<?php
/**
 * DateAddFunction ::=
 *      "DATE_ADD" "(" ArithmeticPrimary ", INTERVAL" ArithmeticPrimary Identifier ")"
 */
class DateAdd extends FunctionNode
{
```

```
    public $firstDateExpression = null;
    public $intervalExpression = null;
    public $unit = null;

    public function parse(\Doctrine\ORM\Query\Parser $parser)
    {
        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);

        $this->firstDateExpression = $parser->ArithmeticPrimary();

        $parser->match(Lexer::T_COMMA);
        $parser->match(Lexer::T_IDENTIFIER);

        $this->intervalExpression = $parser->ArithmeticPrimary();

        $parser->match(Lexer::T_IDENTIFIER);

        /* @var $lexer Lexer */
        $lexer = $parser->getLexer();
        $this->unit = $lexer->token['value'];

        $parser->match(Lexer::T_CLOSE_PARENTHESIS);
    }

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'DATE_ADD(' .
            $this->firstDateExpression->dispatch($sqlWalker) . ', INTERVAL ' .
            $this->intervalExpression->dispatch($sqlWalker) . ' ' . $this->unit .
        ')';
    }
}
```

The only difference compared to the DATEDIFF here is, we additionally need the `Lexer` to access the value of the `T_IDENTIFIER` token for the Date Interval unit, for example the MONTH in:

```
SELECT p FROM DoctrineExtensions\Query\BlogPost p WHERE DATE_ADD(CURRENT_TIME(), INTERVAL 4 MONTH) >
```

The above method now only supports the specification using `INTERVAL`, to also allow a real date in DATE_ADD we need to add some decision logic to the parsing process (makes up for a nice exercise).

Now as you see, the Parsing process doesn't catch all the possible SQL errors, here we don't match for all the valid inputs for the interval unit. However where necessary we rely on the database vendors SQL parser to show us further errors in the parsing process, for example if the Unit would not be one of the supported values by MySql.

3.3.4 Conclusión

Now that you all know how you can implement vendor specific SQL functionalities in DQL, we would be excited to see user extensions that add vendor specific function packages, for example more math functions, XML + GIS Support, Hashing functions and so on.

For 2.0 we will come with the current set of functions, however for a future version we will re-evaluate if we can abstract even more vendor sql functions and extend the DQL languages scope.

Code for this Extension to DQL and other Doctrine Extensions can be found in my Github DoctrineExtensions repository.

3.4 Implementing ArrayAccess for Domain Objects

Autor de la sección: Roman Borschel (roman@code-factory.org)

This recipe will show you how to implement ArrayAccess for your domain objects in order to allow more uniform access, for example in templates. In these examples we will implement ArrayAccess on a Layer Supertype for all our domain objects.

3.4.1 Option 1

In this implementation we will make use of PHPs highly dynamic nature to dynamically access properties of a subtype in a supertype at runtime. Note that this implementation has 2 main caveats:

- It will not work with private fields
- It will not go through any getters/setters

```
<?php
abstract class DomainObject implements ArrayAccess
{
    public function offsetExists($offset) {
        return isset($this->$offset);
    }

    public function offsetSet($offset, $value) {
        $this->$offset = $value;
    }

    public function offsetGet($offset) {
        return $this->$offset;
    }

    public function offsetUnset($offset) {
        $this->$offset = null;
    }
}
```

3.4.2 Option 2

In this implementation we will dynamically invoke getters/setters. Again we use PHPs dynamic nature to invoke methods on a subtype from a supertype at runtime. This implementation has the following caveats:

- It relies on a naming convention
- The semantics of offsetExists can differ
- offsetUnset will not work with typehinted setters

```
<?php
abstract class DomainObject implements ArrayAccess
{
    public function offsetExists($offset) {
        // In this example we say that exists means it is not null
        $value = $this->{"get$offset"}();
        return $value !== null;
    }
```

```
    public function offsetSet($offset, $value) {
        $this->{"set$offset"}($value);
    }

    public function offsetGet($offset) {
        return $this->{"get$offset"}();
    }

    public function offsetUnset($offset) {
        $this->{"set$offset"}(null);
    }
}
```

3.4.3 Read-only

You can slightly tweak option 1 or option 2 in order to make array access read-only. This will also circumvent some of the caveats of each option. Simply make offsetSet and offsetUnset throw an exception (i.e. BadMethodCallException).

```
<?php
abstract class DomainObject implements ArrayAccess
{
    public function offsetExists($offset) {
        // option 1 or option 2
    }

    public function offsetSet($offset, $value) {
        throw new BadMethodCallException("Array access of class " . get_class($this) . " is read-only
    }

    public function offsetGet($offset) {
        // option 1 or option 2
    }

    public function offsetUnset($offset) {
        throw new BadMethodCallException("Array access of class " . get_class($this) . " is read-only
    }
}
```

3.5 Implementing the Notify ChangeTracking Policy

Autor de la sección: Roman Borschel (roman@code-factory.org)

The NOTIFY change-tracking policy is the most effective change-tracking policy provided by Doctrine but it requires some boilerplate code. This recipe will show you how this boilerplate code should look like. We will implement it on a Layer Supertype for all our domain objects.

3.5.1 Implementing NotifyPropertyChanged

The NOTIFY policy is based on the assumption that the entities notify interested listeners of changes to their properties. For that purpose, a class that wants to use this policy needs to implement the `NotifyPropertyChanged` interface from the `Doctrine\Common` namespace.

```php
<?php
use Doctrine\Common\NotifyPropertyChanged,
    Doctrine\Common\PropertyChangedListener;

abstract class DomainObject implements NotifyPropertyChanged
{
    private $_listeners = array();

    public function addPropertyChangedListener(PropertyChangedListener $listener) {
        $this->_listeners[] = $listener;
    }

    /** Notifies listeners of a change. */
    protected function _onPropertyChanged($propName, $oldValue, $newValue) {
        if ($this->_listeners) {
            foreach ($this->_listeners as $listener) {
                $listener->propertyChanged($this, $propName, $oldValue, $newValue);
            }
        }
    }
}
```

Then, in each property setter of concrete, derived domain classes, you need to invoke _onPropertyChanged as follows to notify listeners:

```php
<?php
// Mapping not shown, either in annotations, xml or yaml as usual
class MyEntity extends DomainObject
{
    private $data;
    // ... other fields as usual

    public function setData($data) {
        if ($data != $this->data) { // check: is it actually modified?
            $this->_onPropertyChanged('data', $this->data, $data);
            $this->data = $data;
        }
    }
}
```

The check whether the new value is different from the old one is not mandatory but recommended. That way you can avoid unnecessary updates and also have full control over when you consider a property changed.

3.6 Implementing Wakeup or Clone

Autor de la sección: Roman Borschel (roman@code-factory.org)

As explained in the restrictions for entity classes in the manual, it is usually not allowed for an entity to implement `__wakeup` or `__clone`, because Doctrine makes special use of them. However, it is quite easy to make use of these methods in a safe way by guarding the custom wakeup or clone code with an entity identity check, as demonstrated in the following sections.

3.6.1 Safely implementing __wakeup

To safely implement `__wakeup`, simply enclose your implementation code in an identity check as follows:

```php
<?php
class MyEntity
{
    private $id; // This is the identifier of the entity.
    //...

    public function __wakeup()
    {
        // If the entity has an identity, proceed as normal.
        if ($this->id) {
            // ... Your code here as normal ...
        }
        // otherwise do nothing, do NOT throw an exception!
    }

    //...
}
```

3.6.2 Safely implementing __clone

Safely implementing `__clone` is pretty much the same:

```php
<?php
class MyEntity
{
    private $id; // This is the identifier of the entity.
    //...

    public function __clone()
    {
        // If the entity has an identity, proceed as normal.
        if ($this->id) {
            // ... Your code here as normal ...
        }
        // otherwise do nothing, do NOT throw an exception!
    }

    //...
}
```

3.6.3 Resumen

As you have seen, it is quite easy to safely make use of `__wakeup` and `__clone` in your entities without adding any really Doctrine-specific or Doctrine-dependant code.

These implementations are possible and safe because when Doctrine invokes these methods, the entities never have an identity (yet). Furthermore, it is possibly a good idea to check for the identity in your code anyway, since it's rarely the case that you want to unserialize or clone an entity with no identity.

3.7 Integrating with CodeIgniter

This is recipe for using Doctrine 2 in your CodeIgniter framework.

Nota: This might not work for all CodeIgniter versions and may require slight adjustments.

Here is how to set it up:

Make a CodeIgniter library that is both a wrapper and a bootstrap for Doctrine 2.

3.7.1 Setting up the file structure

Here are the steps:

- Add a php file to your system/application/libraries folder called Doctrine.php. This is going to be your wrapper/bootstrap for the D2 entity manager.
- Put the Doctrine folder (the one that contains Common, DBAL, and ORM) inside that same libraries folder.
- Your system/application/libraries folder now looks like this:

  system/applications/libraries -Doctrine -Doctrine.php -index.html
- If you want, open your config/autoload.php file and autoload your Doctrine library.

  <?php $autoload['libraries'] = array('doctrine');

3.7.2 Creating your Doctrine CodeIgniter library

Now, here is what your Doctrine.php file should look like. Customize it to your needs.

```
<?php
use Doctrine\Common\ClassLoader,
    Doctrine\ORM\Configuration,
    Doctrine\ORM\EntityManager,
    Doctrine\Common\Cache\ArrayCache,
    Doctrine\DBAL\Logging\EchoSQLLogger;

class Doctrine {

  public $em = null;

  public function __construct()
  {
    // load database configuration from CodeIgniter
    require_once APPPATH.'config/database.php';

    // Set up class loading. You could use different autoloaders, provided by your favorite framework
    // if you want to.
    require_once APPPATH.'libraries/Doctrine/Common/ClassLoader.php';

    $doctrineClassLoader = new ClassLoader('Doctrine',  APPPATH.'libraries');
    $doctrineClassLoader->register();
    $entitiesClassLoader = new ClassLoader('models', rtrim(APPPATH, "/" ));
    $entitiesClassLoader->register();
    $proxiesClassLoader = new ClassLoader('Proxies', APPPATH.'models/proxies');
    $proxiesClassLoader->register();

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

```
    $driverImpl = $config->newDefaultAnnotationDriver(array(APPPATH.'models/Entities'));
    $config->setMetadataDriverImpl($driverImpl);
    $config->setQueryCacheImpl($cache);

    $config->setQueryCacheImpl($cache);

    // Proxy configuration
    $config->setProxyDir(APPPATH.'/models/proxies');
    $config->setProxyNamespace('Proxies');

    // Set up logger
    $logger = new EchoSQLLogger;
    $config->setSQLLogger($logger);

    $config->setAutoGenerateProxyClasses( TRUE );

    // Database connection information
    $connectionOptions = array(
        'driver' => 'pdo_mysql',
        'user' =>     $db['default']['username'],
        'password' => $db['default']['password'],
        'host' =>     $db['default']['hostname'],
        'dbname' =>   $db['default']['database']
    );

    // Create EntityManager
    $this->em = EntityManager::create($connectionOptions, $config);
  }
}
```

Please note that this is a development configuration; for a production system you'll want to use a real caching system like APC, get rid of EchoSqlLogger, and turn off autoGenerateProxyClasses.

For more details, consult the Doctrine 2 Configuration documentation.

3.7.3 Now to use it

Whenever you need a reference to the entity manager inside one of your controllers, views, or models you can do this:

```
<?php
$em = $this->doctrine->em;
```

Eso es todo lo que hay que hacer. Once you get the reference to your EntityManager do your Doctrine 2.0 voodoo as normal.

Note: If you do not choose to autoload the Doctrine library, you will need to put this line before you get a reference to it:

```
<?php
$this->load->library('doctrine');
```

Good luck!

3.8 SQL-Table Prefixes

This recipe is intended as an example of implementing a loadClassMetadata listener to provide a Table Prefix option for your application. The method used below is not a hack, but fully integrates into the Doctrine system, all SQL generated will include the appropriate table prefix.

In most circumstances it is desirable to separate different applications into individual databases, but in certain cases, it may be beneficial to have a table prefix for your Entities to separate them from other vendor products in the same database.

3.8.1 Implementing the listener

The listener in this example has been set up with the DoctrineExtensions namespace. You create this file in your library/DoctrineExtensions directory, but will need to set up appropriate autoloaders.

```
<?php

namespace DoctrineExtensions;
use \Doctrine\ORM\Event\LoadClassMetadataEventArgs;

class TablePrefix
{
    protected $prefix = '';

    public function __construct($prefix)
    {
        $this->prefix = (string) $prefix;
    }

    public function loadClassMetadata(LoadClassMetadataEventArgs $eventArgs)
    {
        $classMetadata = $eventArgs->getClassMetadata();
        $classMetadata->setTableName($this->prefix . $classMetadata->getTableName());
        foreach ($classMetadata->getAssociationMappings() as $fieldName => $mapping) {
            if ($mapping['type'] == \Doctrine\ORM\Mapping\ClassMetadataInfo::MANY_TO_MANY) {
                $mappedTableName = $classMetadata->associationMappings[$fieldName]['joinTable']['name
                $classMetadata->associationMappings[$fieldName]['joinTable']['name'] = $this->prefix
            }
        }
    }

}
```

3.8.2 Telling the EntityManager about our listener

A listener of this type must be set up before the EntityManager has been initialised, otherwise an Entity might be created or cached before the prefix has been set.

Nota: If you set this listener up, be aware that you will need to clear your caches and drop then recreate your database schema.

```
<?php

// $connectionOptions and $config set earlier
```

```
$evm = new \Doctrine\Common\EventManager;

// Table Prefix
$tablePrefix = new \DoctrineExtensions\TablePrefix('prefix_');
$evm->addEventListener(\Doctrine\ORM\Events::loadClassMetadata, $tablePrefix);

$em = \Doctrine\ORM\EntityManager::create($connectionOptions, $config, $evm);
```

3.9 Strategy-Pattern

This recipe will give you a short introduction on how to design similar entities without using expensive (i.e. slow) inheritance but with not more than * the well-known strategy pattern * event listeners

3.9.1 Scenario / Problem

Given a Content-Management-System, we probably want to add / edit some so-called "blocks" and "panels". What are they for?

- A block might be a registration form, some text content, a table with information. A good example might also be a small calendar.
- A panel is by definition a block that can itself contain blocks. A good example for a panel might be a sidebar box: You could easily add a small calendar into it.

So, in this scenario, when building your CMS, you will surely add lots of blocks and panels to your pages and you will find yourself highly uncomfortable because of the following:

- Every existing page needs to know about the panels it contains - therefore, you'll have an association to your panels. But if you've got several types of panels - what do you do? Add an association to every panel-type? This wouldn't be flexible. You might be tempted to add an AbstractPanelEntity and an AbstractBlockEntity that use class inheritance. Your page could then only confer to the AbstractPanelType and Doctrine 2 would do the rest for you, i.e. load the right entities. But - you'll for sure have lots of panels and blocks, and even worse, you'd have to edit the discriminator map *manually* every time you or another developer implements a new block / entity. This would tear down any effort of modular programming.

Therefore, we need something thats far more flexible.

3.9.2 Solution

The solution itself is pretty easy. We will have one base class that will be loaded via the page and that has specific behaviour - a Block class might render the front-end and even the backend, for example. Now, every block that you'll write might look different or need different data - therefore, we'll offer an API to these methods but internally, we use a strategy that exactly knows what to do.

First of all, we need to make sure that we have an interface that contains every needed action. Such actions would be rendering the front-end or the backend, solving dependencies (blocks that are supposed to be placed in the sidebar could refuse to be placed in the middle of your page, for example).

Such an interface could look like this:

```
<?php
/**
 * This interface defines the basic actions that a block / panel needs to support.
 *
```

```
 * Every blockstrategy is *only* responsible for rendering a block and declaring some basic
 * support, but *not* for updating its configuration etc. For this purpose, use controllers
 * and models.
 */
interface BlockStrategyInterface {
    /**
     * This could configure your entity
     */
    public function setConfig(Config\EntityConfig $config);

    /**
     * Returns the config this strategy is configured with.
     * @return Core\Model\Config\EntityConfig
     */
    public function getConfig();

    /**
     * Set the view object.
     * @param  \Zend_View_Interface $view
     * @return \Zend_View_Helper_Interface
     */
    public function setView(\Zend_View_Interface $view);

    /**
     * @return \Zend_View_Interface
     */
    public function getView();

    /**
     * Renders this strategy. This method will be called when the user
     * displays the site.
     *
     * @return string
     */
    public function renderFrontend();

    /**
     * Renders the backend of this block. This method will be called when
     * a user tries to reconfigure this block instance.
     *
     * Most of the time, this method will return / output a simple form which in turn
     * calls some controllers.
     *
     * @return string
     */
    public function renderBackend();

    /**
     * Returns all possible types of panels this block can be stacked onto
     *
     * @return array
     */
    public function getRequiredPanelTypes();

    /**
     * Determines whether a Block is able to use a given type or not
     * @param string $typeName The typename
     * @return boolean
```

```
     */
    public function canUsePanelType($typeName);

    public function setBlockEntity(AbstractBlock $block);

    public function getBlockEntity();
}
```

As you can see, we have a method "setBlockEntity" which ties a potential strategy to an object of type AbstractBlock. This type will simply define the basic behaviour of our blocks and could potentially look something like this:

```
<?php
/**
 * This is the base class for both Panels and Blocks.
 * It shouldn't be extended by your own blocks - simply write a strategy!
 */
abstract class AbstractBlock {
    /**
     * The id of the block item instance
     * This is a doctrine field, so you need to setup generation for it
     * @var integer
     */
    private $id;

    // Add code for relation to the parent panel, configuration objects, ....

    /**
     * This var contains the classname of the strategy
     * that is used for this blockitem. (This string (!) value will be persisted by Doctrine 2)
     *
     * This is a doctrine field, so make sure that you use an @column annotation or setup your
     * yaml or xml files correctly
     * @var string
     */
    protected $strategyClassName;

    /**
     * This var contains an instance of $this->blockStrategy. Will not be persisted by Doctrine 2.
     *
     * @var BlockStrategyInterface
     */
    protected $strategyInstance;

    /**
     * Returns the strategy that is used for this blockitem.
     *
     * The strategy itself defines how this block can be rendered etc.
     *
     * @return string
     */
    public function getStrategyClassName() {
        return $this->strategyClassName;
    }

    /**
     * Returns the instantiated strategy
     *
     * @return BlockStrategyInterface
```

```
     */
    public function getStrategyInstance() {
        return $this->strategyInstance;
    }

    /**
     * Sets the strategy this block / panel should work as. Make sure that you've used
     * this method before persisting the block!
     *
     * @param BlockStrategyInterface $strategy
     */
    public function setStrategy(BlockStrategyInterface $strategy) {
        $this->strategyInstance  = $strategy;
        $this->strategyClassName = get_class($strategy);
        $strategy->setBlockEntity($this);
    }
```

Now, the important point is that $strategyClassName is a Doctrine 2 field, i.e. Doctrine will persist this value. This is only the class name of your strategy and not an instance!

Finishing your strategy pattern, we hook into the Doctrine postLoad event and check whether a block has been loaded. If so, you will initialize it - i.e. get the strategies classname, create an instance of it and set it via setStrategyBlock().

This might look like this:

```
<?php
use \Doctrine\ORM,
    \Doctrine\Common;

/**
 * The BlockStrategyEventListener will initialize a strategy after the
 * block itself was loaded.
 */
class BlockStrategyEventListener implements Common\EventSubscriber {

    protected $view;

    public function __construct(\Zend_View_Interface $view) {
        $this->view = $view;
    }

    public function getSubscribedEvents() {
       return array(ORM\Events::postLoad);
    }

    public function postLoad(ORM\Event\LifecycleEventArgs $args) {
        $blockItem = $args->getEntity();

        // Both blocks and panels are instances of Block\AbstractBlock
        if ($blockItem instanceof Block\AbstractBlock) {
            $strategy  = $blockItem->getStrategyClassName();
            $strategyInstance = new $strategy();
            if (null !== $blockItem->getConfig()) {
                $strategyInstance->setConfig($blockItem->getConfig());
            }
            $strategyInstance->setView($this->view);
            $blockItem->setStrategy($strategyInstance);
        }
    }
```

```
}
```

In this example, even some variables are set - like a view object or a specific configuration object.

3.10 Validation of Entities

Autor de la sección: Benjamin Eberlei <kontakt@beberlei.de>

Doctrine 2 does not ship with any internal validators, the reason being that we think all the frameworks out there already ship with quite decent ones that can be integrated into your Domain easily. What we offer are hooks to execute any kind of validation.

Nota: You don't need to validate your entities in the lifecycle events. Its only one of many options. Of course you can also perform validations in value setters or any other method of your entities that are used in your code.

Entities can register lifecycle event methods with Doctrine that are called on different occasions. For validation we would need to hook into the events called before persisting and updating. Even though we don't support validation out of the box, the implementation is even simpler than in Doctrine 1 and you will get the additional benefit of being able to re-use your validation in any other part of your domain.

Say we have an `Order` with several `OrderLine` instances. We never want to allow any customer to order for a larger sum than he is allowed to:

```
<?php
class Order
{
    public function assertCustomerAllowedBuying()
    {
        $orderLimit = $this->customer->getOrderLimit();

        $amount = 0;
        foreach ($this->orderLines AS $line) {
            $amount += $line->getAmount();
        }

        if ($amount > $orderLimit) {
            throw new CustomerOrderLimitExceededException();
        }
    }
}
```

Now this is some pretty important piece of business logic in your code, enforcing it at any time is important so that customers with a unknown reputation don't owe your business too much money.

We can enforce this constraint in any of the metadata drivers. First Annotations:

```
<?php
/**
 * @Entity
 * @HasLifecycleCallbacks
 */
class Order
{
    /**
     * @PrePersist @PreUpdate
     */
```

```
    public function assertCustomerAllowedBuying() {}
}
```

In XML Mappings:

```
<doctrine-mapping>
    <entity name="Order">
        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist" method="assertCustomerallowedBuying" />
            <lifecycle-callback type="preUpdate" method="assertCustomerallowedBuying" />
        </lifecycle-callbacks>
    </entity>
</doctirne-mapping>
```

YAML needs some little change yet, to allow multiple lifecycle events for one method, this will happen before Beta 1 though.

Now validation is performed whenever you call `EntityManager#persist($order)` or when you call `EntityManager#flush()` and an order is about to be updated. Any Exception that happens in the lifecycle callbacks will be cached by the EntityManager and the current transaction is rolled back.

Of course you can do any type of primitive checks, not null, email-validation, string size, integer and date ranges in your validation callbacks.

```
<?php
class Order
{
    /**
     * @PrePersist @PreUpdate
     */
    public function validate()
    {
        if (!($this->plannedShipDate instanceof DateTime)) {
            throw new ValidateException();
        }

        if ($this->plannedShipDate->format('U') < time()) {
            throw new ValidateException();
        }

        if ($this->customer == null) {
            throw new OrderRequiresCustomerException();
        }
    }
}
```

What is nice about lifecycle events is, you can also re-use the methods at other places in your domain, for example in combination with your form library. Additionally there is no limitation in the number of methods you register on one particular event, i.e. you can register multiple methods for validation in "PrePersist" or "PreUpdate" or mix and share them in any combinations between those two events.

There is no limit to what you can and can't validate in "PrePersist" and "PreUpdate" as long as you don't create new entity instances. This was already discussed in the previous blog post on the Versionable extension, which requires another type of event called "onFlush".

Further readings: *Lifecycle Events*

3.11 Working with DateTime Instances

There are many nitty gritty details when working with PHPs DateTime instances. You have know their inner workings pretty well not to make mistakes with date handling. This cookbook entry holds several interesting pieces of information on how to work with PHP DateTime instances in Doctrine 2.

3.11.1 DateTime changes are detected by Reference

When calling `EntityManager#flush()` Doctrine computes the changesets of all the currently managed entities and saves the differences to the database. In case of object properties (@Column(type="datetime") or @Column(type="object")) these comparisons are always made **BY REFERENCE**. That means the following change will **NOT** be saved into the database:

```
<?php
/** @Entity */
class Article
{
    /** @Column(type="datetime") */
    private $updated;

    public function setUpdated()
    {
        // will NOT be saved in the database
        $this->updated->modify("now");
    }
}
```

The way to go would be:

```
<?php
class Article
{
    public function setUpdated()
    {
        // WILL be saved in the database
        $this->updated = new \DateTime("now");
    }
}
```

3.11.2 Default Timezone Gotcha

By default Doctrine assumes that you are working with a default timezone. Each DateTime instance that is created by Doctrine will be assigned the timezone that is currently the default, either through the `date.timezone` ini setting or by calling `date_default_timezone_set()`.

This is very important to handle correctly if your application runs on different serves or is moved from one to another server (with different timezone settings). You have to make sure that the timezone is the correct one on all this systems.

3.11.3 Handling different Timezones with the DateTime Type

If you first come across the requirement to save different you are still optimistic to manage this mess, however let me crush your expectations fast. There is not a single database out there (supported by Doctrine 2) that supports timezones correctly. Correctly here means that you can cover all the use-cases that can come up with timezones. If you don't believe me you should read up on Storing DateTime in Databases.

The problem is simple. Not a single database vendor saves the timezone, only the differences to UTC. However with frequent daylight saving and political timezone changes you can have a UTC offset that moves in different offset directions depending on the real location.

The solution for this dialemma is simple. Don't use timezones with DateTime and Doctrine 2. However there is a workaround that even allows correct date-time handling with timezones:

1. Always convert any DateTime instance to UTC.
2. Only set Timezones for displaying purposes
3. Save the Timezone in the Entity for persistence.

Say we have an application for an international postal company and employees insert events regarding postal-package around the world, in their current timezones. To determine the exact time an event occoured means to save both the UTC time at the time of the booking and the timezone the event happend in.

```
<?php

namespace DoctrineExtensions\DBAL\Types;

use Doctrine\DBAL\Platforms\AbtractPlatform;
use Doctrine\DBAL\Types\ConversionException;

class UTCDateTimeType extends DateTimeType
{
    static private $utc = null;

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if ($value === null) {
            return null;
        }


        return $value->format($platform->getDateTimeFormatString(),
            (self::$utc) ? self::$utc : (self::$utc = new \DateTimeZone(\DateTimeZone::UTC))
        );
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        if ($value === null) {
            return null;
        }

        $val = \DateTime::createFromFormat(
            $platform->getDateTimeFormatString(),
            $value,
            (self::$utc) ? self::$utc : (self::$utc = new \DateTimeZone(\DateTimeZone::UTC))
        );
        if (!$val) {
            throw ConversionException::conversionFailed($value, $this->getName());
        }
        return $val;
    }
}
```

This database type makes sure that every DateTime instance is always saved in UTC, relative to the current timezone that the passed DateTime instance has. To be able to transform these values back into their real timezone you have to

save the timezone in a seperate field of the entity requiring timezoned datetimes:

```
<?php
namespace Shipping;

/**
 * @Entity
 */
class Event
{
    /** @Column(type="datetime") */
    private $created;

    /** @Column(type="string") */
    private $timezone;

    /**
     * @var bool
     */
    private $localized = false;

    public function __construct(\DateTime $createDate)
    {
        $this->localized = true;
        $this->created = $createDate;
        $this->timezone = $createDate->getTimeZone()->getName();
    }

    public function getCreated()
    {
        if (!$this->localized) {
            $this->created->setTimeZone(new \DateTimeZone($this->timezone));
        }
        return $this->created;
    }
}
```

This snippet makes use of the previously discussed "changeset by reference only" property of objects. That means a new DateTime will only be used during updating if the reference changes between retrieval and flush operation. This means we can easily go and modify the instance by setting the previous local timezone.

3.12 Mysql Enums

The type system of Doctrine 2 consists of flyweights, which means there is only one instance of any given type. Additionally types do not contain state. Both assumptions make it rather complicated to work with the Enum Type of MySQL that is used quite a lot by developers.

When using Enums with a non-tweaked Doctrine 2 application you will get errors from the Schema-Tool commands due to the unknown database type "enum". By default Doctrine does not map the MySQL enum type to a Doctrine type. This is because Enums contain state (their allowed values) and Doctrine types don't.

This cookbook entry shows two possible solutions to work with MySQL enums. But first a word of warning. The MySQL Enum type has considerable downsides:

- Adding new values requires to rebuild the whole table, which can take hours depending on the size.
- Enums are ordered in the way the values are specified, not in their "natural" order.

- Enums validation mechanism for allowed values is not necessarily good, specifying invalid values leads to an empty enum for the default MySQL error settings. You can easily replicate the "allow only some values" requirement in your Doctrine entities.

3.12.1 Solution 1: Mapping to Varchars

You can map ENUMs to varchars. You can register MySQL ENUMs to map to Doctrine varchars. This way Doctrine always resolves ENUMs to Doctrine varchars. It will even detect this match correctly when using SchemaTool update commands.

```
<?php
$conn = $em->getConnection();
$conn->getDatabasePlatform()->registerDoctrineTypeMapping('enum', 'string');
```

In this case you have to ensure that each varchar field that is an enum in the database only gets passed the allowed values. You can easily enforce this in your entities:

```
<?php
/** @Entity */
class Article
{
    const STATUS_VISIBLE = 'visible';
    const STATUS_INVISIBLE = 'invisible';

    /** @Column(type="varchar") */
    private $status;

    public function setStatus($status)
    {
        if (!in_array($status, array(self::STATUS_VISIBLE, self::STATUS_INVISIBLE))) {
            throw new \InvalidArgumentException("Invalid status");
        }
        $this->status = $status;
    }
}
```

If you want to actively create enums through the Doctrine Schema-Tool by using the **columnDefinition** attribute.

```
<?php
/** @Entity */
class Article
{
    /** @Column(type="varchar", columnDefinition="ENUM('visible', 'invisible')") */
    private $status;
}
```

In this case however Schema-Tool update will have a hard time not to request changes for this column on each call.

3.12.2 Solution 2: Defining a Type

You can make a stateless ENUM type by creating a type class for each unique set of ENUM values. For example for the previous enum type:

```
<?php
namespace MyProject\DBAL;

use Doctrine\DBAL\Types\Type;
```

```
use Doctrine\DBAL\Platforms\AbstractPlatform;

class EnumVisibilityType extends Type
{
    const ENUM_VISIBILITY = 'enumvisibility';
    const STATUS_VISIBLE = 'visible';
    const STATUS_INVISIBLE = 'invisible';

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        return "ENUM('visible', 'invisible') COMMENT '(DC2Type:enumvisibility)'";
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return $value;
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if (!in_array($value, array(self::STATUS_VISIBLE, self::STATUS_INVISIBLE))) {
            throw new \InvalidArgumentException("Invalid status");
        }
        return $value;
    }

    public function getName()
    {
        return self::ENUM_VISIBILITY;
    }
}
```

You can register this type with `Type::addType('enumvisibility', 'MyProject\DBAL\EnumVisibilityType');`. Then in your entity you can just use this type:

```
<?php
/** @Entity */
class Article
{
    /** @Column(type="enumvisibility") */
    private $status;
}
```

You can generalize this approach easily to create a base class for enums:

```
<?php
namespace MyProject\DBAL;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

abstract class EnumType extends Type
{
    protected $name;
    protected $values = array();

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        $values = array_map(function($val) { return "'".$val."'"; }, $this->values);
```

```
        return "ENUM(".implode(", ", $values).") COMMENT '(DC2Type:".$this->name.")'";
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return $value;
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if (!in_array($value, $this->values)) {
            throw new \InvalidArgumentException("Invalid '".$this->name."' value.");
        }
        return $value;
    }

    public function getNombre()
    {
        return $this->name;
    }
}
```

With this base class you can define an enum as easily as:

```
<?php
namespace MyProject\DBAL;

class EnumVisibilityType extends EnumType
{
    protected $name = 'enumvisibility';
    protected $values = array('visible', 'invisible');
}
```