Doctrine2. Справочное руководство (Reference Guide)

В переводе: Alexey Plotnik

Содержание

О переводе

Автор перевода: Alexey Plotnik[1]

Ссылка на авторский перевод[2]

Ссылка на оригинальный 'Reference Guide' по последней версии doctrine (english)[3]

Редактор: Роман Бысов[4]

ПРИМЕЧАНИЯ ПЕРЕВОДЧИКА:
Когда я работал над переводом, в тексте основных глав документации по Doctrine мне неоднократно встречались термины, значение которых понятны лишь в контексте ORM, трактовать их можно по-разному. При прямом переводе без пояснений их смысл может быть утерян. Поэтому в разделе Глоссарий я просто своими словами опишу, что по моему мнению они означают, чтобы вам было понятно о чем будет идти речь и не возникло проблем при чтении руководства.

ПРИМЕЧАНИЕ РЕДАКТОРА:
Переведена примерно половина руководства, предположительно версии 2.0..2.2 (на сайте предыдущих версий нет, на гитхабе дока появилась с 2.4 и уже расходится с переводом).
Данная копипаста создана с целью выверки технологии для генерации pdf ebook из web-статей.
Основная проблема, это генерация линков в PDF из межстраничных ссылок. Из 'коробки' более-менее приемлимые результаты дало использование AsciidocFX (docbook), который выполняет основную работу по автоматическому переформатированию HTML в asciidoc и экспорту в xml и pdf/mobi/epub. Генерация ссылок между главами пока не взлетели. Возможно что то 'выгорит', если покопаться 'под капотом' Docbook.
Прошу прощения за выпил главовых TOC-ов. Я признаю только автоматическую генерацию, а нужного результата по стандарту (:toc:[]) мне пока добиться не удалось. Остановился на общем TOC-е, который генериться при экспорте в ebook.
Для генерации pdf/mobi/ebook/html/odt, проще всего использовать AsciidocFx. На выбор есть инсталяшка, которая сама поставит для себя нужный jdk. Gitprint.com,

1 https://github.com/odiszapc
2 http://odiszapc.ru/doctrine
3 http://docs.doctrine-project.org/en/latest/#reference-guide
4 https://github.com/bisoff

на момент моих испытаний, умел работать с asciidoc минимально (форматирование текста и наверно всё).
С markdown, как со стартовой альтернативой, совсем не задалось. Экспорт в pdf - танцы с бубном (tex), внутренние ссылки по файлам книги - не задалось. TOC-а нет. До reStructuredText и Sphinx уже не добрался так как 'подвзлетел' asciidoc. Кстати документация к doctrine написана на нём[5]. И генерация есть для linux[6]. Под виндой Sphinx не взлетел.

Содержание[7]

[5] https://github.com/doctrine/doctrine2/tree/master/docs/en/reference
[6] https://github.com/doctrine/doctrine2/tree/master/docs
[7] doctrine2-ru.asc#toc

Глоссарий

Annotations

Аннотации. Комментарии, несущие смысловые последствия в интерпретации кода.

Ass

Некоторые предложения по тексту глав вызывали именно такое определение, по этой причине я не смог их перевести, не обессудьте.

Association

Ассоциация. Взаимосвязь. Связь. Отношение. У сущности кроме простых полей могут быть поля, олицетворяющие ее взаимосвязь с другими сущностями. Что-то вроде отношений в базах данных (собственно это они и есть, но покрытые слоем абстракции ORM). Я предпочитаю называть это просто связью. Это соединения между сущностями.

Collection

Коллекция, список. Используется в контексте отношений “один ко многим” и “многие ко многим”. Похожи на массив, но конечно более навороченная концепция.

DQL

Язык запросов Doctrine. Чем-то похож на SQL. Не вижу смысла переводить аббревиатуру.

DiscriminatorColumn

Столбец дискриминатора. Столбец дискриминатора содержит значение, которое определяет, какому классу принадлежит каждая запись. Что это такое? Это пошло еще с Java Hibernate. Например, рассмотрим таблицу Persons, которая содержит данные всех сотрудников компании. Некоторые лица являются служащими, а некоторые — менеджерами. Таблица Persons содержит столбец с именем EmployeeType, который имеет значение 1 для менеджеров и значение 2 для служащих; это и есть столбец дискриминатора. В этом случае можно создать подкласс служащих и заполнить класс только записями, которые имеют в столбце EmployeeType значение 2. Кроме того, можно удалить из каждого класса неприменимые столбцы.

Entity, Persistent classes

Сущность, хранимый (персистентный) класс. По сути одно и тоже. Основная концепция Doctrine. Сущностью может быть одна таблица, либо таблица со связанными с ней другими таблицами. Да что там, сущностью может быть целая база данных! Чтобы класс считался хранимым (то есть обрабатывался Доктриной)

он должен быть помечен как сущность при помощи метаданных, например аннотацией к классу со свойством @Entity:

```
/** @Entity */
class Entity_MyItem
{
}
```

Metadata, Object-relational mapping metadata

Метаданные. Набор параметров, описывающих как объектная модель будет проецироваться на физическую модель в базе данных. Эти директивы превращают обычный класс в сущность. Вы берете обычный класс, добавляете к нему метаданные и все — вот она ваша сущность. Метаданные могут быть определены тремя strong textcпособами:

- XML
- YAML
- Аннотации DocBlock (комментарии в PHP-коде)

Hydration

Гидрация. Вменяемого перевода на русский нет. Процесс гидрации – это процедура преобразования результата SQL запроса в объектную модель. В этот момент создаются экземпляры сущностей, подключаются связи и так далее.

Inverse/owning side

Владеющая (прямая) и обратная стороны связи. Что это значит? К примеру, у вас есть сущность “Новость” и есть сущность “Комментарий”. У одной новости может быть несколько комментариев, поэтому комментарий содержит в себе ссылку на новость (физически в базе данных есть соответствующее поле, вроде new_id). Как вы видите, это классический пример отношения “один ко многим”, где “один” это новость, а “многие” это комментарии. Комментарии однозначно связаны с новостью, поэтому они — сторона владельца или прямая сторона. Это позволяет из сущности комментария перейти к сущности новости. Обратная сторона связи позволяет из новости получить указатель на ее комментарии.

Mapping

Отображение. Соответствия. Основная концепция любой ORM. Отображение объектной структуры на модель базы данных.

Metadata drivers

Драйвер метаданных. Управляют различными способами представления метаданных (XML, YAML, аннотации Dockblock).

Metadata Cache

Кеш метаданных. Чтобы каждый раз не парсить метаданные.

Orphan removal

Переводится как “удаление объектов-сирот”. Технология каскадного удаления объектов, на которые потеряны ссылки. Пример с новостями и комментариями: комментарий связан с новостью, при удалении новости сам комментарий перестает иметь смысл (конечно если вы не решите иначе). Так вот данный подход позволяет удалить его автоматически, если связь содержащая такой комментарий была соответствующим образом настроена.

Partial objects

“Обрезанные объекты” без некоторых внешних связей. Вы сами задаете какие поля объекта будут запрошены из базы. Иными словами, вы получите объект у которого будут отсутствовать некоторые поля. Partial Objects сделаны для повышения производительности в узких местах, но, честно говоря, реальный профит от них под большим вопросом, MySQL очень быстр.

Persistent properties

Хранимые свойства. Хранимые члены. Свойства класса сущности, непосредственно имеющие отношения к базе данных. Именно они и олицетворяют сущность как таковую. Такие свойства помечены соответствующими тегами в аннотациях к ним.

Proxy classes

Классы прокси. Прокси. Классы-заглушки. Используются для ускорения.

Query cache

Кеш запросов. Не то, что вы подумали. Он кеширует не результаты SQL запросов, а результат разбора запроса DQL.

Schema-Tool

Инструмент для работы со схемой данных. Он перестраивает ее, удаляет, создает. Переводить не нужно.

Введение

Содержание

1. Начало

Doctrine 2 представляет собой хороший пример механизма объектно-реляционного отображения (ORM) для *PHP 5.3+*, позволяющий работать с базой данных максимально прозрачно, где в качестве промежуточного слоя используются обычные объекты *PHP*. В качестве основы используется весьма мощный слой абстракции от базы данных (*DBAL*). Основная задача*ORM* — связать две концепции: объекты *PHP* и записи в реляционной базе данных.

Одна из ключевых особенностей Doctrine — возможность написания запросов на собственном объектно-ориентированном языке, чем-то напоминающим SQL, называемым *Doctrine Query Language (DQL)*, созданным под вдохновением от *Hibernates HQL*. Помимо небольших отличий от *SQL*, он позволяет значительно усилить степень абстракции между объектами и строками базы данных, что позволяет создавать мощные и гибкие запросы, при этом сохраняя целостность.

2. Достоверность информации

Данный документ представляет собой справочную документацию по *Doctrine 2*. Руководства для начинающих и обучалки, как это было для *Doctrine 1.x* будут доступны позднее.

3. Применение ORM

Как уже было сказано, задача *Doctrine 2* — упростить взаимодействие между строками в СУБД и объектной моделью *PHP*. Так что основной эффект от ее применения может быть достигнут лишь в объектно-ориентированной среде и приложениях, использующих парадигму ООП. В остальных случаях от *Doctrine 2* будет мало пользы.

4. Требования

Doctrine 2 требуется версия *PHP* не менее *5.3.0*. Для увеличения производительность рекомендуется использовать модуль *APC*для *PHP*.

5. Составляющие Doctrine 2

Doctrine 2 состоит из трех основных библиотек.

- *Common*
- *DBAL* (включает в себя *Common*)
- *ORM* (включает в себя как *DBAL*, так и *Common*)

Данное руководство в основном относится к пакету *ORM*, однако иногда будут затронуты и остальные два. Есть определенные причины, по которым *Doctrine* была разбита на несколько частей:

- Декомпозиция библиотеки дает большую гибкости
- Пакет *Common* можно использовать отдельно от пакетов *ORM* и *DBAL*
- *DBAL* можно использовать отдельно от *ORM*

5.1. Библиотека Common

Этот пакет содержит несколько компонентов, которые не имеют никаких внешних зависимостей, в т.ч. и от PHP. В *Common* в качестве корневого пространства имен используется **Doctrine\Common**.

5.2. Библиотека DBAL

DBAL содержит слой абстракции от БД, в качестве надстройки над библиотекой *PDO*, однако плотно на ней не завязана. Задача этого слоя — дать один общий API, который покрывает все различия между интерфейсами СУБД. В качестве корневого пространства имен для классов *DBAL* используется **Doctrine\DBAL**.

5.3. Библиотека ORM

Пакет *ORM* содержит в себе реализацию алгоритма объектно-реляционного отображения, реализуя прозрачные взаимосвязи и отношения для обычных *PHP* объектов. В *ORM* в качестве корневого пространства имен используется **Doctrine\ORM**.

6. Установка

Существует несколько способов установки Doctrine. Ниже будут описаны все возможные варианты, так выбирайте какой вам больше нравится.

6.1. PEAR

Все три части *Doctrine* можно легко установить из командной строки **PEAR**.

Установка только библиотеки **Common** осуществляется следующей командой:

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

DBAL устанавливается так:

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

А **ORM** так:

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

Тэг <*version*> представляет собой номер версии, которую нужно установить. Например, если текущая версия *ORM 2.07*, то команда будет выглядеть так:

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

Когда установка будет завершена, вам нужно будет подключить загрузчик классов *ClassLoader*:

```
<?php
require 'Doctrine/ORM/Tools/Setup.php';
Doctrine\ORM\Tools\Setup::registerAutoloadPEAR();
```

Все установленные библиотеки будут находится в корневой директории *PEAR* в подкаталоге *Doctrine/*. Помимо этого, вы получите в свое распоряжении небольшую утилиту командной строки, с ее помощью вы будете выполнять типовые задачи. Запустите команду *doctrine* без параметров, чтобы просмотреть доступные ключи запуска:

```
$ doctrine
Doctrine Command Line Interface version 2.0.0BETA3-DEV

Usage:
  [options] command [arguments]

Options:
  --help -h Display this help message.
  --quiet -q Do not output any message.
  --verbose -v Increase verbosity of messages.
  --version -V Display this program version.
  --color -c Force ANSI color output.
  --no-interaction -n Do not ask any interactive question.

Available commands:
  help Displays help for a command (?)
  list Lists commands
dbal
  :import Import SQL file(s) directly to Database.
  :run-sql Executes arbitrary SQL directly from the command line.
orm
  :convert-d1-schema Converts Doctrine 1.X schema into a Doctrine 2.X schema.
  :convert-mapping Convert mapping information between supported formats.
  :ensure-production-settings Verify that Doctrine is properly configured for a
 production  environment.
  :generate-entities Generate entity classes and method stubs from your mapping
 information.
  :generate-proxies Generates proxy classes for entity classes.
  :generate-repositories Generate repository classes from your mapping information.
  :run-dql Executes arbitrary DQL directly from the command line.
  :validate-schema Validate that the mapping files.
orm:clear-cache
```

```
  :metadata Clear all metadata cache of the various cache drivers.
  :query Clear all query cache of the various cache drivers.
  :result Clear result cache of the various cache drivers.
orm:schema-tool
  :create Processes the schema and either create it directly on EntityManager
 Storage Connection or generate the SQL output.
  :drop Processes the schema and either drop the database schema of EntityManager
 Storage Connection or generate the SQL output.
  :update Processes the schema and either update the database schema of
 EntityManager Storage Connection or generate the SQL output.
```

6.2. Загрузка с сайта

Doctrine 2 можно также установить, скачав последнюю версию пакета со страницы загрузки[1].

Настройка и инициализация этой версии *Doctrine* описываются в разделе configuration[Настройка] данного руководства.

6.3. GitHub

В качестве альтернативы можно загрузить последнюю версию Doctrine 2 с GitHub.com:

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

Но, таким образом, у вас будут только исходные коды для пакета *ORM*. Для пакетов *Common* и *DBAL* нужно будет отдельно инициализировать дочерние модули:

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

Этот финт обновит вашу рабочую копию для использования *Doctrine* и тех версий ее дочерних библиотек, которые подходят к клонированной master-ветви *Doctrine 2*.

В разделе Настройка[2] описано как настроить установленную с *Github* версию *Doctrine* для использования автозагрузки классов.

> Не стоит пытаться совместно использовать модули Doctrine-Common, Doctrine-DBAL и Doctrine-ORM, полученные из основной ветви

[1] http://www.doctrine-project.org/downloads
[2] http://odiszapc.ru/doctrine/configuration

> репозитория. ORM просто может не захотеть работать с текущими версиями Common и DBAL из основной ветви. Вместо этого используйте Git Submodules

6.4. Subversion

> Не рекомендуется использовать зеркала Subversion. Они предоставят доступ только к последнему коммиту основной ветви и не загрузят дочерние модули.

Если вы предпочитаете *Subversion*, код можно получить как и раньше с *GitHub.com* по протоколу *Subversion*: $ svn co http://svn.github.com/doctrine/doctrine2.git doctrine2 Однако, так вы получите лишь основную ветвь *Doctrine 2*, без зависимых модулей *Common* и *DBAL*. Их вам придется получить самостоятельно, но тут есть риск столкнуться с несовместимостью версий разных master-ветвей этих библиотек, о чем уже было сказано ранее.

7. Песочница

> Песочница доступна только в версии Doctrine 2, полученной из репозитория GitHub либо в ближайшее время будет доступна в качестве отдельного пакета на странице загрузки[3] (на сегодняшний день (11.02.12) отдельного пакета с песочницей нет на странице загрузки). Вы найдете ее в каталоге $root/tools/sandbox folder.

Песочница представляет собой настроенное и готовое к работе окружение для проведения разных экспериментов или с целью просто поиграться с *Doctrine 2*.

7.1. Обзор

Каталог с песочницей имеет следующую структуру:

```
sandbox/
    Entities/
        Address.php
        User.php
    xml/
        Entities.Address.dcm.xml
        Entities.User.dcm.xml
```

[3] http://www.doctrine-project.org/projects/orm/download

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

Ниже краткое описание того, какие директории и файлы за что отвечают:

В каталоге *Entities* будут располагаться классы ваших моделей-сущностей. В качестве примера там уже лежат две сущности.

В каталоге *xml* находятся описания сущностей в формате XML (если, конечно, вы предпочтете этот формат для их описания). Тут тоже есть два примера их использования.

Каталог *yaml* — то же самое, но для формата YAML.

Файл *cli-config.php* содержит код инициализации, необходимый для консольной утилиты.

doctrine/doctrine.php — это утилита командной строки.

index.php — обычный индексный файл приложения на PHP, в котором используется *Doctrine 2*.

7.2. Небольшая обучалка

Выполните следующую команду из директории *tools/sandbox*: $ php doctrine orm:schema-tool:create Creating database schema… Database schema created successfully! Еще раз взгляните на каталог *tools/sandbox*. Внутри появилась новая база данных *SQLite* под названием *database.sqlite*.

Откройте *index.php* и в самом низу добавьте несколько строк кода, что-то вроде этого:

```
<?php
//... bootstrap stuff

## PUT YOUR TEST CODE BELOW

$user = new \Entities\User;
$user->setName('Garfield');
```

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

echo "User saved!";
Запустите _index.php_ через браузер или из командной строки. На экране появится
 строка “User saved!”.
```

Теперь давайте заглянем в наше хранилище *SQLite*. Снова выполните команду из папки *tools/sandbox*:

```
$ php doctrine dbal:run-sql “select * from users”
```

Результат выполнения:

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

Поздравляем, вы только что сохранили в базе *SQLite* вашу первую сущность с автоматически сгенерированным ID.

Теперь внесите следующие правки в *index.php*:

```
<?php
//... bootstrap stuff

## PUT YOUR TEST CODE BELOW

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

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

Вы создали свой первый *DQL*-запрос для получения из базы данных пользователя с именем ‘Garfield’. Безусловно, это можно было сделать и более простым способом, мы лишь хотели продемонстрировать сам *DQL*. Догадаетесь как это можно сделать проще?

Когда вы создаете новые классы моделей или изменяете существующе, можно перестроить схему базы данных с помощью двух последовательных команд: команды **doctrine orm:schema-tool –drop** для удаления базы (будьте внимательны) а затем **doctrine orm:schema-tool –create** для создание новой схемы.

Глава 1. Архитектура

1.1. Сущности Doctrine

Сущность — легковесный хранимый в БД объект из предметной области вашего приложения. Сущностью может являться любой *PHP* объект, который обладает следующими особенностями:

Сущность не должна быть финальным (*final*) классом или иметь финальные методы.

Все хранимые свойства класса сущности должны быть либо закрытыми (*private*), либо защищенными (*protected*), в противном случае “ленивая загрузка” будет работать некорректно. О ленивой загрузке будет сказано позднее.

В классе сущности не должно быть метода *_clone,* либо определять его нужно с осторожностью.

То же самое касается метода *wakeup*. Постарайтесь вместо него использовать интерфейс *Serializable*.

Если две сущности связаны между собой наследованием (напрямую, либо косвенно), у них не должно быть свойств с одинаковыми именами. Так что, если, например, некая сущность *B* наследуется от сущности *A*, то у нее не должно быть свойства с теми же именами, которые уже есть в *A* (которые наследовалось от *A* к *B*).

В сущности нельзя использовать функцию *func_get_args()* для того, чтобы узнать параметры вызова. Сгенерированные классы прокси не поддерживают этот подход по причине проблем с производительностью, так что ваш код может работать некорректно.

Сущности поддерживают наследование, полиморфизм для своих связей и запросов. Сущностями могут быть как абстрактные, так и обычные классы. Сущности могут наследоваться как от других сущностей, так и от обычных классов. И, наоборот, обычный класс можно запросто наследовать от сущности.

> Замечание по поводу конструкторов. Конструктор класса сущности будет вызван лишь когда вы сами конструируете экземпляр этого класса с помощью оператора **new**. Сама Doctrine никогда не вызывает конструкторы сущностей, так что можете использовать их для собственных задач и передавать им любые аргументы.

1.1.1. Возможные состояния сущностей

Экземпляр определенной сущности может быть иметь из четырех возможных типов: *NEW*, *MANAGED*, *DETACHED* и *REMOVED*.

- *NEW*. Сущность в этом состоянии пока не имеет своего постоянного идентификатора в хранилище, и пока никак не связана с менеджером сущностей (*EntityManager*) и компонентом *UnitOfWork* (например, она была только что создана оператором *new*).
- *MANAGED*. Существующая сущность со своим идентификатором, находящаяся под управление менеджера сущностей*EntityManager*.
- *DETACHED*. Сущность, идентификатор которой больше не связан с менеджером сущностей или компонентом *UnitOfWork*.
- *REMOVED*. Сущность со своим постоянным идентификатором, связанная с *EntityManager*, которая будет удалена из базы данных при завершении транзакции.

1.1.2. Хранимые свойства сущности

Текущее состояние сущности определяется специальными “хранимыми” членами ее экземпляра (они помечаются специальным образом). Доступ к таким переменным должен осуществляться только самими экземпляром объекта сущности с помощью ее методов и никак иначе. Доступ извне к таким полям должен быть закрыт. Обращаться снаружи к ним можно лишь через методы-члены экземпляра класса, например через методы-аксессоры (getter/setter) или другим подобным образом.

Поля-коллекции сущности должны быть определены в соответствии с интерфейсом *Doctrine\Common\Collections\Collection*. Типы коллекций можно использовать для инициализации полей и свойств пока сущность еще не сохранена в базе данных. После того как сущностью станет управляемой (*MANAGED*) или будет отсоединена (*DETACHED*), доступ к ней должен предоставляться через этот интерфейс.

1.1.3. Сериализация сущностей

Не рекомендуется использовать сериализацию сущностей, потому как это может породить определенные проблемы, по крайней мере пока она будет содержать ссылки на объекты “прокси” или находиться под управлением *EntityManager*. Если вы все же решите сериализовать или де-сериализовать сущность, которая все еще содержит ссылки на прокси-объекты, то могут появиться проблемы с приватными свойствами из-за технических ограничений. Прокси-объекты определяют метод *_sleep*, который

не может вернуть имена приватных членов родительских классов. С другой стороны реализовывать интерфейс*Serializable* тоже не решение, потому что тут нас ждут потенциальные проблемы с циклическими ссылками (пока мы не нашли другого пути, так что если вы знаете больше, сообщите).

1.2. EntityManager

Класс *EntityManager* — центральное звено *ORM* в *Doctrine 2*. *EntityManager* позволяет управлять хранимыми объектами и запрашивать их из хранилища.

1.2.1. Модель отложенных транзакций

В *EntityManager* и *UnitOfWork* применяется стратегия под названием “отложенные транзакции”, которая откладывает исполнение *SQL*-запросов до поры до времени, чтобы затем выполнить их максимально эффективно при завершении транзакции, таким образом, что все блокировки на запись смогут быть быстро освобождены. Представьте себе, что *Doctrine* это инструмент для синхронизации сущестующих объектов в памяти с базой данных, которая происходит последовательно и небольшиим порциями. Работайте с вашими объектами, изменяйте их, а когда закончите вызовите метод *EntityManager#flush()*и все изменения будут сохранены.

1.2.2. Паттерн Unit of Work

Внутри *EntityManager* используется механизм *UnitOfWork*, который представляет собой реализацию одноименного паттерна Unit of Work[1]. Он отслеживает все изменения данных в процессе работы и сохраняет их при исполнении транзакции через метод вроде *flush()*. Врядли вам придется работать напрямую с *UnitOfWork*, в основном все задачи будут решаться через класс*EntityManager*.

[1] http://martinfowler.com/eaaCatalog/unitOfWork.html

Глава 2. Установка и конфигурирование

2.1. Начальная загрузка

Инициализация *Doctrine* относительно проста и осуществляется в два этапа:

1. Убедиться, что *Doctrine* имеет доступ ко всем небходимые классам, которые могут использоваться в процессе работы.
2. Получить экземпляр класса *EntityManager*.

2.1.1. Загрузка классов

Давайте начнем с загрузки классов. Вот что нам нужно: настроить несколько загрузчиков (часто называемых “автозагрузчиками”) таким образом, чтобы *Doctrine* при необходимости смогла сама подгружать нужные классы. Пространство имен*Doctrine* содержит очень быстрый и легковесный загрузчик, который можно использовать не только для *Doctrine*, но и других библиотек, если они соответствуют таким требованиям: местоположение классов в дереве каталогов отображается на их именах и пространствах имен, при этом у этих классов должно быть одно общее корневое пространство имен.

> Вы не обязаны использовать только один этот метод для загрузки классов Doctrine. Нет никакой разницы как именно будет загружен класс. Если желаете использовать другой загрузчик, или подключать класс вручную, пожалуйста. Аналогично, загрузчик *Doctrine* может применяться не только для нее самой, но и для других классов, именование которых соответствует описанным выше стандартам.

Ниже будут приведены несколько типичных конфигураций автозагрузчика для разных способов установки *Doctrine*.

> Для тестирования кода ниже, создайте какой-нибудь файл, наапример test.php.

Установка с помощью PEAR

```
<?php
```

```
// test.php

require 'Doctrine/ORM/Tools/Setup.php';

Doctrine\ORM\Tools\Setup::registerAutoloadPEAR();
```

Установка из TAR-архива

```
<?php
// test.php
 require 'Doctrine/ORM/Tools/Setup.php';

$lib = "/path/to/doctrine2-orm/lib";
Doctrine\ORM\Tools\Setup::registerAutoloadDirectory($lib);
```

Установка из Git-репозитория

Здесь подразумевается, что у вас есть все зависимые пакеты, их можно получить с помощью *git submodule update –init*

```
<?php
// test.php
require 'Doctrine/ORM/Tools/Setup.php';

$lib = '/path/to/doctrine2-orm-root';
Doctrine\ORM\Tools\Setup::registerAutoloadGit($lib);
```

Дополнительные компоненты Symfony

Три приведенных примера подразумевают и загрузку классов Symphony: *Symfony Console* и *YAML*, которые также могут понадобиться для *Doctrine 2*.

2.1.2. Получение экземпляра EntityManager

Итак, с загрузкой классов закончили, теперь нужно создать экземпляр менеджера сущностей *EntityManager*. Именно через осуществляется вся основная работа в *ORM Doctrine*.

Для настройки *EntityManager* необходим экземпляр класса *Doctrine\ORM\Configuration* и параметры подключения к базе данных. Ниже показан возможный вариант настройки.

```
<?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/MyProject/
Entities');
$config->setMetadataDriverImpl($driverImpl);
$config->setQueryCacheImpl($cache);
$config->setProxyDir('/path/to/myproject/lib/MyProject/Proxies');
$config->setProxyNamespace('MyProject\Proxies');

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

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

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

При работе с *Doctrine* обязательно используйте метаданные и кэш запросов. *Doctrine*, к слову сказать, прекрасно оптимизирована для работы с кешем. Основные части *Doctrine*, которые оптимизированы в плане кеширования — это метаданные, описывающие отображения (кеш метаданных) и процесс преобразования *DQL* в *SQL* (кеш запросов). Для работы кеширования необходимо минимум памяти, при этом оно позволяет значительно повысить производительность. Помимо этого рекомендуется использовать кеширование с помощью инструментов вроде *APC*. Прежде всего, он закеширует ваш байт-код, что будет полезно при любом раскладе. Кроме того, он поставляет

высокопроизводительное хранилище прямо в оперативной памяти, в котором можно держать кеши для метаданных и запросов.

2.1.3. Быстрая настройка

Пример выше — это полноценная настройка всех необходимых параметров *Doctrine*. Можно упростить этот этап, воспользовавшись одним из предопределенных методов конфигурирования:

```
<?php
use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

$paths = array("/path/to/entities-or-mapping-files");
$isDevMode = false;

$config = Setup::createAnnotationMetadataConfiguration($paths, $isDevMode);
$em = EntityManager::create($dbParams, $config);

// Или если вы используете YAML или XML
$config = Setup::createXMLMetadataConfiguration($paths, $isDevMode);
$config = Setup::createYAMLMetadataConfiguration($paths, $isDevMode);
```

Тут делается несколько предположений:

Если *$devMode* установлена в *TRUE*, в качестве механизма кеширования будет использован стандартный *ArrayCache*, а параметр setAutoGenerateProxyClasses будет установлен в *TRUE* .

В противном случае, механизм кеширования выбирается в порядке перебора очереди: *APC*, *Xcache*, *Memcache*_(127.0.0.1:11211). *_setAutoGenerateProxyClasses* в этом случае устанавливается в *FALSE*.

Если третий аргумент (*$proxyDir*) отсутствует, будет использован системный каталог для временных файлов.

2.2. Параметры конфигурации

Следующие разделы описывают все возможные параметры конфигурации экземпляра класса *Doctrine\ORM\Configuration*.

2.2.1. Каталог для прокси (* Обязательно *)

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

Тут происходит установка и получение пути к директории, в которой будут храниться сгенерированные Доктриной прокси-классы. Более детальная информация о прокси-классах и их использовании в *Doctrine* приведена в разделе “Прокси-объекты” далее по тексту.

2.2.2. Пространства имен для прокси (* Обязательно *)

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

Здесь указывается какое пространство имен будет использовано для сгенерированных прокси-классов. Более подробное описание приведено в разделе, посвященном прокси-объектам.

2.2.3. Драйвер метаданных (* Обязательно *)

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

Код выше демонстрирует как определить какой вариант реализации драйвера метаданных будет использован для обработки метаданных в ваших классах.

Существует 4 разновидности их реализаций:

```
_Doctrine\ORM\Mapping\Driver\AnnotationDriver
_Doctrine\ORM\Mapping\Driver\XmlDriver_
_Doctrine\ORM\Mapping\Driver\YamlDriver_
_Doctrine\ORM\Mapping\Driver\DriverChain_
```

Во многих примерах к этому руководству используется *AnnotationDriver*. Для получения информации об использовании драйверов XML и YAML обратитесь к соответствующим разделам.

Драйвер, работающий с аннотациями настраивается с помощью фабричного метода класса *Doctrine\ORM\Configuration*:

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

Здесь необходимо указать путь к классам сущностей, иначе массовые операции над всеми сущностями, осуществляемые через консоль, не будут работать корректно. Все драйвера метаданных в качестве этого параметра позволяют указать как одиночную директорию, так и список, это позволит одному единственному драйверу работать с несколькими директориями, в которых лежат сущности.

2.2.4. Кеш метаданных (* Рекомендуется *)

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

Здесь устанавливается какой механизм будет использоваться для кеширования метаданных, т.е. всей той информации, что вы укажете в аннотациях, через *XML* или *YAML*, так что при каждом запросе ее не нужно будет заново парсить и обрабатывать, что весьма накладно. Какую бы реализацию вы не выбрали, она должна наследовать общий для всех интерфейс*Doctrine\Common\Cache\Cache*.

Кеширование метаданных очень полезная штука. Нет ни одной адекватной причины, чтобы непользоваться ею.

Для продакшн-среды рекомендуется использовать следующие механизмы кеширования:

```
_Doctrine\Common\Cache\ApcCache_
_Doctrine\Common\Cache\MemcacheCache_
_Doctrine\Common\Cache\XcacheCache_
```

В среде для разработки более предпочтителен *Doctrine\Common\Cache\ArrayCache*, которые будет постоянно перекешировать данные при каждом запросе.

2.2.5. Кеш запросов (* Рекомендуется *)

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

Здесь устанавливается возможная реализация механизма кеширования для *DQL*-запросов, т.е. для результата разбора самого выражения *DQL*, которое затем преобразуется в *SQL* и информацию о том, как обрабатывать набор *SQL* запросов. Обратите внимание, это не кеш запросов в обычном понимании этого слова, он не затрагивает результаты запроса, так что кеширование не приведет к искажению в данных как это иногда случается. Этот вид кеша создан чисто для оптимизации и не имеет побочных эффектов, кроме разве что небольшого потребления памяти при работе.

Как и для метаданных, крайне рекомендуется использовать кеш и для запросов.

На продакшне используйте эти три варианта:

```
_Doctrine\Common\Cache\ApcCache_
_Doctrine\Common\Cache\MemcacheCache_
_Doctrine\Common\Cache\XcacheCache_
```

При разработке проще использовать *Doctrine\Common\Cache\ArrayCache*.

2.2.6. Журналирование SQL-запросов (* Опционально *)

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

Устанавливается регистратор для сохранения всех *SQL* запросов, исполняемых через *Doctrine*. Класс регистратора должен определять интерфейс *Doctrine\DBAL\Logging \SQLLogger*. Самый простой вариант реализации журнала находится в *Doctrine\DBAL \Logging\EchoSQLLogger*, он выводит логи на стандартный поток вывода с помощью *echo* и *var_dump*.

2.2.7. Автоматическое создание классов прокси (* Опционально *)

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

Определяет, следует ли генерировать классы прокси автоматически во время выполнения скрипта. Если поставить *FALSE*, то классы нужно будет сгененрировать вручную консольной командой с ключом *generate-proxies*. В продакшн среде крайне рекомендуется отключить авто-генерацию.

2.3. Конфигурация для development- и production-сред

Инициализация *Doctrine 2* должна осуществляться двумя способами с использованием двух различных моделей исполнения. На продакшне лучше использовать *APC* или *Memcache*, которые имеют много преимуществ в плане производительности. Но при разработке это не совсем то, что нужно, потому что вам не избежать частых ошибок, например, когда вы обновили какую-то сущность, а кеш содержит устаревшие данные. Так что для разработки больше подойдет *ArrayCache*.

Помимо этого, на продакшне следует отключить авто-генерацию прокси классов, при разработке же, наоборот, оставить эту возможность. Включенная авто-генерации может негативно влиять на производительность, например, когда несколько прокси классов будут повторно генерироваться во время исполнения скрипта. В этом случае запросы к файловой системе могут даже больше сказаться на производительности, чем сами запросы к базе данных. Кроме того, прокси устанавливает блокировку на файл, что при выполнении нескольких параллельных запросов будет порождать “бутылочные горлышки” .

2.4. Параметры подключения

Параметр *$connectionOptions*, передаваемый в качестве первого аргумента в метод *EntityManager::create()* может быть массивом либо экземпляром класса *Doctrine\DBAL \Connection*. Если был передан массив, он сразу же передается в фабричный метод *DBAL Doctrine\DBAL\DriverManager::getConnection()*. Конфигурация *DBAL* описана в соответствующем разделе.

2.5. Объекты прокси

Прокси — это объект, который может подставляться и использоваться вместо “реального”. С помощью прокси объекта можно незаметно добавить новый функционал к основному объекту, так что самому объекту об этом беспокоиться не нужно. В*Doctrine 2* объекты прокси используются для реализации некоторых функций, основная из которых — ленивая загрузка.

Прокси объекты с помощью ленивой загрузки помогают сохранить взаимосвязь между набором объектов, находящимся в памяти и остальными объектами, которые еще не были загружены. Это очень важное свойство, без которого на границах графа ваших объектов будут всегда находится как бы неполноценные объекты, с некоторым количеством оборванных связей.

В *Doctrine 2* используется вариация паттерна прокси, которая работает следующим образом: сначала генерируются классы, расширяющие ваши сущности, затем к ним добавляется возможность ленивой загрузки. После этого *Doctrine* при запросе основного объекта, может вернуть вместо него прокси-версию. Это происходит в двух случаях:

2.5.1. Прокси для ссылок

С помощью метода *EntityManager#getReference($entityName, $identifier)* можно получить ссылку на сущность по заданному идентификатору без ее загрузки из базы данных. Это бывает полезно, например, в целях повышения производительности, когда нужно установить связь с сущностью, зная ее идентификатор. Вы можете просто сделать следующее:

```
<?php
// $em это EntityManager, $cart это экземпляр класса MyProject\Model\Cart
// $itemId идентификатор товара, полученный откуда угодно
$item = $em->getReference('MyProject\Model\Item', $itemId);
$cart->addItem($item);
```

Смотрите, мы добавляем товар в корзину без его реальной загрузки из базы данных. Если мы вызовем какой-нибудь метод у нашего объекта *$item*, он будет полностью проинициализирован, его состояние обновится до актуального из базы данных, и это произойдет прозрачно, именно это и называется ленивой загрузкой *(lazy-loading)*. Здесь *$item* это не что иное как экземпляр прокси класса, который был сгенерирован для

класса *Item*, но для вас это не должно иметь значения. Просто забейте, работа объектов прокси полностью прозрачна.

2.5.2. Прокси для связей

Второй случай, когда *Doctrine* использует прокси это при запросах на получение объектов. Всякий раз, когда вы запрашиваете объект, имеющий однозначную взаимосвязь с другим объектом, поддерживающим ленивую загрузку, и при этом не используете *JOIN*, *Doctrine* заменит все такие объекты их прокси аналогами. Как и остальные прокси объекты, они будут инициализированы при первой же попытке доступа к ним.

> Использование соединений *JOIN* в *DQL* или *SQL*, подразумевает “жадную загрузку” всех взаимосвязей. Тем самым для такого запроса будет переопределен параметр *“fetch”*, установленный в отображении для связи.

2.5.3. Генерация классов прокси

Классы прокси можно сгенерировать либо вручную через консольную утилиту *Doctrine*, либо автоматически. Параметр конфигурации, отвечающий за это:

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

Для большего удобства за параметр по умолчанию принято значение *TRUE*. Однако, на продакшене в целях повышения производительности лучше отключить эту возможность. Не делайте этого в процессе разработки, т.к. это может породить ошибки, связанные с невозможностью загрузки классов (они не будут найдены), либо загрузки устаревших классов. Например, вы добавили новый метод к сущности, а в прокси классе он пока отсутствует. Чтобы избежать этого, заранее сгенерируйте новые прокси через консоль:

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

2.6. Несколько источников метаданных

При работе с различными компонентами с помощью *Doctrine 2*, вы можете применять два разных драйвера метаданных, например *XML* и *YAML*. У вас есть возможность

объединить эти драйвера с помощью компонента *DriverChain* на основе пространств имен:

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

При этом загрузка определенного пространства имен возлагается на соответствующий драйвер. Драйвер пробегает по всем пространствам имен и ищет совпадения, сравнивая название класса сущности с этим пространством имен при помощью простого выражения *strpos() === 0*. Поэтому, если дочерние пространства имен используют иные реализации драйвера метаданных для обработки, сами драйвера нужно расположить в определенном порядке.

2.7. Репозиторий по-умолчанию (* Опционально *)

Устанавливает полное имя *(Fully-Qualified Class Name, FQCN)* для дочернего класса *EntityRepository*. Этот класс будет использован для всех сущностей, у которых отсутствует собственный класс репозитория.

```
<?php
$config->setDefaultRepositoryClassName($fqcn);
$config->getDefaultRepositoryClassName();
```

По-умолчанию используется класс *Doctrine\ORM\EntityRepository*. Каждый класс репозитория необходимо наследовать от*EntityRepository*, иначе вы получите исключение типа *ORMException*.

Глава 3. Часто задаваемые вопросы

Этот раздел постоянно пополняется. От вас приходит много вопросов, на многие из них мы не отвечаем, но помним о чем спрашивают чаще всего. Если у вас есть вопрос, подписывайтесь на рассылку или подключайтесь в *IRC* к каналу *#doctrine*.

3.1. Схема базы данных

3.1.1. Как установить кодировку и COLLATION для таблиц MySQL?

Кодировку нельзя установить через аннотации и файлы XML- или YAML- отображений. Для настройки базы на работу с определенной кодировкой нужно установить ее как кодировку по-умолчанию в конфигурации MySQL, либо указать эти параметры при создании таблиц. В этом случае они будут автоматически применяться ко всем создаваемым таблицам и столбцам.

3.2. Классы сущностей

3.2.1. Обращаюсь к переменной-члену, а мне возвращается значение NULL, что не так?

Похоже вы объявили переменную как публичную (public), тем самым нарушив одно из ограничений, накладываемых на сущности. Для корректной работы объектов прокси хранимые переменные-члены у сущности должны быть объявлены как приватные или защищенные.

3.2.2. Как добавить для столбца значение по умолчанию?

Doctrine не поддерживает установку значений по умолчанию для столбцов как это делает в язык SQL через ключевое слово*DEFAULT*. Однако, это и не нужно, просто используйте для этого языковыми конструкции PHP:

```
class User
{
```

```
    const STATUS_DISABLED = 0;
    const STATUS_ENABLED = 1;

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

3.3. Отображения

3.3.1. При выполнении $em→flush() появляются ошибки, связанные с нарушением уникальности данных

Doctrine не проверяет не добавляете ли вы в базу данных сущности с уже дублирующимися ключами *PRIMARY*, или не добавляете ли вы дважды в коллекцию одну и ту же сущность. Если есть какие-то опасения на этот счет, всегда можно сделать соответствующую проверку перед вызовом *$em→flush()*.

В Symfony2[1] для этого существует соответствующий компонент под названием _Unique Entity Validato_r.

Проверить содержит ли коллекция заданную сущность можно с помощью метода *$collection→contains($entity)*. Для коллекции с параметром *FETCH* равным *LAZY* этот метод просто инициализирует коллекцию, однако если *FETCH* равен *EXTRA_LAZY*, то для проверки принадлежности Doctrine выполнит *SQL*-запросы.

3.4. Связи

3.4.1. Мне выдается исключение InvalidArgumentException с сообщением “A new entity was found through the relationship..”. Что не так?

Это исключение выбрасывается при исполнении метода *EntityManager#flush()* в случае, если в общей схеме данных существует объект, содержащий ссылку на некоторый объект, неизвестный *Doctrine*. Чтобы было проще понять, представьте, например, что вы загрузили из базы данных сущность пользователя *(User)* со своим идентификатором, затем создали новый объект и привязали его к объекту *User* через одну из существующих у него взаимосвязей. Затем, если

[1] http://www.symfony.com/

вы вызовете*EntityManager#flush()*, не сообщив об свежесозданном объекте *Doctrine* с помощью метода *EntityManager#persist($newObject)*, будет выброшено такое исключение.

Решить проблему можно двумя путями:

1. Предварительно вызвать метод *EntityManager#persist($newObject)* для вновь созданного объекта
2. Использовать свойство *cascade=persist* для связи, содержащей новый объект

3.4.2. Могу ли я как-то отфильтровать связанный с сущностью набор данных?

Такой возможности нет ни в версии_ Doctrine 2.0_, ни в *2.1*. Для фильтрации набора сущностей лучше использовать возможности DQL.

3.4.3. После вызова метода clear() на наборе типа “один ко многим”, сущности почему-то не были удалены

Это ожидаемое поведение, связанное с концепции обратной и владеющей сторон связи, и тем, как с ними работает движок*Doctrine*. Когда сторона взаимосвязи “один ко многим” объявляется обратной, это означает, что Doctrine перестает воспринимать все изменения сущности, внесенные по эту сторону связи.

В качестве замены методу clear() можно использовать перебор по коллекции и установить ссылку на владеющей стороне отношения “многие к одному” в *NULL*, тем самым все сущности будут выброшены из коллекции. После этого будут выполнены соответствующие UPDATE запросы к базе данных .

3.4.4. Как добавить дополнительные столбцы к связи вида “один ко многим”?

В качестве столбцов в определении таблицы для связи “один ко многим” допускается использовать только внешние ключи. Для добавления в такие связи дополнительных столбцов есть возможность использования внешних ключей как первичных (primary). Такая фича появилась в *Doctrine 2.1*.

Как это сделать читайте в обучалке по составным первичным ключам[2].

[2] http://www.doctrine-project.org/docs/orm/2.1/en/tutorials/composite-primary-keys.html

3.4.5. Можно ли осуществлять постраничную выборку из присоединенных коллекций?

Если вам нужно выражение DQL, которое может это сделать, мы вас огорчим, нет простого способа итерации по такой коллекции с помощью оператора LIMIT.

Doctrine не предоставляет решения “из коробки”, однако существует несколько расширений, позволяющих это сделать:

1. DoctrineExtensions[3]
2. Pagerfanta[4]

3.4.6. Почему постраничная выборка некорректно работает с соединениями (fetch-joins)?

Для ограничения результирующего набора данных движок постраничной выборки в *Doctrine* использует ключевое слово*LIMIT* (или его эквивалент). Однако, когда вы выполняете JOIN запрос, он не возвращает верное число записей, т.к. соединения через связи вида “один ко многим” и “многие ко многим” в качестве результирующего количества строк возвращают произведение числа строк на число связанных сущностей.

Для решения проблемы смотрите предыдущий вопрос .

3.5. Наследование

3.5.1. Можно ли применять наследование в Doctrine 2?

Да, допускается наследование применительно как к единичным, так и присоединенным таблицам.

Детали описаны в соответствующей главе.

[3] http://github.com/beberlei/DoctrineExtensions
[4] http://github.com/whiteoctober/pagerfanta

3.5.2. Почему Doctrine не создает прокси объекты для иерархии классов?

Когда в сущности создается связь с родительским классом вида “многие к одному” или “один к одному”, *Doctrine* не может определить какой из классов в действительности выступает в качестве “внешнего ключа”. Чтобы выяснить это, *Doctrine* должна выполнить *SQL* запрос для поиска этой информации в базе данных.

3.6. EntityGenerator

3.6.1. Почему EntityGenerator не делает ту или иную вещь?

EntityGenerator не является полноценным генератором кода, решающим любые задачи. Генерация кода отныне не является основным приоритетом *Doctrine 2* (в отличие от *Doctrine 1*). *EntityGenerator* конечно помогает в работе, но это не панацея.

3.6.2. Почему EntityGenerator некорректно работает с иерархиями?

Дело в том, что *EntityGenerator* не может правильно угадать взаимоотношения в иерархии классов. Вот почему создание наследуемых сущностей затруднено и пока не работает как нужно. Обрабатывать подобные ситуации вам нужно будет вручную.

3.7. Производительность

3.7.1. Почему когда я обращаюсь к данным через отношение “один к одному” это всякий раз порождает SQL запрос?

Если *Doctrine* обнаружила, что обращение к данным происходит с обратной стороны связи (inverse side), то для загрузки целевого объекта будет выполнен дополнительный запрос. Так происходит потому что нельзя определить, вдруг там вообще нет никакого объекта (стоит NULL), или там прокси-объект, а какой у него идентификатор одному богу известно.

Чтобы избежать этой проблемы, используйте запрос который сразу получает всю необходимую информацию.

3.8. Язык DQL

3.8.1. Что представляет собой DQL?

DQL это язык, очень похожий на *SQL*, имеющий ряд особенностей:

1. Вместо названия таблиц и столбцов он использует названия классов и их полей, абстрагируясь между бэкендом и объектной моделью.
2. *DQL* применяет информацию, хранящуюся в метаданных, давая возможность использовать сокращения при написании запросов. Например, если вы не указали оператор *ON* при соединении таблиц *Doctrine* сделает это за вас.
3. *DQL* добавляет некоторую функциональность, связанную с управлением объектами и преобразованием их в запросы*SQL*.

Конечно, есть и недостатки:

1. Синтаксис его немного отличается от *SQL*, поэтому нужно четко уяснить различия.
2. Чтобы не зависеть от особенностей каждой конкретной СУБД, *DQL* реализует лишь общее подмножество *SQL*, поддерживаемое всеми продуктами.
3. В DQL нельзя использовать специфичный для конкретной СУБД функционал и различные оптимизации, пока они в явном виде не будет реализованы вами.
4. Для некоторых конструкций в *DQL* применяются вложенные *SELECT'ы*, которые в том же *MySQL* работают медленно.

3.8.2. Можно ли в DQL выполнить сортировку по заданной функции (например, ORDER BY RAND())?

Нет. Если вам нужен такой функционал, используйте низкоуровневые запросы или ищите другое решение. Имейте ввиду: начиная с 1000 записей в таблице сортировка вида *ORDER BY RAND()* выполняется крайне медленно.

Глава 4. Отображения

В главе приведено описание механизма отображений объектов и их свойств на структуру базы данных. Отображение для связей будет описано в следующей главе.

4.1. Драйвера отображений

Любые отображения в Doctrine задаются через метаданные. Существует несколько способов для описания метаданных:

1. Аннотации Docblock
2. XML
3. YAML

В примерах к данному руководству в основном будут использоваться аннотации *Dockblock*, однако во многих из них приведены и альтернативы на *XML* и *YAML*. Более детальному описанию отображений через *XML* и *YAML* посвящены отдельные главы. Кроме того, отдельная глава будет посвящена синтаксису аннотаций.

> Если вас беспокоит вопрос, какой же из описанных механизмов имеет лучшую производительность, вот ответ: все они одинаково быстро работают. После того как метаданные считаны из источника (*XML*, *YAML* или аннотаций) они тут же сохраняются в экземпляре класса Doctrine\ORM\Mapping\ClassMetadata, который, в свою очередь, хранится в специальном кеше для метаданных. Так что в итоге все они будут выполняться одинаково быстро. Если вы все же решите отказаться от кеша метаданных (что делать крайне не рекомендуется), то преимущество будет у драйвера XML, т.к. разбор этого формата осуществляется напрямую библиотекой PHP.

4.2. Аннотации Docblock

Вы, наверное, уже сталкивались с *Dockblock* аннотациями, например, когда готовили метаданные для утилит вроде*PHPDocumentor* (*@author*, *@link*, …). С их помощью в документацию можно встроить некоторые метаданные, а затем использовать их в своих целях. *Dockblock* аннотации работают также._ Doctrine 2_ обобщает концепцию аннотаций, так что их можно использовать для определения любых типов метаданных, а это позволяет определять новые аннотации. Для более эффективного использования

и уменьшения вероятности их конфликта имен, в Doctrine 2 для аннотаций используется альтернативный синтаксис, чем-то напоминающий аннотации в Java 5.

Собственно сама реализация Dockblock аннотаций заложена в пространстве имен *Doctrine\Common\Annotations*, относящуюся к пакету *Common*. Среди прочего, в *Dockblock* можно использовать пространства имен, а также вложенные аннотации. Вы все увидите в примерах. Doctrine 2 для задания метаданных, описывающих отображения, использует собственное подмножество директив.

Если вас по каким-то причинам не устраивают Dockblock аннотации, в качестве альтернативы могут быть использованы XML или YAML. Вы даже можете написать свой собственный механизм для работы с метаданными.

4.3. Хранимые классы

Итак, сущность это класс. Чтобы подготовить класс для работы в среде *ORM*, его необходимо превратить в сущность. Это делается с помощью специальной аннотации *@Entity*:

PHP

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

XML

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

YAML

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

По умолчанию сущность будет преобразована в таблицу БД, имя которой соответствует имени класса сущности. Если вы хотите задать иное имя, можно воспользоваться аннотацией @*Table*:

PHP

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

XML

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

YAML

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

В этом примере экземпляр сущности *MyPersistentClass* будет храниться в базе данных в виде таблицы с именем*my_persistent_class*.

4.4. Отображения типов в Doctrine

В *Doctrine* можно устанавливать соответствия между типами *PHP* и соответствующим им типам в базе данных. Все отображаемые типы, поставляемые с *Doctrine*, полностью совместимы с различными СУБД. Как будет показано далее в этой главе, можно создавать и пользовательские типы, совместимость которых, однако, уже не гарантируется.

Например, тип Doctrine **string** устанавливает соответствие между типом *string* в *PHP* и типом *VARCHAR* (или *VARCHAR2*, в зависимости от СУБД). Ниже приведено краткое описание доступных типов:

string: Отображает *VARCHAR* в строку *PHP*. **integer**: Отображает *INT* на целочисленный тип. **smallint**: Отображает *SMALLINT* так же на целочисленный тип в *PHP*. **bigint**: Отображает *BIGINT* на строку *PHP*. **boolean**: Отображает булевый тип (в *MySQL* это_ TINYINT(1))_ на булевый тип в *PHP*. **decimal**: Отображает *DECIMAL* на тип *double*. **date**: Отображает *DATETIME* на объект типа *DateTime*. **time**: Отображает *TIME* также на объект типа *DateTime*. **datetime**: Отображает типы *DATETIME* или *TIMESTAMP* на объект *DateTime* в *PHP*. **text**: Отображает тип *CLOB* на строку *PHP*. **object**: Отображает тип *CLOB* на объект *PHP* с помощью сериализации (*serialize()* и *unserialize())* **array**: Отображает тип *CLOB* на объект *PHP* с помощью сериализации (*serialize()* и *unserialize())* **float**: Отображает тип *FLOAT* (двойной точности) на тип *double* в *PHP*. **Важно**: Отображение работает только с настройками локали, в которых в качестве десятичного разделителя используется точка.

> Запомните, отображаемые типы в *Doctrine* не являются ни *SQL*- ни *PHP*-типами! Они лишь устанавливают соответствие между этим двумя типами данных. Кроме того, они зависимы от регистра. Например, *DateTime* это не одно и тоже что *datetime*, который уже входит в комплект *Doctrine*.
>
> Типы *DateTime* и *Object* всегда сравниваются по ссылке, а не по значению. Обновление их значений будет происходить в случае, если изменилась сама ссылка, поэтому объекты этих типов ведут себя подобно объектам, значение которых не может быть изменено.

При работе с типами даты и времени предполагается, что используется стандартная временная зона, установленная функцией date_default_timezone_set()[1] или через параметр *date.timezone* в *php.ini*. При использовании в приложении различных временных зон результат может быть непредсказуемым.

Если в вашем приложении все же требуется какая-то специальная обработка временных зон, то нужно делать это вручную, перегоняя значения даты в *UTC* и обратно. В руководстве есть соответствующая статья[2], в которой описана работа с типами даты и времени при использовании различных временных зон.

[1] http://docs.php.net/manual/en/function.date-default-timezone-set.php
[2] http://www.doctrine-project.org/docs/orm/2.1/en/cookbook/working-with-datetime.html

4.5. Сопоставление свойств

После того как класс стал сущностью, можно начать устанавливать отображения для его отдельных членов. Ниже буду описаны лишь самые простые поля со скалярными типами, вроде целых чисел и строк. Связи с другими объектами описываются в главе “Отображения связей”.

Свойство может быть отмечено как хранимое с помощью аннотации *@Column*. Эта аннотация требует указания как минимум одного атрибута — типа столбца, в качестве которого можно использовать любой из доступных типов *Doctrine*. Если опустить этот параметр, в качестве типа будет использован *string*, ведь он самый гибкий и универсальный.

Пример:

PHP

```php
<?php
/** @Entity */
class MyPersistentClass
{
    /** @Column(type="integer") */
    private $id;
    /** @Column(length=50) */
    private $name; // 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
```

В этом примере полю *id* соответствует столбец *id* целочисленного типа, а полю name — столбец *name* и тип *string*, используемый по умолчанию. Как видите, по умолчанию именам столбцов будут присвоены такие же имена, как и у полей. Задать столбцу иное имя можно с помощью атрибута *name*:

PHP

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

XML

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

YAML

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

Для более тонкой настройки отображений на столбцы БД можно использовать атрибуты. Вот их полный список:

type: (необязательный параметр, значение по умолчанию: ‘string’) Отображаемый тип для столбца.
name: (необязательный параметр, по умолчанию соответствует имени свойства) Название столбца в базе данных.

length: (необязательный параметр, по умолчанию равен 255) Длина значения столбца в базе данных. (Применяется только для строковых типов).
unique: (необязательный параметр, по умолчанию равен *FALSE*) Определяет, являются ли значения столбца уникальными.
nullable: (необязательный параметр, по умолчанию равен *FALSE*) Могут ли значения столбца принимать пустые значения*(NULL)*.
precision: (необязательный параметр, по умолчанию равен 0) Точность для неупакованных чисел с плавающей точкой. (Применяется только для столбцов типа DECIMAL.)
scale: (необязательный параметр, по-умолчанию равен 0) Задает шкалу (scale) для неупакованных чисел с плавающей точкой. (Применяется только для столбцов типа *DECIMAL*.

4.6. Пользовательские типы

В *Doctrine* можно создавать свои собственные отображаемые типы. Это бывает полезно, когда не удается найти подходящий тип для конкретной задачи или вас по каким-то причинам не устраивает поведение существующих типов.

Для создания нового типа нужно наследовать класс *Doctrine\DBAL\Types\Type* и переопределить нужные вам методы. Ниже приведет шаблон такого типа:

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

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

/**
 * Мой mun datatype.
*/
class MyType extends Type
{
    const MYTYPE = 'mytype'; // modify to match your type name

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform
 $platform)
    {
     // Возвращает SQL, используемый для создания столбца. Если нужен совместимый
 тип, используйте параметр $platform.
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
```

```
    {
     // Вызывается во время чтения значения из БД. Тут можно делать разные
 преобразования, в т.ч. основываясь на параметре $platform.
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
     //  Вызывается во время записи значения в БД . Тут также можно делать разные
 преобразования, в т.ч. основываясь на параметре $platform .
    }

    public function getName()
    {
         return self::MYTYPE; // возвращает имя типа
    }
}
```

Имейте ввиду, у этой схемы есть ограничения:

Если свойство принимает значение *NULL*, метод *convertToDatabaseValue()* не будет вызываться. Модуль *UnitOfWork* никогда не осуществляет конвертацию значений в базу данных, которые не изменились в запросе.

После того как вы реализуете свой тип, нужно сообщить о нем *Doctrine*. Это можно сделать с помощью метода*Doctrine\DBAL\Types\Type#addType($name, $className)*. Вот пример:

```
<?php
// in bootstrapping code

// ...

use Doctrine\DBAL\Types\Type;

// ...

// Register my type
Type::addType('mytype', 'My\Project\Types\MyType');
```

Как видно из примера, при регистрации типа ему присваивается уникальное имя на ваш выбор, которое соответствует полному имени реализующего его класса. После регистрации, тип готов к использованию:

```
<?php
class MyPersistentClass
```

```
{
    /** @Column(type="mytype") */
    private $field;
}
```

Чтобы *Schema-Tool* правильно смогла конвертировать ваш новый тип данных *mytype* в экземпляр класса *MyType*, нужно будет дополнительно зарегистрировать соответствие для этих типов:

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

Теперь когда *Schema-Tool* обнаружит столбец типа *db_mytype*, она преобразует его в экземпляр класса, соответствующий доктриновскому типу *mytype*. Имейте ввиду, во избежание конфликтов каждый тип из базы данных может соответствовать только одному отображаемому типу *Doctrine*.

4.7. Пользовательское определение для столбца

Для любого столбца можно задать его собственное определение с помощью атрибута *columnDefinition* аннотации *@Column*. Таким образом можно задать любое определение, следующие за именем столбца, как если бы вы это делали через SQL запрос, например, задать комментарий для столбца (и он будет сохранен в БД).

И, к сожалению, при использование этого атрибута Schema-Tool не сможет вычислять изменения в схеме базы данных.

4.8. Идентификаторы / Первичные ключи

Любая сущность в *Doctrine* должна иметь свой идентификатор. Задать его можно с помощью аннотации *@Id*:

PHP

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

XML

```
<doctrine-mapping>
    <entity name="MyPersistentClass">

        <field name="name" length="50" />
    </entity>
</doctrine-mapping>
```

YAML

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

Если не задано иное, идентификатор для новой сущности должен задаваться вручную. Это означает, что прежде чем передавать сущность методу *EntityManager#persist($entity)* нужно установить для нее идентификатор.

В качестве альтернативы ручной установке есть возможность генерировать значения идентификатора автоматически. Это достигается использованием аннотации *@GeneratedValue*:

PHP

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

XML

```
<doctrine-mapping>
```

```
    <entity name="MyPersistentClass">

            <generator strategy="AUTO" />

        <field name="name" length="50" />
    </entity>
</doctrine-mapping>
```

YAML

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

Это сообщит Doctrine, что идентификатор должен быть сгенерирован автоматически. Как именно будет происходить генерация определяется атрибутом *strategy*, по умолчанию он принимает значение *AUTO*. Это значит, что база данных сама решит как это будет происходить и использует собственный механизм. Подробности читайте далее.

4.8.1. Стратегии генерации идентификаторов

В предыдущем примере показано как использовать стратегию генерации идентификатора по-умолчанию, при этом не важно какая СУБД используется. Но задать эту стратегию можно и явно, это дасть возможность использовать некторые дополнительные функции.

Ниже приведен список возможных стратегий:

AUTO (по умолчанию): используется наиболее предпочтительная стратегия для текущей платформы БД. Для *MySQL*, *SQLite* и*MsSQL* это значение равно *IDENTITY*, для *Oracle* и *PostgreSQL — SEQUENCE* . Так что этот вариант самый универсальный.

SEQUENCE: Для генерации идентификатора будет использована последовательность. Эта стратегия не обеспечивает полной совместимости, т.к. поддерживается только в *Oracle* и *PostgreSql*.

IDENTITY: Сообщает *Doctrine* использовать специальные идентификационные столбцы, значение которых генерируется само при вставке новой записи в таблицу. Данный механизм поддерживается следующими платформами: *MySQL/ SQLite*_(AUTO_INCREMENT), *_MSSQL (IDENTITY)* и *PostgreSQL (SERIAL)*.

TABLE: позволяет использовать отдельную таблицу для генерации идентификаторов. Этот механизм обеспечивает полую совместимость. **Но эта стратегия пока еще не реализована!**

NONE: Идентификатор не генерируется и должен быть установлен вручную. Назначать его нужно до передачи сущности методу *EntityManager#persist*. Эта стратегия будет автоматически использована при отсутствии аннотации *@GeneratedValue*.

Генератор последовательностей

В настоящий момент генерация последовательностей возможна только для *Oracle* и *Postgres*. При использовании этой стратегии становятся доступны несколько новых параметров:

PHP

```php
<?php
class User
{
    /**
    * @Id
    * @GeneratedValue(strategy="SEQUENCE")
    * @SequenceGenerator(sequenceName="tablename_seq", initialValue=1, allocationSize=100)
    */
    protected $id = null;
}
```

XML

```xml
<doctrine-mapping>
    <entity name="User">

            <generator strategy="SEQUENCE" />
            <sequence-generator sequence-name="tablename_seq" allocation-
size="100" initial-value="1" />

    </entity>
```

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

YAML

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

initialValue задает значение, с которого начнется последовательность.

Обратите внимание на параметр *allocationSize, он* позволяет оптимизировать производительность запросов *INSERT*. Вот как это работает. Он определяет на какое количество будет увеличенная последовательность прежде чем ее следующее значение будет физически запрошено из базы данных. Если это значение больше единицы, Doctrine будет сама генерировать идентификаторы для следующих *allocationSizes* сущностей. В примере выше при значении *allocationSize=100 Doctrine* для того, чтобы сгенерировать идентификаторы для 100 новых сущностей нужно лишь единожды обратиться к объекту *SEQUENCE* в базе данных.

Значение по умолчанию для *allocationSize* равно 10. *Schema-Tool* сама обнаруживает этот параметр и преобразовывает его в конструкцию_ INCREMENT BY_ в операторе *CREATE SEQUENCE*. Если структура базы данных была создана вручную без использования *Schema-Tool*, вам нужно проследить, чтобы *allocationSize* не превышал соответствующего ему значения в конструкции *INCREMENT BY*, иначе начнут генерироваться дубликаты. И последнее. Возможно использования параметра *strategy=”AUTO”* совместно с *@SequenceGenerator*. В этом случае генерация последовательностей будет применяться только если есть поддержка со стороны БД (для *Oracle* и *PostgreSQL*)

4.8.2. Составные ключи

Doctrine 2 позволяет работать с составными первичными ключами. Правда, по сравнению с единичными ключами здесь есть пара ограничений. Так, использовать

аннотацию *@GeneratedValue* можно только для простых одиночных ключей. Для составных ключей идентификатор придется генерировать вручную перед вызовом *EntityManager#persist()*.

Для создания составного ключа просто отметьте аннотацией *@Id* все поля, из которых он состоит.

4.9. Экранирование ключевых слов

Иногда нужно экранировать название столбца или таблицы, если оно конфликтует с ключевыми словами той или иной СУБД. Дать понять *Doctrine* что-именно следует экранировать можно, заключив название таблицы или столбца в обратные кавычки:

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

После этого *Doctrine* будет будет экранировать это значение во всех *SQL* запросах.

Экранирование ключевых слов не работает для имен столбцов, используемых для соединений таблиц, а также в названиях столбцов-дискриминаторов.

Экранирование в основном предназначено для совместимости с устаревшими схемами данных. По возможности следует избегать подобных ситуаций. Не надо применять экранирование только потому, чтобы использовать нестандартные символы, вроде тире, в названиях столбцов. Кроме того, у *Schema-Tool* скорее всего будут проблемы, если экранирование используется в случаях, связанных с регистрозависимостью символов (например, в Oracle).

Глава 5. Отображение связей

В этой главе приводится описание того как Doctrine работает со связанными сущностями, т.е. сущностями, между которыми существуют некоторые отношения. Для начала будет дано описание концепции прямой и обратной сторон связи. Это очень важный момент, он поможет понять принцип работы двусторонних связей. Главное, нужно усвоить, что связи могут быть одно- и дву- сторонними.

5.1. Прямая и обратная стороны связи

При работе с двусторонними связями важно понимать суть прямой (owning) и обратной (inverse) сторон связи. Давайте начнем с простых правил:

- Отношения межу сущностями могут быть двусторонними и односторонними.
- У двустороннего отношения есть как прямая сторона (сторона владельца), так и обратная сторона.
- У односторонних отношений есть только прямая сторона.
- Именно прямая сторона отношения непосредственно влияет на все изменения, которые будут внесены в него в процессе работы приложения.

Для **двусторонних** связей справедливы следующие правила:

- Обратная сторона отношения должна ссылаться на основную сторону с помощью атрибута *mappedBy*, который используется в аннотациях *OneToOne*, *OneToMany* и *ManyToMany*. Этот атрибут указывает на поле сущности, которое является “владельцем” этого отношения (и это поле расположено на “противоположном” конце связи).
- И наоборот, прямая сторона двустороннего отношения ссылается на обратную сторону с помощью атрибута *inversedBy*, который также используется в аннотациях *OneToOne*, *ManyToOne* и *ManyToMany*. Этот атрибут указывает на поле сущности, которое является обратной стороной отношения.
- В отношениях типа *OneToMany* и *ManyToOne* именно “Many”-сторона является прямой стороной связи, поэтому на ней нельзя использовать атрибут *mappedBy* — он применяется только на обратной стороне.
- Для двусторонних отношений типа *OneToOne* прямой стороной связи является та, которая содержит соответствующий внешний ключ (он описывается аннотацией *@JoinColumn(s)*).

- В отношениях типа *ManyToMany* любая сторона может быть прямой. *(Непонятно: the side that defines the @JoinTable and/or does not make use of the mappedBy attribute, thus using a default join table.)*

Несколько запутанно, правда? На самом деле все не так сложно. Самое главное, запомните:

Именно прямая сторона связи определяет какие изменения в существующем отношении попадут в базу данных.

Чтобы понять это, давайте вспомним как работают двусторонние отношения в мире объектов. Возьмем два объекта, между которыми существует связь. На каждой стороне этой связи существует по ссылке, каждая из которых представляет эту самую связь, но изменять эти ссылки можно независимо друг от друга. Безусловно, в правильно спроектированном приложении вся семантика двусторонних связей должна полностью контролироваться разработчиком, это его ответственность. При использовании *Doctrine* ей просто нужно указать, какую из этих ссылок нужно хранить в базе данных, а какая там хранится не должна, потому что обе ссылки хранить невозможно, это абсурд. В этом и заключается концепция прямой и обратной связей.

Когда изменения в отношение вносятся только с обратной стороны связи, *Doctrine* их проигнорирует. Поэтому для двусторонних отношений нужно всегда обновлять обе стороны (ну или с точки зрения Doctrine, хотя бы прямую сторону). Сторона владельца в двусторонней связи — эта та точка, в которой находится условный наблюдатель от *Doctrine*, анализирующий связь, именно так она определяет текущее состояние связи и сам факт ее изменения (например, есть ли необходимость обновить ее в базе данных).

> Концепция прямой и обратной сторон является одним из краеугольных камней технологии *ORM* и к предметной области вашего приложения она не имеет отношения. Поэтому то, что в вашем приложении понимается под стороной владельца, в терминах *Doctrine* может трактоваться иначе. И на самом деле это не играет роли.

5.2. Коллекции

В примерах к этой главе при рассмотрении связей типа “ко многим” мы будет использовать специальный интерфейс *Collection*и соответствующую ему реализацию *ArrayCollection*, которая определена в пространстве имен *Doctrine\Common\Collections*. Для чего это нужно и почему нельзя использовать простые массивы? К сожалению, массивы в *PHP*, конечно, удобны во многих случаях, но с их помощью нельзя

полноценно представлять наборы объектов бизнес-логики, особенно вне контекста *ORM*. Причина состоит в том, что стандартные массивы *PHP* не могут быть прозрачно расширены для работы с продвинутыми фишками *ORM*. Классы и интерфейсы, которые лежат ближе всего к концепции коллекции, это *ArrayAccess* и *ArrayObject*, но пока экземпляры этих классов не смогут применятся в тех же конструкциях, где применяются обычные массивы (возможно, это будет сделано в *PHP6*), эффективность их будет ограничена. В принципе, можно использовать и *ArrayAccess* вместо *Collection*, ведь интерфейс *Collection* расширяет *ArrayAccess*, но это не даст вам нужной гибкости работы с коллекциями, потому что*API ArrayAccess* весьма примитивен (и это сделано специально), и, что более важно, нельзя будет передать эту коллекцию в всякие *PHP* функции, что делает работу с ней очень сложной.

> Интерфейс *Collection* и класс *ArrayCollection*, как и все в пространстве имен *Doctrine*, не относится ни к компоненту *ORM*, ни к *DBAL*. Это просто обычный *PHP* класс, не имеющий никаких внешних зависимостей за исключением разве что *PHP* и библиотеки *SPL*. Использование этого класса в приложении не требует его непосредственного контактирования со слоем, управляющим хранением данных (persistent layer). Класс *Collection*, как и все в модуле *Common*, не являются частью этого слоя. Вы можете вообще удалить *Doctrine*, оставив один этот класс, и весь код, его использующий продолжит нормально функционировать.

5.3. Параметры отображения по-умолчанию

Прежде чем мы начнем описывать все возможные отображения для связей, вам следует уяснить следующее. В процессе описания связей будут применяться аннотации *@JoinColumn* и *@JoinTable*. Они определяют какие столбцы в БД будут непосредственно отвечать за связь. Эти аннотации являются опциональными и имеют значения по умолчанию. Для отношения типов *OneToOne* или *ManyToOne*, используются следующие дефолтные значения:

```
name: "_id"
referencedColumnName: "id"
```

Давайте для примера рассмотрим такое отображение:

PHP

```
/** @OneToOne(targetEntity="Shipping") */
```

```
private $shipping;
```

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
```

Это абсолютно тоже самое, что и следующий, более навороченный вариант:

PHP

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

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
```

Конструкция *@JoinTable*, используемая для отображения связей *ManyToMany* имеет аналогичные значения по умолчанию:

PHP

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

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
```

В более полной нотации это выглядит так:

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

```
<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>

                    <join-column id="Group_id" referenced-column-name="id" />

            </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
```

В этом варианте имя таблицы, используемой для связи по умолчанию соответствует неполным именам участвующих в отношении классов, разделенных символом подчеркивания. Имена столбцов в этой таблице по умолчанию

складывается из неполного имени целевого класса с суффиксом "**_id**". Параметр **referencedColumnName** по умолчанию всегда равен "**id**", это справедливо как для отношений "один к одному", так и для "многие к одному".

Если вас устраивают значения по-умолчанию можно не писать лишнего кода.

5.4. Инициализация коллекций

При работе с полями, содержащими коллекции сущностей стоит быть внимательным. Допустим, у нас есть сущность *User*, которая содержит коллекцию групп:

```
<?php

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

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

Если рассматривать этот код отдельно, то видно, что поле *$groups* является только экземпляром класса *Doctrine\Common\Collections\Collection*, и пользователь может запросить его с помощью соответствующего метода. Однако, сразу после создания объекта *User* поле *$groups*, очевидно, будет иметь значение *NULL*.

Такие поля нужно заранее инициализировать в конструкторе пустыми объектами *ArrayCollection*:

```
<?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;
    }
}
```

Вот. И теперь следующий код будет нормально работать даже если сущность еще не связана с менеджером *EntityManager*:

```
<?php

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

5.5. Валидация отображений в различных средах (Runtime и Development)

По причинам, связанным с производительностью *Doctrine 2* не производит полную валидацию связи, т.е. проверку на предмет того правильно ли она мэппится на схему базы данных. Нужно самостоятельно проверять корректно ли настроена та или иная связь. Сделать это через командную строку:

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

Либо выполнить валидацию вручную:

```
<?php

use Doctrine\ORM\Tools\SchemaValidator;

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

if (count($errors) > 0) {
    // Lots of errors!
    echo implode("\n\n", $errors);
```

```
}
```

Если с отображением что-то не так, массив *$errors* будет содержать сообщения об ошибках. Единственный параметр, валидация которого не производится, это *referencedColumnName*. Он должен всегда равняться первичному ключу, иначе *Doctrine* вообще не будет работать.

Основная ошибка заключается в использовании обратного слеша в полном имени класса. Когда вы записываете это имя в виде строки (например в настройках отображения) обратный слеш в начале строки нужно убрать. Для обратной совместимости *PHP* делает это с помощью функции *get_class()* или с помощью механизма рефлексии.

5.6. Отношения “один к одному”, односторонние

Односторонние связи типа “один к одному” являются, наверное, самыми распространенными. Вот вам пример: сущность*Product* (товар) имеет один объект *Shipping* (отгрузка товара). При этом в *Shipping* нет ссылки обратно на *Product*, поэтому отношение и называется односторонним: *Product* → *Shipping*.

PHP

```
/** @Entity */
class Product
{
    // ...

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

   // ...
}

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

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
```

Обратите внимание, что использовать аннотацию @JoinColumn здесь не обязательно, т.к. значение по умолчанию дадут то же результат.

Итоговая схема MySQL будет выглядеть так:

```
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);
```

5.7. Отношения “один к одному”, двусторонние

В качестве примера возьмем отношения между объектами *Customer* (заказчик) и *Cart* (корзина). Смотрите, у *Cart* есть обратная ссылка на *Customer*, поэтому эта связь является двусторонней:

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
```

Обратите внимание, что использовать аннотацию *@JoinColumn* здесь не обязательно, т.к. значение по-умолчанию дадут то же результат.

Итоговая схема *MySQL* будет выглядеть так:

```
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);
```

Посмотрите как определен внешний ключ на прямой стороне отношения — таблице *Cart*.

5.8. Отношения “один к одному” со ссылкой на себя же

Такие отношения в *Doctrine* реализовываются весьма просто:

```
/** @Entity */
class Student
{
    // ...
```

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

    // ...
}
```

Обратите внимание, что использовать аннотацию *@JoinColumn* здесь не обязательно, т.к. значение по умолчанию дадут то же результат.

Итоговая схема *MySQL* будет выглядеть так:

```
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);
```

5.9. Отношения “один ко многим”, односторонние, с использованием @JoinTable

Односторонние связи типа “один ко многим” можно определять через подсоединяемую таблицу. С точки зрения Doctrine это выглядит как одностороннее отношение “многие ко многим”, где у одной из подсоединяемых колонок указан флаг уникальности, это и обеспечивает функционирование подобно отношениям “один ко многим”. Следующий пример описывает сказанное:

PHP

```
<?php

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

    /**
```

```
     * @ManyToMany(targetEntity="Phonenumber")
     * @JoinTable(name="users_phonenumbers",
     * joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     * inverseJoinColumns={@JoinColumn(name="phonenumber_id", referencedColumnName="id",
 unique=true)}
     * )
     */
    private $phonenumbers;

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

    // ...
}

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

XML

```
<doctrine-mapping>
    <entity name="User">
        <many-to-many field="phonenumbers" target-entity="Phonenumber">
            <join-table name="users_phonenumbers">
                <join-columns>
                    <join-column name="user_id" referenced-column-name="id" />
                </join-columns>

                    <join-column name="phonenumber_id" referenced-column-name="id"
 unique="true" />

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

YAML

```
User:
    type: entity
    manyToMany:
```

```
        phonenumbers:
            targetEntity: Phonenumber
            joinTable:
                name: users_phonenumbers
                joinColumns:
                    user_id:
                    referencedColumnName: id
                inverseJoinColumns
                    phonenumber_id:
                        referencedColumnName: id
                        unique: true
```

Описанные отношения работают только с использованием аннотации *@ManyToMany* совместно с ограничителем *unique*.

Итоговая схема *MySQL* будет выглядеть так:

```
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);
```

5.10. Отношения “многие к одному”, односторонние

Отношение типа “многие к одному” определяются следующим образом:

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
```

> Обратите внимание, что использовать аннотацию *@JoinColumn* здесь не обязательно, т.к. по умолчанию и так будут использоваться колонки *address_id* и *id*. Можно их и не указывать.

Итоговая схема MySQL будет выглядеть так:

```
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);
```

5.11. Отношения “один ко многим”, двусторонние

Двусторонние отношения вида “один ко многим” весьма распространены. Следующий пример показывает их реализацию на примере классов *Product* и *Feature*:

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>
```

Обратите внимание, что использовать аннотацию_ @JoinColumn_ здесь не обязательно, т.к. значение по умолчанию дадут то же результат.

Итоговая схема *MySQL* будет выглядеть так:

```
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);
```

5.12. Отношения “один ко многим” со ссылкой на себя

Пример показывает как настроить иерархию объектов Category с помощью отношения, ссылающегося на само себя. Этот подход позволяет реализовать иерархию категорий, в терминах БД называемой “списком смежных вершин”.

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
```

Обратите внимание, что использовать аннотацию *@JoinColumn* здесь не обязательно, т.к. значение по умолчанию дадут то же результат.

Итоговая схема *MySQL* будет выглядеть так:

```
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);
```

5.13. Отношения “многие ко многим”, односторонние

В реальных преложениях отношения типа “многие ко многим” встречаются реже. Следующий пример показывает как они определяются на примере сущностей *User* и *Group*:

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 name="user_id" referenced-column-name="id" />
                </join-columns>

                    <join-column name="group_id" referenced-column-name="id" />

            </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
```

Итоговая схема MySQL будет выглядеть так:

```
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);
```

> Так почему же такие связи реже встречаются в повседневной жизни? Все дело в том, что часто вам нужно привязать к связи какие-то дополнительные атрибуты, для чего под эту связь потребуется создать отдельный класс (связь*ManyToMany* это сделать не позволяет, здесь лишь будет создана таблица с двумя столбцами.) И, как следствие, связь “многие ко многим” в явном виде исчезает, а вместо нее появятся уже две связи — “один ко многим” и “многие к одному”, связывающие между собой три отдельных класса.

5.14. Отношения “многие ко многим”, двусторонние

Эти отношения аналогичны описанным выше, но они двусторонние:

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 name="user_id" referenced-column-name="id" />
                </join-columns>
```

```
                    <join-column name="group_id" referenced-column-name="id" />

            </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
```

Итоговая схема базы данных будет такая же как в предыдущем примере для односторонней связи.

5.14.1. ??<Picking Owning and Inverse Side>

Для связей “многие ко многим” можно указать какая сущность представляет прямую, а какая обратную сторону связи. Чтобы вам как разработчкику было проще определиться с тем, какая из сущностей больше подходит на роль прямой стороны связи, используйте следующее правило. Просто ответьте на вопрос, какая из сущностей отвечает за управление соединением, и это и будет прямая сторона.

Для примера возьмем две сущности: *Article* (статья) и *Tag* (тег). Всякий раз, когда вам нужно связать эти две сущности, в большинстве случаев именно *Article* будет отвечать за эту связь. И всякий раз при создании новой статьи, вам нужно буде соединить ее с существующими или новыми тегами. HTML-форма, отвечающая за создание статей вероятно так и работает, позволяя непосредственно указывать теги. Вот почему в качестве прямой стороны нужно выбрать *Article*, ваш код в этом случае будет более понятен, т.к. вы создаете модель в *Doctrine* в соответствии с тем, как эта связь функционирует в реальной жизни:

```
<?php

class Article
{
    private $tags;

    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;
    }
}
```

Это позволит разместить механизм добавления тегов на Article-стороне связи:

```
<?php

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

5.15. Отношения “многие ко многим” со ссылкой на себя

Да, они могу ссылаться на сами себя. Типичный сценарий выглядит так: у пользователя *User* есть друзья, при этом целевая сущность этого отношения это тоже *User*, таким образом имеет место ссылка на самого себя. В этом примере используется двусторонняя связь: у *User* есть поле *$friendsWithMe* и поле *$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();
    }

    // ...
}
```

Схема БД:

```
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);
```

5.16. Сортировка коллекций в связях “To-Many”

Во многих случаях при запросе сущности из БД вам нужно получать коллекции в уже отсортированном виде. Чтобы сделать это нужно определить для коллекции аннотацию *@OrderBy*. В этой аннотации указывается специальное DQL-выражение, которое будет добавляться ко всем запросам к этой коллекции. Описать *@OrderBy* для аннотаций *@OneToMany* или *@ManyToMany*можно так:

```
<?php

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

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

DQL должен состоять только из “чистых” имен полей без кавычек, а также опционального параметра ASC/DESC. Если нужна сортировка по нескольким полям, они разделяются запятой. Имена столбцов в этом выражении должны существовать в классе*targetEntity*, который описывается в аннотациях *@ManyToMany* и *@OneToMany*.

Семантику использования этой функции можно описать так:

- *@OrderBy* выступает в роли неявного выражения *ORDER BY*, который будет явно добавляться к запросу при выборке набора.
- Все такие коллекции всегда будут загружаться уже упорядоченными.

- Чтоб сильно не влиять на работу БД этот неявный *ORDER BY* добавляется к запросу только если коллекция выбирается явно с подсоединением (fetch joined).

Для вышеприведенного примера следующий DQL-запрос не будет добавлять *ORDER BY*, потому что сущность **g** здесь не присоединяется к запросу явно:

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

Однако следующий пример:

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

Будет автоматически переписан в:

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

И поменять порядок, явно указав его в DQL, нельзя:

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

Это будет автоматически переписано в:

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

Глава 6. Отображения и наследование

6.1. Отображение родительских классов

Так называемый отображаемый родительский класс это абстрактный либо обычный класс, который содержит в себе состояние сущности и другую информацию, связанную с отображением, передавая все это путем наследования своим производным классам, при этом он сам сущностью не является. Обычно такой класс может быть полезен когда необходимо задать состояние и информацию, связанную с отображением, которая является общей сразу для нескольких сущностей.

Такие классы, подобно обычным, могут быть вставлены где-то посередине цепочки наследования.

> Кроме того, такой класс сам не может являться сущностью, в нем не будут работать запросы (query), а все отношения, в которых он участвует должны быть односторонними (со стороной владельца). Это означает, что связи типа “один ко многим” в нем невозможны в принципе. Кроме того, связи типа “многие ко многим” возможны лишь в случае, если это родительский класс в данный момент используется только одной сущностью. Для более полной поддержки наследования, следует использовать одно-табличное (single table) или наследование “класс-таблица” (joined table.) О них речь пойдет ниже.

Пример:

```
<?php

/** @MappedSuperclass */
class MappedSuperclassBase
{
    /** @Column(type="integer") */
    private $mapped1;

    /** @Column(type="string") */
    private $mapped2;

    /**
```

```
     * @OneToOne(targetEntity="MappedSuperclassRelated1")
     * @JoinColumn(name="related1_id", referencedColumnName="id")
     */
    private $mappedRelated1;

   // ... more fields and methods
}

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

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

   // ... тут идут другие поля и методы
}
```

DDL (Data Definition Language) для такой схемы будет выглядеть примерно так (для *SQLite*):

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

Как видите, здесь присутствует только одна таблица и описывает она дочерний класс. При этом все отображение автоматически было наследовано от класса-родителя так, как будто оно было непосредственно определено в классе-потомке.

6.2. Наследование с единой таблицей (Single Table Inheritance)

Наследование с единой таблицей[1] это стратегии наследования отображений, в которой все классы в иерархии отображаются на одну единственную таблицу в базе данных. И

[1] http://martinfowler.com/eaaCatalog/singleTableInheritance.html

чтобы определить какая запись каком классу соответствует, используется специальный столбец, называемый *столбцом дискриминатора*.

Пример:

```
<?php

namespace MyProject\Model;

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

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

Что здесь стоит отметить:

- В самом верхнем классе в иерархии должны быть указаны маркеры *@InheritanceType*, *@DiscriminatorColumn* и*@DiscriminatorMap*.
- *@DiscriminatorMap* определяет какие значения столбца дискриминатора каким классам соответствуют. Например, значение **“person”** говорит о том, что запись имеет тип *Person*, а **“employee”** соответствует типу *Employee*.
- Названия классов в карте дискриминатора можно полностью не указывать, если они принадлежат одному пространству имен, что и класс, к которому эта карта будет применена.

6.2.1. Нюансы при проектировании

Вышеприведенный подход отлично работает когда иерархия типов проста и постоянна. Но когда в иерархию добавляется новый тип и дополнительные поля в существующие

родительские типы, это неизбежно порождает добавление новых столбцов в таблицу, что, в свою очередь, может негативно влиять на индексы и расположение столбцов в БД.

6.2.2. Производительность

Эта стратегия весьма эффективна как при запросах как ко всем так и к конкретным заданным типам. Дело в том, что здесь не требуется никакого соединения таблиц, идентификаторы типов фигурируют лишь в выражении *WHERE*. В частности, связи с типами, которые используют эту стратегию, работают очень быстро.

Основное правило производительности при одно-табличном наследовании звучит так: если целевая сущность в связях “многие к одному” или “один ко многим” использует стратегию *STI (Single Table Inheritance)*, то в плане производительности более предпочтительно, если она будет “оконечной” сущностью в иерархии наследования (например, у нее не будет потомков). В противном случае *Doctrine* **не сможет** создать для нее прокси-объекты, и, как следствие, будет **каждый раз** подгружать сущность целиком.

6.2.3. Замечания по схеме БД

При работе с устаревшей или самописной схемой БД (без использования SchemaTool) нужно убедиться, что все столбцы, которые не принадлежат корневой сущности (головному классу), но присутствуют в любой из сущностей-потомков, позволяют хранить значения *NULL*. Использование столбцов, имеющих ограничение *“NOT NULL”* допустимо использовать только в корневой сущности.

6.3. Наследование с таблицами классов (Class Table Inheritance)

Наследование с таблицами классов[2] представляет собой стратегию, в которой каждый класс в иерархии отображается в итоге на несколько таблиц: одну — его собственную, остальные — для всех родительских классов. При этом таблица производного класса будет связана с таблицей класс-родителя спомощью внешнего ключа. В *Doctrine 2* эта стратегия реализована с применением стлбца дискриминатора для “верхней” таблицы в иерархии, потому что это самый простой способ, при котором можно применять полиморфные запросы.

Пример:

[2] http://martinfowler.com/eaaCatalog/classTableInheritance.html

```php
<?php

namespace MyProject\Model;

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

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

Что здесь стоит отметить:

- *@InheritanceType*, *@DiscriminatorColumn* и *@DiscriminatorMap* должны быть указаны в самом верхнем классе в иерархии сущностей.
- *@DiscriminatorMap* определяет какие значения столбца дискриминатора каким классам соответствуют. Например, значение **“person”** говорит о том, что запись имеет тип *Person*, а **“employee”** соответствует типу *Employee*.
- Названия классов в карте дискриминатора можно полностью не указывать, если они принадлежат одному пространству имен, что и класс, к которому эта карта будет применена.

> Когда вы не используете *SchemaTool* для генерации *SQL*, то имейте ввиду, что при отказе от наследования “класс-таблица” какая бы БД не применялась, будет задействовано свойство внешнего ключа “ON DELETE CASCADE”. Невыполнения этого требования приведет к появлению в ваше базе данных “мертвых записей”.

6.3.1. Нюансы при проектировании

Добавление нового типа в иерархию просто порождает включение новой таблицы в общую схему. Подтипы для заданного типа автоматически будут связаны с ним во

время выполнения. Кроме того, внесение изменений в любой тип сущности в иерархии путем добавления, редактирования или удаления полей затронет лишь таблицу, непосредственно связанную с данным типом. Так что данная стратегия дает хорошую гибкость, ведь, как уже было сказано, изменение любого типа оказывает влияние лишь на выделенные под него таблицы.

6.3.2. Производительность

Эта стратегия из-за своей сути при выполнении практически любого запроса требует нескольких операций *JOIN*, а это негативно сказывается на производительности. Особенно это касается больших таблиц и глубоких иерархий. Когда допустимо использование частичных объектов (partial objects), глобально или для конкретного запроса, то в этом случае запрос к любому типу не будет требовать подключения таблиц для его дочерних типов через *OUTER JOIN*, а это в свою очередь увеличить производительность. Но имейте ввиду, частичные объекты при доступе к полям, определенных в их дочерних типах, не будут подгружать сами себя, поэтому доступ к таким полям будет не безопасен.

Основное замечание, касающееся производительности при наследовании “класс-таблица” звучит так: если целевая сущность в связях “многие к одному” или “один ко многим” использует стратегию *CTI (Class Table Inheritance)*, то в плане производительности более предпочтительно, если она будет “оконечной” сущностью в иерархии наследования (например, у нее не будет классов-потомков). В противном случае *Doctrine* не сможет создать для нее прокси-объекты, и, как следствие, будет каждый раз подгружать сущность целиком.

6.3.3. Замечания по схеме БД

У каждой сущности в иерархии наследования “класс-таблица” все отображаемые поля должны являться колонками в базе данных применительно к своей сущности. Дополнительно, у каждой дочерней таблицы должен быть столбец *id*, определенный так же как и одноименный столбец в корневой таблице (за исключением последовательностей (sequences) и автоинкрементных полей). Кроме того, у каждой дочерней таблицы должен быть внешний ключ, являющийся указателем со столбца *id* этой таблицы на столбец *id* корневой таблицы, и поддерживающий каскадное удаление.

Глава 7. Работа с объектами

Эта глава поможет понять принципы, на которых строится работа компонентов *EntityManager* и *UnitOfWork*. *Unit of Work*(Единица работы) это что-то вроде транзакций на уровне объектов. При первом создании *EntityManager* или после вызова метода *EntityManager#flush()* неявно произойдет создание новой единицы работы. Коммит этой единицы (и создание новой) происходит при вызове *EntityManager#flush()*.

Вызвав метод *EntityManager#close()*, можно отменить все несохраненные изменения в *Unit of Work*.

> Здесь важно понимать следующее. Только сам *EntityManager#flush()* непосредственно выполняет запись в базу данных. Все остальные методы, такие как *EntityManager#persist($entity)* или *EntityManager#remove($entity)* лишь уведомляют *UnitOfWork* о последующем исполнении этих операций при вызове метода *flush()*. По сути они лишь подготавливают транзакцию, а flush() исполняет ее. Если не вызвать EntityManager#flush(), то все внесенные в контексте текущего запроса изменения будут утеряны.

7.1. Паттерн “Карта соответствия” (Identity Map)

Сущности, как нам уже известно, это объекты с некоторыми идентификаторами. Эти идентификаторы имеют концептуальное значение в приложении. Вы, наверное, знаете, что в CMS каждая статья или новость имеет свой уникальный идентификатор, именно за счет него можно отличить одну статью от другой, т.е. идентифицировать ее.

Возьмем следующий пример. В нем мы происходит поиск статьи с заголовком **“Hello World”** и идентификатором **1234**:

```
<?php

$article = $entityManager->find('CMS\Article', 1234);
$article->setHeadline('Hello World dude!');

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

В этом примере сущность *Article* дважды запрашивается менеджером сущностей, а между вызовами происходит ее модификация. *Doctrine* без разницы сколько раз будет

запрошен объект из менеджера сущностей. В данном примере *Article* с*ID* **1234** всегда будет существовать только в одном экземпляре. И не важно как вы получаете эту сущность: через метод *find*, через репозиторий или через *DQL*. Паттерн, который делает такое поведение возможным, называется картой соответствия*(identity map)*. *Doctrine* хранит в ней соответствие каждой сущности ее идентификатору, которые были полученные в результате запроса в PHP, и, таким образом, всегда возвращает вам одни и те же экземпляры.

В предыдущем примере в результате вывода на экран вы увидите сообщение **“Hello World dude!”**. Несложно проверить, что**$article** и **$article2** действительно указывают на один и тот же экземпляр:

```
<?php

if ($article === $article2) {
    echo "Yes we are the same!";
}
```

Иногда нужно полностью очистить карту соответствия менеджера сущностей для того, чтобы начать некоторую процедуру, так сказать, с чистого листа. Этот прием часто используется в юнит-тестах для того, чтобы заставить движок загружать объекты из базы данных, вместо поиска соответствий в карте. Очистить карту соответствий можно с помощью метода*EntityManager#clear()*.

7.2. Обход графа сущностей

Хотя Doctrine и позволяет осуществлять полное разделение вашей доменной модели, при обходе связей никогда не возникнет ситуации, когда какие-то объекты будут отсутствовать. Поэтому пробежаться по связям в ваших сущностях можно сколь угодно глубоко.

Возьмем следующий пример, в нем мы получаем единственную статью Article из менеджера сущностей.

```
<?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);
```

Здесь происходит получение экземпляра сущности *Article* с идентификатором **1** путем исполнения единственного запроса*SELECT*. После этого можно получить доступ к ее свойствам **author** и **comments**, а также объектам, которые они содержат.

Это достигается за счет использования паттерна “ленивая загрузка” (lazy loading). Вместо того, чтобы давать доступ сразу к экземпляру *Author* и коллекции comments *Doctrine* создаст специальные прокси-объекты, куда и перенаправит вас. И при первом доступе они обратятся к *EntityManager* и загрузят свое текущее состояние из базы данных. Сделано это для повышения производительности.

Процесс ленивой загрузки работает неявно, он скрыт от ваших глаз (но ничто не мешает отследить его при помощь отладки). Рассмотрим следующий код:

```
<?php

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

// запрос метода экземпляра сущности запускает ленивую загрузку
echo "Author: " . $article->getAuthor()->getName() . "\n";

// Проверим экзмеплар
if ($article->getAuthor() instanceof User) {
    // Прокси для класса User реализован в классе UserProxy
}

// Запрос комментарие через итерацию также запускает ленивую загрузку,
```

```
// возвращая ВСЕ комментарии этой статьи из базы данных
// при этом используется один SELECT запрос
foreach ($article->getComments() AS $comment) {
    echo $comment->getText() . "\n\n";
}

// Article::$comments имеет интерфейся Collection
 // Но при этом не соответствует интерфейсу ArrayCollection
if ($article->getComments() instanceof \Doctrine\Common\Collections\Collection) {
    echo "This will always be true!";
}
```

Сгенерированные прокси классы выглядят как показано ниже. Реально такой класс переопределяет все общедоступные методы подобно переопределению метода *getName()*, как показано ниже:

```
<?php

class UserProxy extends User implements Proxy
{
    private function _load()
    {
        // lazy loading code
    }

    public function getName()
    {
        $this->_load();
        return parent::getName();
    }
    // .. other public methods of User
}
```

> При обходе графа те его части, которые используют ленивую загрузку, будут порождать много SQL-запросов, и не очень хорошо если это будет происходить слишком часто. Вместо этого используйте DQL, он позволяет делать все за один запрос, агрегируя все сущности в графе в один запрос, это гораздо эффективней.

7.3. Сохранение сущностей

Сохранить сущность можно с помощью метода *EntityManager#persist($entity)*. После осуществления над сущностью операции сохранения, она получает статус **MANAGED**, который означает, что отныне она персистирована (готова к сохранению в БД) и

управляется менеджером сущностей *EntityManager*. После этого персистированная сущность может быть синхронизирована с базой данных вызовом метода *EntityManager#flush()*.

> Передача сущности методу persist не запусукает на исполнение SQL запросы INSERT. Doctrine придерживается транзакционной модели выполнения, которая означает, что SQL-запросы будут придержаны до вызова метода EntityManager#flush(). И только после его вызова они будут выполнены, т.о ваши объекты будут синхронизированы с базой данных через единую транзакцию, что сохранит целостность операции.

Пример:

```
<?php

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

> После вызова метода *flush* всем идентификаторам и первичным ключам гарантированно будут присвоены значения. Нельзя полагаться на то, что значения они получат сразу после вызова метода persist. Справедливо и обратное — нельзя полагаться на то, что идентификаторам **не будут** присвоены значения после неудачного вызова метода flush.

Семантика операции персистенции (persist operation), применительно к сущности X, заключается в следующем:

- Если *X* является свежесозданной (**NEW**) сущностью, она станет управляемой (**MANAGED**). Сущность X попадет в базу данных после операции *flush()*;
- Если *X* является **MANAGED** сущностью, она будет проигнорирована методом *persist()*. Однако, *persist()* каскадно пройдет по всем сущностям, на которые ссылается *X*, если отношения между *X* и этими сущностями имеют флаги**cascade=PERSIST** или **cascade=ALL**.
- Если *X* является удаленной (**REMOVED**) сущностью, она снова станет управляемой (**MANAGED**).
- Есил *X* является отсоединенной (**DETACHED**) сущностью, то метод *flush()* выбросит исключение.

7.4. Удаление сущностей

Удалить сущность из постоянного хранилища можно с помощью передачи ее методу *EntityManager#remove($entity)*. При этом сущность будет помечена как **REMOVED**, что означает, что она будет удалена их хранилища при следующем вызове метода *EntityManager#flush()*.

> Тут ситуация аналогичная методу persist() — вызов метода remove() фактически ничего не удаляет (запросы DELETE не исполняются). Удалена сущность будет лишь после вызова EntityManager#flush(). Это значит, что к запланированным на удаление сущностям по прежнему можно обращаться посредством запросов, и они совершенно спокойно могут появляться в результатах запросов. Для более подробной информации читайте разделhttp://odiszapc.ru/doctrine/working-with-objects/#871_Effects_of_Database_and_UnitOfWork_being_Out-Of-Sync[Database and UnitOfWork Out-Of-Sync].

Пример:

```
<?php

$em->remove($user);
$em->flush();
Семантика операции удаления, применительно к сущности _X_, заключается в следующем:
```

- Если *X* является свежесозданной (**NEW**) сущностью, она будет проигнорирована операцией удаления. Однако, *remove()*каскадно пройдет по всем сущностям, на которые ссылается *X*, если отношения между *X* и этими сущностями имеют флаги **cascade=REMOVE** или **cascade=ALL**
- Если *X* является существующей **MANAGED** сущностью, она будет помечена как **REMOVED**.
- Если *X* является удаленной (**REMOVED**) сущностью, она будет проигнорирована операцией удаления.
- Есил *X* является отсоединенной (**DETACHED**) сущностью, будет выброшено исключение *InvalidArgumentException*.
- Сущность, помеченная как **REMOVED** будет удалена из базы данных при следующем вызове метода *flush()*.

После того как сущность будет удалена, ее текущее состояние (ее поля) останется таким же как до удаления, за исключением сгенерированных идентификаторов.

Помимо этого, при удалении сущности будут также удалены все связанные с ней записи во всех *“many-to-many”* таблицах, которые указывают на нее. Произойдет это или нет, зависит от значения атрибута *onDelete _в аннотации _@joinColumn*. Тут два варианта: либо *Doctrine* вызовет отдельные *DELETE* запросы для каждой записи в связанных таблицах, либо удаление произойдет на основе внешних ключей при **onDelete=”CASCADE”**.

Удалить сам объект вместе со всеми связанными с ним другими объектами можно несколькими способами, которые значительно отличаются по производительности.

Если связь помечена флагом **CASCADE=REMOVE** *Doctrine 2* целиком возьмет ее обработку на себя. Если эта связь представляет собой одиночную сущность, она просто будет передана методу *EntityManager#remove()*. Если связь представляет собой коллекцию, *Doctrine* сделает итерацию по ней и для каждого элемента вызовет *EntityManager#remove()*. В обоих случаях каскадное удаление работает рекурсивно. Для графа, в котором много объектов, выполнять такую операцию будет весьма накладно.

С помощью оператора *DQL DELETE* можно удалить сразу несколько сущностей без необходимости их обработки. Это больше подходит для удаления больших графов объектов.

Использование семантики внешних ключей **onDelete=”CASCADE”** заставляет базу данных саму произвести удаление всех связанных объектов. Хоть эта стратегия и более сложна, и тут можно конкретно напортачить, но работает это очень быстро, да и сам подход удобен. Однако, следует знать, что использовании первой стратегии (**CASCADE=REMOVE**) заставляет Doctrine полностью исключить из анализа все опции внешних ключей **onDelete=CASCADE**, потому что в данной ситуации *Doctrine* будет явно запрашивать и удалять все связанные сущности.

7.5. Отсоединение сущностей

Отсоединить сущность от *EntityManager* можно как с помощью передачи ее методу *EntityManager#detach($entity)* так и при каскадной операции отсоединения. После того как сущность была отсоединена все изменения, внесенные в нее (включая операцию удаления) не будут синхронизированы с базой данных.

Doctrine, в свою очередь, не будет пытаться хранить ссылки на отсоединенную сущность.

Пример:

```
<?php

$em->detach($entity);
```

Семантика операции отсоединения применительно к сущности *X*, заключается в следующем:

- Если *X* является существующей **MANAGED** сущностью, при отсоединении она будет помечена как **DETACHED**. При этом*detach()* каскадно пройдет по всем сущностям, на которые ссылается *X*, если отношения между *X* и этими сущностями имеют флаги **cascade=DETACH** или **cascade=ALL**. Сущности, которые до этого ссылались на *X* продолжат на нее ссылаться.
- Если *X* является только что созданной (**NEW**) или уже отсоединенной сущностью, она будет проигнорирована операцией отсоединения.
- Если *X* является удаленной (**REMOVED**) сущностью, операция отсоединения каскадно пройдет по всем сущностям, на которые ссылкается *X*, если отношения между *X* и этими сущностями имеют флаги **cascade=DETACH** или **cascade=ALL**. Сущности, которые до этого ссылались на *X* продолжают на нее ссылаться.

Существует несколько ситуаций, при которых сущность будет отсоединена автоматически без вызова соответствующего метода:

- При вызове метода *EntityManager#clear()* все сущности, находящиеся под управлением менеджера сущностей будут отсоединены от него.
- При сериализации сущности. Сущность, полученная при последующей де-сериализации станет отсоединенной. (Это касается всех сущностей, которые были сериализованы и размещены в каком-нибудь кеше, например, при кешировании результатов запросов).

Вообще говоря, отсоединять сущности нужно будет не так часто как, например, удалять и сохранять их.

7.6. Слияние сущностей

Под объединением имеется ввиду слияние (обычно отсоединенных) сущностей в контекст менеджера *EntityManager*, в результате чего эти сущности снова попадут в общий граф и станут управляемыми (**MANAGED**). Чтобы объединить сущность

с графом нужно воспользоваться методом *EntityManager#merge($entity)*. Состояние переданной ему сущности будет объединено в управляемую копию данной сущности, и эта копия будет впоследствии возвращена в качестве результата вызова метода.

Пример:

```
<?php

$detachedEntity = unserialize($serializedEntity); // какая-то отсоединенная
 сущность
$entity = $em->merge($detachedEntity);
// $entity теперь ссылается на управляемую копию, возвращенную методом merge().
// EntityManager $em теперь как обычно может работать с сущностью $entity.
```

> Когда необходимо произвести сериализацию или десериализацию сущности, нужно сделать все ее свойства защищенными (protected), но не закрытыми (private). Причина в том, что когда сериализуется класс, который до этого был экземпляром прокси-объекта, его закрытые члены не будут участвовать в сериализации, о чем PHP выдаст сообщение типа Notice.

Семантика слияния применительно к сущности *X* заключается в следующем (без бутылки не разобраться):

- Если *X* является отсоединенной сущностью, ее состояние будет скопировано на уже существующий экзмепляр сущности*X'* с тем же идентификатором.
- Если *X* новая сущность, будет создана ее управляемая (**MANAGED**) копия *X'*, и состояние *X* будет скопировано в этот управляемый экземпляр.
- Если *X* экземпляр удаленной сущности, будет выброшено исключение *InvalidArgumentException*.
- Если *X* является **MANAGED** сущностью, операция слияния проигнорирует ее. Однако, операция *merge()* каскадно пройдет по всем сущностям, на которые ссылкается *X*, если отношения между *X* и этими сущностями имеют флаги**cascade=MERGE** или **cascade=ALL**.
- Для всех сущностей *Y*, на которые ссылается *X*, имеющая флаг **cascade=MERGE** или **ALL**, *Y* будет рекурсивно смерджена в *Y'*. Для всех таких *Y*, на которые ссылаются *X*, *X'* будет ссылаться на *Y'*. (Обратите внимание, если *X* это управляемая (**MANAGED**) сущность, то экземпляр *X* это одно и тоже что и *X'*).
- Если *X* сущность, которая после слияния перешла в *X'*, и в ней была ссылка на другую сущность *Y*, и при этом не был задан флаг **cascade=MERGE** или **cascade=ALL**, то

при попытке перехода по такой связи от *X'* управление будет передано по ссылке на управляемый объект *Y'*, имеющий такой же постоянный идентификатор, что и *Y*.

Операция слияния может выбросить исключение *OptimisticLockException* в случае, если сущность была смерджена с использованием оптимистичной блокировки *(optimistic locking)* чеерез поле с версией и при этом версии смердженной сущности и ее управляемой копии не совпадают. Обычно это означает, что после операции отсоединения сущность была каким-то образом изменена.

В отличии от операции сохранения и удаления, слияние применяется не так часто. Типичный сценарий, где она может пригодится — когда нужно заново прикрепить сущности к *EntityManager* после того как они были получены из какого-нибудь кеша (и как следствие до этого были отсоединены).

> Если необходимо сделать множественное слияние сущностей, которые разделют между собой определенные части графа, следует вызвать метод *EntityManager#clear()* между последовательными вызовами метода*EntityManager#merge()*. В противном случае вы можете получить в базе данных несколько копий одного и того же объекта, но с разными идентификаторами.
>
> При загрузке отсоединенных сущностей их кеша, если вам не нужно сохранять и удалять их или вы хотите работать с ними без участия сервисов вроде EntityManager, слияние как таковое вам не нужно. Например ничто не мешает передать отсоединенные объекты из кеша напрямую в представление (view).

7.7. Синхронизация с базой данных

Состояние сохраненных сущностей будет синхронизировано с базой данных при помощи операции *EntityManager#flush()*, которая, в свою очередь, вызовет методы компонента *UnitOfWork*. Синхронизация включает в себя запись в базу данных любых изменений в сущностях и их отношениях. Как описано в главе “Отображение связей”[1], двусторонние отношения сохраняются на основе ссылок, исходящих со стороны владельца.

Когда происходит вызов метода *EntityManager#flush()*, *Doctrine* анализирует все **MANAGED**-, **NEW**- и **REMOVED**- сущности и выполняет соответствующие этим состояниям операции.

[1] http://odiszapc.ru/doctrine/association-mapping/

7.7.1. Effects of Database and UnitOfWork being Out-Of-Sync

Когда вы изменили состояние сущностей, вызовите метод *persist()* или *remove()* компонента *UnitOfWork* и база данных **will drive out of sync** (???). Но фактически синхронизированы они могут быть только путем вызова метода *EntityManager#flush()*. Этот раздел описывает **the effects of database and UnitOfWork being out of sync**.

- Сущности, которые запланированы на удаление, все еще могут быть запрошены из базы данных. Они могут быть получены в результате *DQL*-запросов или через репозитории, также они отображаются в коллекциях.
- Сущности, переданные методу *EntityManager#persist()* не будут появляться в результатах запросов.
- Состояние сущностей, которые были изменены, не будет перезаписано соответствующим их состоянием, уже хранящимся в базе данных. Так происходит потому, что карта соответствия_ (identity map)_ будет каждый раз обнаруживать, что создаваемая сущность уже существует и делать предположение, что это и есть ее наиболее свежая версия.

EntityManager#flush() никогда не вызывается Доктриной автоматически. Вы всегда должны вызывать его вручную.

7.7.2. Синхронизация новых (new) и управляемых (managed) сущностей

Операция flush(), применяемая по отношению к управляемым сущностям работает следующим образом:

- Если у сущности было изменено хотя бы одно поле, она будет синхронизирована в базу данных при помощи *SQL*-запроса*UPDATE*.
- Если сущность не изменялась, никаких *SQL*-запросов выполнено не будет.

По отношению к новой сущности *flush* работает так:

- Сущность будет синхронизирована в базу данных при помощи *SQL*-запроса *INSERT*. Для всех отношений у **NEW**- и **MANAGED**- сущностей, каждая связанная с ними сущность *X* будет обработана согласно следующим правилам:
- Если *X* была только что создана (**NEW**) и у нее настроена каскадность при сохранении, то *X* будет сохранена в базе данных.

- Если *X* была только что создана (**NEW**) и у нее **не была** задана каскадность при сохранении, будет выброшено исключение.
- Если *X* имеет статус **REMOVED** и у нее настроена каскадность при сохранении, будет выброшено исключение (потому что*Doctrine* попытается повторно сохранить *X*)
- Если *X* имеет статус **DETACHED** и у нее настроена каскадность при сохранении, будет выброшено исключение (семантика такая же как и при передаче *X* методу *persist()*).

7.7.3. Синхронизация удаленных сущностей

Если операция *flush* исполняется по отношению к **REMOVED**- сущности, то такая сущность будет удалена из базы данных. В этом случае при исполнении операции *flush* опции каскадности не применяются, т.к. каскадное удаление уже отработало при вызове метода *EntityManager#remove($entity)*.

7.7.4. Размер Unit of Work

Размер *Unit of Work* в определенный момент времени представляет собой ни что иное как число *MANAGED*-сущностей на этот момент.

7.7.5. Производительность операции flush

Насколько дорога операция *flush* зависит от двух факторов:

- Размер текущего “модуля работы” *UnitOfWork* в менеджере сущностей.
- Конфигурация политик отслеживания изменений *(change tracking policies)*

Размер *UnitOfWork* можно узнать следующим образом:

```
<?php

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

Размер в данном случае представляет собой количество обслуживаемых сущностей в модуле Unit of Work. И этот размер непосредственно виляет на производительность *flush()* за счет отслеживания изменений (см. раздел “Change Tracking Policies”) и потребления памяти, так что при разработке время от времени проверяйте эти параметры.

> Не нужно вызывать flush() при каждом изменении сущности или каждом вызове операций persist, remove, merge и т.д. Это только лишний раз снизит производительность вашего приложения. Вместо этого формируйте модули работы (units of work), которые и будут производить действия над вашими объектами, а когда закончите вызывайте flush(). При обработке одиночного HTTP запроса обычно достаточно не более двух вызовов flush().

7.7.6. Прямой доступ к Unit of Work

Получить прямой доступ к *Unit of Work* можно путем вызова метода *EntityManager#getUnitOfWork()*. Этот метод вернет экземпляр того *UnitOfWork*, который в данный момент используется менеджером сущностей.

```
<?php

$uow = $em->getUnitOfWork();
```

> Не оперировать UnitOfWork напрямую. Когда вы напрямую работаете с API UnitOfWork вы используете внтуренние мезанизмы, поэтому внимательно прочтите документацию к API.

7.7.7. Состояние сущности

Как уже было отмечено в архитектурном обзоре, сущность может находиться в одном из четырех состояний: **NEW**, **MANAGED**,**REMOVED** и **DETACHED**. Если вам нужно узнать текущее состояние в контексте соответствующего *EntityManager*, можно спросить об этом у лежащего в его основе компонента *UnitOfWork*:

```
<?php

switch ($em->getUnitOfWork()->getEntityState($entity)) {
    case UnitOfWork::STATE_MANAGED:
    // ...
    case UnitOfWork::STATE_REMOVED:
    // ...
    case UnitOfWork::STATE_DETACHED:
    // ...
    case UnitOfWork::STATE_NEW:
    // ...
}
```

Сущность находится в состоянии **MANAGED**, если она связана с *EntityManager* и при этом не имеет статус **REMOVED**.

Сущность находится в состоянии **REMOVED** после того как она была передана методу *EntityManager#remove()* до последующего вызова *EntityManager#flush()*. **REMOVED**-сущность будет все еще связана с менеджером до следующей операции *flush*.

Сущность находится в состоянии **DETACHED** если она присутствует в хранилище и у нее есть идентификатор, но в данный момент она не связана с *EntityManager*.

Сущность находится в состоянии **NEW** если она отсутствует в хранилище, у нее нет идентификатора и в данный момент она не связана с *EntityManager* (например она была создана оператором _new). _

7.8. Запросы

Осуществлять запросы к хранилищу сущностей в Doctrine 2 можно по-разному. Ниже представлены несколько способов, расположенных в порядке возрастания мощи и гибкости. Всегда старайтесь использовать самые простые способы, и только если они не подходят переходите к более навороченным вариантам.

7.8.1. По первичному ключу

Самый простой способ запроса объектов заключается в использовании идентификатора в качестве критерия. Делается это при помощи метода *EntityManager#find($entityName, $id)*. Вот пример:

```
<?php

// $em instanceof EntityManager
$user = $em->find('MyProject\Domain\User', $id);
```

Возвращаемое значение представляет собой найденный экземпляр сущности либо *NULL*, если ничего не найдено.

Вообще говоря, *EntityManager#find()* это просто сокращенная форма следующей записи:

```
<?php

// $em instanceof EntityManager
$user = $em->getRepository('MyProject\Domain\User')->find($id);
```

Метод *EntityManager#getRepository($entityName)* возвращает объект репозитория, который предоставлят много разных способов получения сущностей соответствующего типа. По умолчанию, экземпляр репозитория имеет тип*Doctrine\ORM\EntityRepository*. Но вы можете создавать свои типы репозиториев, мы позже покажем как это делается.

7.8.2. С простыми условиями

Когда нужно запросить одну или несколько сущностей с несколькими условиями, формирующих логическое умножение, используйте методы репозитория *findBy* и *findOneBy* как показано ниже:

```
<?php

// $em это екземпляр EntityManager

// Пользователи, которым 20 лет
$users = $em->getRepository('MyProject\Domain\User')->findBy(array('age' => 20));

// 20-летние с фамилией 'Miller'
$users = $em->getRepository('MyProject\Domain\User')->findBy(array('age'
 => 20, 'surname' => 'Miller'));

// Одиночный юзер по нику
$user = $em->getRepository('MyProject\Domain\User')->findOneBy(array('nickname'
 => 'romanb'));
```

Можно сделать и так:

```
<?php

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

Обратите внимание, здесь нужно передавать именно **ID** связанной сущность, а нее ее саму.

Метод *EntityRepository#findBy()* также допускает сортировку и срез результатов путем задания дополнительных параметров:

```
<?php
```

```
$tenUsers = $em->getRepository('MyProject\Domain\User')->findBy(array('age'
 => 20), array('name' => 'ASC'), 10, 0);
```

Если в качестве значений фильтра будет передан массив, Doctrine автоматически преобразует его в выражение *WHERE field IN (..):*

```
<?php

$users = $em->getRepository('MyProject\Domain\User')->findBy(array('age'
 => array(20, 30, 40)));
// Будет преобразовано в: SELECT * FROM users WHERE age IN (20, 30, 40)
```

EntityRepository позволяет использовать более короткую запись вызова методов при момощие магии **_call**. Нижеприведенные примеры вызова полностью эквивалентны:

```
<?php

// Ищем одного пользователя по его нику
$user = $em->getRepository('MyProject\Domain\User')->findOneBy(array('nickname'
 => 'romanb'));

// Тоже самое, но при помощи _call
$user = $em->getRepository('MyProject\Domain\User')->findOneByNickname('romanb');
```

7.8.3. “Жадная” загрузка

При запросе сущности, в которй имеются связи, настроенные как **EAGER**, они будут автоматически загружены вместе с запрашиваемой сущностью и будут сразу же доступны приложению.

7.8.4. “Ленивая” загрузка

Когда у вас есть **MANAGED**- сущность, можно прозрачно запросить любые ее связи, помеченные как **LAZY**, так, как будто они уже загружены в память. В соответствии с концепцией “ленивой загрузки” все связанные объекты будут автоматически загружены *Doctrine* при поптыке доступа к ним.

7.8.5. С помощью DQL

Doctrine Query Language — самый мощный и гибкий способ запроса объектов. *DQL* позволяет написать запрос к объектам на языке самих объектов. *DQL* понимает классы, поля, наследование и связи. Синтаксически *DQL* похож на *SQL*, но это не он.

DQL-запрос представлен экземпляром класса *Doctrine\ORM\Query*. Создается запрос с помощью метода*EntityManager#createQuery($dql)*. Пример:

```
<?php

// $em это экземпляр EntityManager

// Запрос всех пользователей, имеющих возраст от 20 до 30 лет включительно
$q = $em->createQuery("select u from MyDomain\Model\User u where u.age >= 20 and
 u.age <= 30");
$users = $q->getResult();
```

Заметьте, что этот запрос совершенно не знает о существующей схеме отношений, он оперирует лишь объектной моделью.*DQL* поддерживает как именованные, так и основанные на позиции параметры, множество функций, соединения, агрегации, дочерние запросы и много чего интересного. Более подробная информация о *DQL* будет рассмотрена в отдельной главе. Для программного построения запросов с условиями в Doctrine есть специальный класс *Doctrine\ORM\QueryBuilder*. Подробно построение запросов с помощью *QueryBuilder* будет рассмортено в главе *“QueryBuilder”*.

7.8.6. Через нативный SQL

В качестве альтернативы DQL можно использовать нативные *SQL*-запросы. Такие запросы пишутся вручную на *SQL* и настраиваются с помощью *ResultSetMapping*, которая описывает каким именно образом результат *SQL*-запроса будет трактоваться Доктриной. Подробно это будет рассмотрено в отдельной главе.

7.8.7. Пользовательские репозитории

Когда вы вызываете метод *EntityManager#getRepository($entityClass) EntityManager* возвращает вам стандартный репозиторий*Doctrine\ORM\EntityRepository*. Используя метаданные к аннотациям, XML или YAML, можно описать свой класс репозитория. Использование пользовательских репозиториев более предпочтительно в крупных приложениях, использующих тонны различных *SQL*-запросов, потому что запросы будут сгруппированы в одном месте.

```
<?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();
    }
}
```

Теперь можно обращаться к собственным репозиториям:

```
<?php

// $em это экземпляр EntityManager

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

Глава 8. Работа со связями

В *Doctrine* связи между сущностями выглядят так же как PHP выглядят ссылки на другие объекты и их коллекции. Но когда подобные структуры хранятся в базе данных нужно понимать следующие три вещи:

- Концепция прямой и обратной сторон связи[1] в двусторонних отношениях
- Если сущность удаляется из коллекции, это означает лишь удаление связи, но никак не удаление самой сущности. Связь представлена коллекцией входящих в нее сущностей, но не самими сущностями.
- Поля сущности, являющиеся коллекциями должны реализовывать интерфейс **Doctrine\Common\Collections\Collection**.

Все изменения, внесенные в связь в течение работы приложения не будут моментально синхронизироваться, это произойдет только после вызова метода **EntityManager#flush()**.

Чтобы описать все возможные варианты работы со связями мы подготовим специальный набор тестовых сущностей, на примере которых будут продемонстрированы разные способы работы со связями в *Doctrine*.

8.1. Тестовые сущности

В качестве примеров мы будем использовать простую систему, в которой есть Пользователи **(Users)** и Комментарии**(Comments)**. Посмотрите на нижеприведенный код, из него будет понятно что к чему:

```
<?php

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

    /*
     * Двусторонняя связь - множество пользователей имеют множество избранных
 комментариев (это сторона владельца)
     *
```

[1] http://odiszapc.ru/doctrine/association-mapping#61

```
     * @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;

    /*
     * Односторонняя связь - множество пользователей могут пометить множество
 комментариев как прочтенные
     *
     * @ManyToMany(targetEntity="Comment")
     * @JoinTable(name="user_read_comments",
     *   joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *   inverseJoinColumns={@JoinColumn(name="comment_id",
 referencedColumnName="id")}
     * )
     */
    private $commentsRead;

    /*
     * Двусторонняя связь - один ко многим (обратная сторона)
     *
     * @OneToMany(targetEntity="Comment", mappedBy="author")
     */
    private $commentsAuthored;

    /*
     * Односторонняя связь - многие к одному
     *
     * @ManyToOne(targetEntity="Comment")
     */
    private $firstComment;
}

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

    /*
     * Двусторонняя связь - множество комментариев добавлены в избранное множеством
 пользователей (обратная сторона)
     *
     * @ManyToMany(targetEntity="User", mappedBy="favorites")
```

```
     */
    private $userFavorites;

    /*
     * Двусторонняя связь - множество комментариев написано одним пользователем
 (сторона владельца)
     *
     * @ManyToOne(targetEntity="User", inversedBy="authoredComments")
     */
     private $author;
}
```

Для этих двух сущностей будет сгенерирована следующая схема в *MySQL* (определение внешних ключей опущено)

```
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;
```

8.2. Создание связи

Устанавливается связь между двумя сущностями просто. Вот примеры для односторонних отношений:

```
<?php
```

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

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

Рабочий код будет выглядеть как показано ниже (**$em** здесь это EntityManager):

```
<?php

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

// односторонняя связь "многие ко многим"
$comment = $em->find('Comment', $readCommentId);
$user->getReadComments()->add($comment);

$em->flush();

// односторонняя связь "многие к одному"
$myFirstComment = new Comment();
$user->setFirstComment($myFirstComment);

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

В случае двусторонних связей следует изменять поля у обоих сущностей:

```
<?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;
    }
}

// Многие ко многим
$user->getFavorites()->add($favoriteComment);
$favoriteComment->getUserFavorites()->add($user);

$em->flush();

// Многие к одному / один ко многим, двусторонняя связь
$newComment = new Comment();
$user->getAuthoredComments()->add($newComment);
$newComment->setAuthor($user);

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

Обратите внимание на внесенные изменения — у двусторонней связи обновляются обе стороны. Предыдущий пример с односторонними отношениями был проще.

8.3. Удаление связей

Удаление связи между двумя сущностями осуществляется аналогично. Сделать это можно двумя способами: по элементу или ключу. Примеры:

```
<?php

// Удаление по элементам
$user->getComments()->removeElement($comment);
$comment->setAuthor(null);

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

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

// Удаление по ключу
$user->getComments()->remove($ithComment);
$comment->setAuthor(null);
```

Для внесения этих изменений в базу данных нужно будет вызвать метод **$em→flush()**.

Заметьте, что в двусторонней связи всегда обновляются обе стороны. Односторонние связи в этом отношении проще. Также имейте ввиду, что если вы явно укажете тип параметра в методах_, например _*setAddress(Address $address)**, то PHP разрешит передачу этому методу значения *NULL** только если оно явно задано в качестве значения по умолчанию. В противном случае при удалении связи метод **setAddress(null)** потерпит неудачу. Если вам все-таки необходимо явно задать тип параметра**(type-hinting)**, то лучше создать специальный метод вроде **removeAddress()**. Такой подход улучшит инкапсуляцию класса, скрыв то, как этот класс будет обрабатывать ситуацию с отсутствующим адресом.

При работе с коллекциями имейте ввиду, что коллекция это, по сути, упорядоченная карта (подобно обычному массиву в PHP). Вот почему при удалении нужно указать индекс или ключ. Метод **removeElement** имеет сложность **O(n)**, т.к. работает через функцию **array_search**, где **n** — это размер карты.

> Doctrine на предмет обновлений всегда просматривает только сторону владельца связи, поэтому нет необходимости писать код, который будет обновлять коллекцию с обратной стороны. Это даст вам пару очков в производительности, т.к. не нужно будет лишний загружать эту коллекцию.

Очистить содержимое коллекции можно с помощью метода **Collections::clear()**. Следует помнить, что его использование в последующей операции **flush()** может привести к вызовам DELETE и UPDATE, которые не знают о сущностях, добавленных в коллекцию до этого. (Оригинал: **You should be aware that using this method can lead to a straight and optimized database delete or update call during the flush operation that is not aware of entities that have been re-added to the collection**).

Скажем, вы очистили коллекцию тегов с помощью **$post→getTags()→clear()**, а затем вызывали **$post→getTags()→add($tag)**, добавив новый тег. В этом случае движок ORM не сможет распознать тег, который был там до этого, и как следствие, произведет два различных вызова к базе данных.

8.4. Способы управления связями

На самом деле было бы отлично, если весь механизм работы со связями был спрятан внутри сущностей. Это привнесет целостность в архитектуру вашего приложения, ведь все детали касательно связей будут инкапсулированы в классе.

Нижеприведенный код демонстрирует соответствующие изменения в сущностях **User** и **Comment**:

```
<?php

class User
{
    //...
    public function markCommentRead(Comment $comment) {
        // Collections определяет интерфейс 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);
    }
}
```

Как вы могли заметить, методы **addUserFavorite** и **removeUserFavorite** не вызывают соответствующих методов **addFavorite** и**removeFavorite**, таким образом, двустороннее отношение, старого говоря, является незаконченным. Однако, если вы, наивно полагая, добавите вызов **addFavorite** в метод **addUserFavorite**, то получите бесконечный цикл. Как видите, работа с двусторонними связями в ООП не тривиальная задача, а инкапсуляция деталей работы с ними внутрь классов иногда бывает сложна.

> Чтобы добиться идеальной инкапсуляции коллекций, не следует возвращать их напрямую из метода**getCollectionName()**, вместо него используйте **$collection→toArray()**. Таким образом, пользователь сущности не сможет обойти определенную вами логику обработки связей. Пример:

```
<?php

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

Этот подход, однако, всегда приводит к предварительной инициализации коллекции, со всеми вытекающими отсюда проблемами с производительностью, хотя все зависит от размера коллекции. При работе с коллекциями больших размеров хорошей идеей будет полностью скрыть весь механизм считывания в методах репозитория (EntityRepository).

Не существует единственно верного способа работы с коллекциями. Все зависит, с одной стороны, от требований к вашим моделями, а с другой — от ваших предпочтений.

8.5. Синхронизация двусторонних коллекций

При работе со связями типа “многие ко многим” вы как разработчик, возможно, захотите, чтобы при редактировании коллекций они всегда оставались синхронизированы с обоих

сторон связи. Но Doctrine гарантирует согласование лишь при гидрации (при сохранении в БД), но не для вашего клиентского кода.

Давайте посмотрим с чем вы можете столкнуться на примере уже известной связки **User-Comment**:

```
<?php

$user->getFavorites()->add($favoriteComment);
// Здесь не происходит вызова $favoriteComment->getUserFavorites()->add($user);

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

Существует два способа решения этой проблемы: Игнорировать обновление обратной стороны связи в двусторонних отношениях, но никогда ничего не считывать оттуда при запросах, которые изменяют их состояние. При следующем запросе Doctrine соответствующим образом подготовит и согласует состояние связи. Осуществлять синхронизацию двусторонних коллекций при помощи соответствующих методов. **Reads of the Collections directly after changes are consistent then.**

8.6. Transitive persistence / Каскадные операции

Когда мы имеем дело с навороченным графом объектов, в котором переплетено множество связей, то операции сохранения, удаления, отсоединения и слияния отдельных сущностей могут оказаться весьма громоздкими. Поэтому Doctrine 2 обеспечивает механизм **transitive persistence** путем каскадного выполнения соответствующих операций. По умолчанию каскадность отключена.

Существуют следующие опции каскадности:

persist : Операции сохранения каскадно применяются к связанным сущностям.
remove : Аналогично для удаления.
merge : Аналогично для слияния.
detach : Аналогично для отсоединения.
all : Все вышеприведенные операции будут каскадно применены к связанным сущностям.

> Все каскадные операции производятся в оперативной памяти. Это значит, что непосредственно перед началом операции коллекции и

связанные с ними сущности загружаются в память, даже если у них настроена “ленивая загрузка”. При этом для каждой операции будут соответствующим образом запущенны обработчики событий жизненного цикла сущностей (коллбеки: **beforPersist**, **afterPersists** и т.д) если они есть. Однако, при больших размерах коллекций, размещение графа объектов в памяти может привести к проблемам с производительностью. Поэтому взвесьте все “за” и “против”, определите преимущества и узкие места каждой каскадной операции. Если вместо этого при удалении нужно использовать механизмы каскадности, предоставляемые базой данных, то каждую **join column** можно сконфигурировать с опцией **onDelete**. Читайте об этом в соответствующих главах, касающихся драйверов метаданных.

Следующий пример показывает расширение уже известной нам связки **User-Comment**. Предположим, что пользователь должен быть создан, когда он напишет свой первый комментарий:

```
<?php

$user = new User();
$myFirstComment = new Comment();
$user->addComment($myFirstComment);

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

Есть в этом примере удалить вызов **EntityManager#persist($myFirstComment)**, то код потерпит неудачу, даже если вы сохраните вашего нового Юзера, добавив к нему новый Комментарий. Все дело в том, что Doctrine 2 не обрабатывает каскадно сущности, которые были только что созданы и не прикреплены к менеджеру сущностей.

Вот аналогичный, но более сложный пример, в нем при удалении пользователя из системы удаляются все его комментарии:

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

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

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

Если не выполнить цикл и не пройтись по всем комментариям пользователя, Doctrine выполнит запрос **UPDATE** только для того, чтобы установить значения внешних ключей в **NULL**, таким образом после операции **flush()** из базы будет удален только сам пользователь.

Чтобы Doctrine корректно обрабатывала оба случая можно изменить свойство **User#commentsAuthored**, добавив к нему опции каскадности **“persist”** и **“remove”**:

```
<?php

class User
{
    //...
    /*
     * Bidirectional - One-To-Many (INVERSE SIDE)
     *
     * @OneToMany(targetEntity="Comment", mappedBy="author", cascade={"persist",
 "remove"})
     */
    private $commentsAuthored;
    //...
}
```

Хотя автоматическая каскадность весьма удобна, использовать ее нужно с осторожностью. Не присваивать каждой связи опцию**cascade=all**, это приведет лишь к снижению производительности. При активации каждую каскадную операцию Doctrine применит и к связи, будь она одиночной или коллекцией.

8.6.1. Persistence by Reachability: Cascade Persist

У операций каскадного сохранения существует дополнительная семантика. Если при вызове **flush()** Doctrine обнаружит в какой-нибудь из коллекций свежесозданные сущности **(NEW)**, то далее события будут развиваться по одному из трех сценариев:

- Новые сущности в коллекции, помещенные cascade=persist будут сохранены напрямую Doctrine
- Новые сущности в коллекции, не имеющие такой опции выдадут исключение в результате чего откат операции **flush()**.
- Коллекции, не имеющие новых сущностей будут пропущены.

Этот подход называется **Persistence by Reachability**: когда связь настроена на каскадное сохранение, то все новые сущности, найденные в коллекциях у уже существующих сущностей, будут автоматически сохранены.

8.7. Паттерн “Orphan Removal”

Есть еще один подобный механизм, он задействуется только при удалении сущностей из коллекций. Если сущность типа **A**содержит внутренние ссылки на сущности типа **B**, то когда ссылка **A**→*B* удаляется (а, следовательно, связь разрывается), то и сущность **B** также будет удалена, потому что с этого момента она нигде не используется.

Паттерн **OrphanRemoval** (“удаление объектов-сирот”) работает как со связями “один к одному” так и “один ко многим”.

> При использовании параметра **orphanRemoval=true** Doctrine делает предположение, что сущности являются закрытыми и **не будут** повторно использоваться другими сущностями. Если пренебречь этим допущением, сущности будут удалены даже если вы присвоите такую осиротевшую сущность какой-либо другой.

Для примера рассмотрим приложение **Addressbook** (адресная книга), в котором есть сущности **Contacts**, **Addresses** и**StandingData**:

```
<?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]);
    }
}
```

Следующий пример показывает, что произойдет, когда вы удалите ссылки:

```
<?php


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

$em->flush();
```

Здесь вы не только изменили саму сущность **Contact**, вы таже удалили ссылку на контактные данные **(standing data)** и ссылку на один контактный адрес. Когда будет вызван метод **flush()** из базы данных будут удалены не только ссылки, но и старая сущность с контактными данными **(staging data)**, а также сущность Address. Они будут удалены потому что они ни с кем не связаны.

Глава 9. События

В *Doctrine 2* имеется весьма удобная система событий, она идет как часть пакета *Common*.

9.1. Система событий

Любые события всегда находятся под контролем менеджера сущностей, именно он является центральным звеном в системе событий. На нем регистрируются слушатели событий, а вся обработка события также завязана на нем.

```
<?php

$evm = new EventManager();
```

После создания **$evm**, к нему можно добавлять слушателей. Давайте для примера создадим класс **EventTest**.

```
<?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;
    }
}

// Создадим экземпляр
$test = new EventTest($evm);
```

Запустить обработчик можно с помощью метода **dispatchEvent()**.

```
<?php

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

При помощи метода **removeEventListener()** можно отключить слушателя.

```
<?php

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

Помимо слушателей в системе событий **Doctrine 2** существует понятие подписчиков. Мы можем создать простой класс**TestEventSubscriber**, который определяет интерфейс **\Doctrine\Common\EventSubscriber** и имееь метод**getSubscribedEvents()**. Этот метод будет возвращать массив событий, на которые следует подписаться.

```
<?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);
```

При наступлении определенного события всем подписчикам будет отправлено соответствующее уведомление.

```
<?php

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

Можете проверить экземпляр **$eventSubscriber** и убедится, что был вызван метод **preFoo()**.

```
<?php

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

9.1.1. Именование

Назначать имена событиям лучше всего в стиле *CamelCase*, а значение соответствующей константы должно соответствовать ее имени, даже несмотря на орфографию. Этому есть следующие причины:

- Это легко читается.
- Простота.
- Каждый метод в _EventSubscriber _именуется после соответствующей константы. Если имя и значение константы отличаются, вам придется использовать новое значение, и, таким образом, менять сам код после изменения значения, что противоречит самой сути констант.

Пример правльной нотации был приведен выше в примере с **EventTest**.

9.2. События жизненного цикла

EntityManager и *UnitOfWork* могут вызывать множество различных событий в течение жизненного цикла подписанных на них сущностей.

- preRemove — возникает для заданной сущности перед тем как *EntityManager* применит к ней операцию удаления. Это событие не вызывается при вызове *DQL* запроса **DELETE**.

- postRemove — вызывается после того как сущность была удалена. Событие будет вызвано после выполнения операций удаления в базе даннах. Не вызывается для *DQL* запросов **DELETE**.
- prePersist — возникает для заданной сущности перед тем как *EntityManager* применит к ней операцию сохранения (персистирования).
- postPersist — возникает после того как сущность была сохранена. Событие вызывается посе операции вставки новой записи в базу данных. В этом событии будут доступны сгенерированные значения первичного ключа.
- preUpdate — возникает перед выполнением операций обновления в БД. Не вызывается для *DQL* запросов **UPDATE**.
- postUpdate — возникает после выполнения операций обновления в БД. Не вызывается для *DQL* запросов **UPDATE**.
- postLoad — возникает после того как сущность была загружена из БД в текущий *EntityManager* или после того как к ней была применена операция **refresh**.
- loadClassMetadata — возникает после того как для класса были загружены метаданные из одного из возможных источников (аннотации, *XMLили YAML*).
- onFlush — возникает после того как для всех *MANAGED*-сущностей были вычислены наборы необходимых изменений**(change-sets)**. Это событие не относится к колбеку жизненного цикла.
- onClear — возникает при выполнении операции **EntityManager#clear()**, после того как из *UnitOfWork* были удалены все ссылки на сущности.

> Заметьте, что событие *postLoad* возникает еще до того как были инициализированы связи сущности. Поэтому обращаться к связям из postLoad или обработчика события небезопасно.

Получить доступ к константам события можно из класса *Event*, относящемуся к пакету *ORM*.

```
<?php

use Doctrine\ORM\Events;
echo Events::preUpdate;
```

Существует два различных типа слушателей, которые могут перехватывать события:

- **Lifecycle Callbacks** – это методы самих сущностей, они исполняются при возникновении того или иного события. Им не передаются аргументы, они созданы

лишь для того, чтобы дать возможность вносить изменения изнутри контекста сущности.

- **Lifecycle Event Listeners** – это классы со своими специальными callback-методами, им передается соответствующий экземпляра класса EventArgs, который предоставляет доступ к сущности, EntityManager'у или иным данным.

> События, возникающие во время выполнения **EntityManager#flush()** накладывают специфические ограничения на перечень допустимых к использованию операций. В разделе "http://odiszapc.ru/doctrine/events/#115[Реализация обработчиков событий]" описано какие операции в каких событиях допустимы.

9.3. Обратный вызов

Событие жизненного цикла представляет собой обычное событие, на которое можно повесить метод-коллбек внутри класса сущности, который будет вызван при его наступлении.

```
<?php


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

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

    /* @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!';
    }
}
```

Обратите внимание, при использовании коллбеков к классу сущности нужно добавить маркер **@HasLifecycleCallbacks**.

Регистрация событий при использовании *YAML* или *XML* осуществляется как показано ниже.

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

XML будет выглядеть так:

```
<?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>
```

Единственное, нужно убедится, что в модели **User** определены открытые методы **doStuffOnPrePersist()** и **doStuffOnPostPersist()**.

```
<?php

// ...

class User
{
    // ...

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

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

В теге *lifecycleCallbacks *ключ представляет собой тип события, а значение — имена методов. Допустимые типы событий были перечислены в предыдущем разделе.

9.4. Обработка событий

Обработчики событий **(event listeners)** гораздо интереснее методов-коллбеков, определяемых внутри классов. Их использование позволяет реализовать механизмы, которые можно повторно использовать в различных классах, правда это требует более глубокого понимания работы внутренних аспектов работы *EntityManager* и *UnitOfWork*.

Понять как написать свой собственный обработчик поможет глава “Реализация обработчиков событий”[1].

Для регистрации обработчика нужно подключить его к *EventManager*, а затем передать последний в фабрику *EntityManager*:

```
<?php

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

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

Получить экзмепляр EventManager можно после создание EntityManager:

```
<?php

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

9.5. Реализация обработчиков событий

Эта секция объясняет какие действия можно, а какие нельзя выполнять в обработчиках событий *UnitOfWork*. Внимательно следуйте этим замечаниям, ведь хотя экземпляр *EntityManager* и передается всем обработчикам, выполнение некорректной операции может повлечь за собой большое число ошибок вроде нарушения консистентности данных, либо потерь операций обновления, сохранения или удаления данных.

Для описываемых событий, если они являются также и событиями жизненного цикла, действует еще одно ограничение — внутри обработчиков у вас не будет доступа к *API EntityManager* и *UnitOfWork*.

9.5.1. prePersist

Есть два способы вызвать событие *prePersist*. Первый очевиден — это когда вызывается **EntityManager#persist()**. Также, событие будет вызвано по иерархии для каскадных связей.

[1] http://odiszapc.ru/doctrine/events/#115

Другим способом событие может быть вызвано изнутри метода **flush()** после того как просчитаны все изменения, которые нужно будет внести в связи, и заданная связь была отмечена как **cascade persist**. Теперь любая новая сущность найденная при выполнении этой операции будет сохранена в базу, и для нее будет вызвано событие *prePersist*. Такой подход называется**"persistence by reachability"**.

В обоих случаях в обработчик передается экземпляр *LifecycleEventArgs*, который будет иметь доступ к самой сущности и ее*EntityManager*.

Для события *prePersist* действуют следующие ограничения:

- Если используется какой-нибудь *PrePersist Identity Generator,* например последовательность, то значение *ID* не будет доступно в событии *PrePersist*.
- *Doctrine* не умеет распознавать изменения, внесенные в связи в событии *PrePersist*, если оно было вызвано при каскадном сохранении (**by "reachability"**) как было описано выше, пока вы не будете использовать для этого внутренний*API UnitOfWork*. Мы не рекомендуем использовать подобные операции в таком контексте, так что делайте это на свой страх и риск, и да поможет вам Бог (и unit-тесты).

9.5.2. preRemove

Событие *preRemove* выполняется для каждой сущности, переданной методу **EntityManager#remove()**. Событие каскадно распространяется на все связи, которые позволяют каскадное удаление.

Нет никаких ограничений на вызываемые методы внутри этого события, за исключением того когда метод **remove()** вызывается во время операции **flush()**.

9.5.3. onFlush

OnFlush одно из самых навороченных. Оно вызывается внутри метода **EntityManager#flush()** после того как будут просчитаны все изменения в *MANAGED* сущностях и их связях. Это означает, что *onFlush* имее досутп к наборам:

- Сущностей, запланированных для вставки
- Сущностей, запланированных для обновления
- Сущностей, запланированных для удаления
- Коллекций, запланированных для обновления
- Коллекций, запланированных для удаления

Чтобы нормально работать с этим событием нужно разбираться во внутреннем *API UnitOfWork*, именно он предоставляет доступ к вышеприведенным наборам данных. Рассмотрим пример:

```
<?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) {

        }
    }
}
```

Для *onFlush* действуют следующие ограничения:

- Если в обработчике *onFlush* создается и сохраняется новую сущность, то одного вызова **EntityManager#persist()**недостаточно. Нужно сделать еще один — **$unitOfWork→computeChangeSet($classMetadata, $entity)**
- Изменение полей или связей требует явного пересчета набора изменений **(changeset)** затрагиваемой сущности. Делается это вызовом **$unitOfWork→recomputeSingleEntityChangeSet($classMetadata, $entity)**.

9.5.4. preUpdate

У этого события больше всего ограничений, т.к. вызывается оно внутри метода **EntityManager#flush()** непосредственно перед*SQL UPDATE*.

В этом событии нельзя вносить изменения в связи обновляемой сущности, т.к. на данном этапе операции **flush** *Doctrine* не сумеет гарантированно обеспечить ссылочную целостность. Однако у этого события есть существенный плюс — оно вызывается с набором аргументов *PreUpdateEventArgs*, в котором содержится ссылка на просчитанный **change-set** для заданной сущности.

Это означает, что можно получить доступ ко всем затронутым при изменении полям, при этом будет доступно как старое, так и новое значения поля. У *PreUpdateEventArgs* есть следующие методы:

- **getEntity()** возвращает саму сущность.
- **getEntityChangeSet()** возвращает копию массива с набором изменений. Изменения, внесенные в этот массив никак не повлияют на операцию обновления.
- **hasChangedField($fieldName)** проверяе изменилось ли заданное поле или нет.
- **getOldValue($fieldName)** и **getNewValue($fieldName)** возвращают значения поля до и после его изменения соответственно.
- **setNewValue($fieldName, $value)** позволяет изменить значение поля.

Типичный пример работы с *preUpdate* выглядит примерно так:

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

На основе обработки этого события можно строить валидацию полей. Это гораздо эффективней использования **lifecycle callback**, в которых валидация дается недешево:

```php
<?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)
    {
        // throw an exception to interrupt flush event. Transaction will be rolled
 back.
    }
}
```

Для этого события существуют следующие ограничения:

- Изменения в связях переданных сущностей не будут восприниматься операцией **flush()**.
- Изменение полей также не будут восприняты операцией **flush()**, для этого нудно использовать сформированный **change-set**, которые передается событию.
- Настоятельно не рекомендуется вызывать **EntityManager#persist()** или **EntityManager#remove()** даже в комбинации с *API UnitOfWork*, т.к. вне операции **flush()** они не будут работать как ожидается.

9.5.5. postUpdate, postRemove, postPersist

Эти три события вызываются внтури **EntityManager#flush()**. Изменения, выполненные в этих событиях не повлияют на базу данных, но можно использовать эти события для воздействия на несохраняемые элементы сущности, например простые поля класса, которые не отображены на БД, логирование или даже какие-то связанные классы, которые непосредственно обрабатываются Доктриной.

9.5.6. postLoad

Это событие возникает после того как сущность была сконструирована менеджером сущностей.

9.6. Событие loadClassMetadata

После считывания метаданных сущности они передаются в объект класса **ClassMetadataInfo**. Для манипуляции этим объектом нужно обработать событие **loadClassMetadata**.

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

Глава 10. Doctrine Internals explained

Object relational mapping is a complex topic and sufficiently understanding how Doctrine works internally helps you use its full power.

10.1. How Doctrine keeps track of Objects

Doctrine uses the Identity Map pattern to track objects. Whenever you fetch an object from the database, Doctrine will keep a reference to this object inside its UnitOfWork. The array holding all the entity references is two-levels deep and has the keys "root entity name" and "id". Since Doctrine allows composite keys the id is a sorted, serialized version of all the key columns.

This allows Doctrine room for optimizations. If you call the EntityManager and ask for an entity with a specific ID twice, it will return the same instance:

```
public function testIdentityMap()
{
    $objectA = $this->entityManager->find('EntityName', 1);
    $objectB = $this->entityManager->find('EntityName', 1);

    $this->assertSame($objectA, $objectB)
}
```

Only one SELECT query will be fired against the database here. In the second `EntityManager#find()` call Doctrine will check the identity map first and doesn't need to make that database roundtrip.

Even if you get a proxy object first then fetch the object by the same id you will still end up with the same reference:

```
public function testIdentityMapReference()
{
    $objectA = $this->entityManager->getReference('EntityName', 1);
    // check for proxyinterface
    $this->assertInstanceOf('Doctrine\ORM\Proxy\Proxy', $objectA);

    $objectB = $this->entityManager->find('EntityName', 1);
```

```
    $this->assertSame($objectA, $objectB)
}
```

The identity map being indexed by primary keys only allows shortcuts when you ask for objects by primary key. Assume you have the following `persons` table:

```
id | name
-------------
1  | Benjamin
2  | Bud
```

Take the following example where two consecutive calls are made against a repository to fetch an entity by a set of criteria:

```
public function testIdentityMapRepositoryFindBy()
{
    $repository = $this->entityManager->getRepository('Person');
    $objectA = $repository->findOneBy(array('name' => 'Benjamin'));
    $objectB = $repository->findOneBy(array('name' => 'Benjamin'));

    $this->assertSame($objectA, $objectB);
}
```

This query will still return the same references and $objectA and $objectB are indeed referencing the same object. However when checking your SQL logs you will realize that two queries have been executed against the database. Doctrine only knows objects by id, so a query for different criteria has to go to the database, even if it was executed just before.

But instead of creating a second Person object Doctrine first gets the primary key from the row and check if it already has an object inside the UnitOfWork with that primary key. In our example it finds an object and decides to return this instead of creating a new one.

The identity map has a second use-case. When you call `EntityManager#flush` Doctrine will ask the identity map for all objects that are currently managed. This means you don't have to call `EntityManager#persist` over and over again to pass known objects to the EntityManager. This is a NO-OP for known entities, but leads to much code written that is confusing to other developers.

The following code WILL update your database with the changes made to the `Person` object, even if you did not call `EntityManager#persist`:

```
<?php
```

```
$user = $entityManager->find("Person", 1);
$user->setName("Guilherme");
$entityManager->flush();
```

10.2. How Doctrine Detects Changes

Doctrine is a data-mapper that tries to achieve persistence-ignorance (PI). This means you map php objects into a relational database that don't necessarily know about the database at all. A natural question would now be, "how does Doctrine even detect objects have changed?".

For this Doctrine keeps a second map inside the UnitOfWork. Whenever you fetch an object from the database Doctrine will keep a copy of all the properties and associations inside the UnitOfWork. Because variables in the PHP language are subject to "copy-on-write" the memory usage of a PHP request that only reads objects from the database is the same as if Doctrine did not keep this variable copy. Only if you start changing variables PHP will create new variables internally that consume new memory.

Now whenever you call `EntityManager#flush` Doctrine will iterate over the Identity Map and for each object compares the original property and association values with the values that are currently set on the object. If changes are detected then the object is queued for a SQL UPDATE operation. Only the fields that actually changed are updated.

This process has an obvious performance impact. The larger the size of the UnitOfWork is, the longer this computation takes. There are several ways to optimize the performance of the Flush Operation:

- Mark entities as read only. These entities can only be inserted or removed, but are never updated. They are omitted in the changeset calculation.
- Temporarily mark entities as read only. If you have a very large UnitOfWork but know that a large set of entities has not changed, just mark them as read only with `$entityManager->getUnitOfWork()->markReadOnly($entity)`.
- Flush only a single entity with `$entityManager->flush($entity)`.
- Use id1[:doc:`Change Tracking Policies <change-tracking-policies>`] to use more explicit strategies of notifying the UnitOfWork what objects/properties changed.

10.3. Query Internals

tbr

10.4. The different ORM Layers

Doctrine ships with a set of layers with different responsibilities. This section gives a short explanation of each layer.

10.5. Hydration

Responsible for creating a final result from a raw database statement and a result-set mapping object. The developer can choose which kind of result he wishes to be hydrated. Default result-types include:

- SQL to Entities
- SQL to structured Arrays
- SQL to simple scalar result arrays
- SQL to a single result variable

Hydration to entities and arrays is one of most complex parts of Doctrine algorithm-wise. It can build results with for example:

- Single table selects
- Joins with n:1 or 1:n cardinality, grouping belonging to the same parent.
- Mixed results of objects and scalar values
- Hydration of results by a given scalar value as key.

10.6. Persisters

tbr

10.7. UnitOfWork

tbr

10.8. ResultSetMapping

tbr

10.9. DQL Parser

tbr

10.10. SQLWalker

tbr

10.11. EntityManager

tbr

10.12. ClassMetadataFactory

tbr

Глава 11. Association Updates: Owning Side and Inverse Side

When mapping bidirectional associations it is important to understand the concept of the owning and inverse sides. The following general rules apply:

- Relationships may be bidirectional or unidirectional.
- A bidirectional relationship has both an owning side and an inverse side
- A unidirectional relationship only has an owning side.
- Doctrine will **only** check the owning side of an association for changes.

11.1. Bidirectional Associations

The following rules apply to **bidirectional** associations:

- The inverse side has to use the `mappedBy` attribute of the OneToOne, OneToMany, or ManyToMany mapping declaration. The mappedBy attribute contains the name of the association-field on the owning side.
- The owning side has to use the `inversedBy` attribute of the OneToOne, ManyToOne, or ManyToMany mapping declaration. The inversedBy attribute contains the name of the association-field on the inverse-side.
- ManyToOne is always the owning side of a bidirectional association.
- OneToMany is always the inverse side of a bidirectional association.
- The owning side of a OneToOne association is the entity with the table containing the foreign key.
- You can pick the owning side of a many-to-many association yourself.

11.2. Important concepts

Doctrine will only check the owning side of an association for changes.

To fully understand this, remember how bidirectional associations are maintained in the object world. There are 2 references on each side of the association and these 2 references both represent the same association but can change independently of one another. Of course, in

a correct application the semantics of the bidirectional association are properly maintained by the application developer (that's his responsibility). Doctrine needs to know which of these two in-memory references is the one that should be persisted and which not. This is what the owning/inverse concept is mainly used for.

Changes made only to the inverse side of an association are ignored. Make sure to update both sides of a bidirectional association (or at least the owning side, from Doctrine's point of view)

The owning side of a bidirectional association is the side Doctrine "looks at" when determining the state of the association, and consequently whether there is anything to do to update the association in the database.

> "Owning side" and "inverse side" are technical concepts of the ORM technology, not concepts of your domain model. What you consider as the owning side in your domain model can be different from what the owning side is for Doctrine. These are unrelated.

Глава 12. Транзакции и параллелизм

12.1. Разделение транзакций

Разделение транзакций есть ни что иное как определение границ кода работы с БД, которые они будут охватывать. Это немаловажный аспект, ведь от неверного определения границ может снизиться производительность вашего приложения. Многие БД и прослойки вроде библиотеки *PDO* по умолчанию работают в режиме авто-коммитов, т.о. любой сколь угодно малый SQL-запрос будет завернут в свою отдельную транзакцию. Транзакции — мощных механизм, поэтому без явного вмешательства со стороны программиста их использование скорее приведет к снижению производительности, чем к ускорению.

К счастью, *Doctrine 2* уже позаботилась о правильном разделении транзакций: все операции записи (**INSERT** / **UPDATE** /**DELETE**) по умолчанию ставятся в очередь, а когда происходит вызов **EntityManager#flush()**, все они исполняются в рамках единой транзакции.

Помимо этого, в *Doctrine 2* есть возможность самостоятельно управлять поведением транзакций, и именно такой подход рекомендуется к применению на практике.

Существует два способа работы с транзакциями, рассмотрим их подробнее.

12.1.1. Подход 1: косвенный

Первый подход основан на неявной поддержке транзакций механизмом *ORM EntityManager*. Возьмем следующий пример, без явного разграничения на транзакции:

```
<?php

// $em это экземпляр EntityManager
$user = new User;
$user->setName('George');
$em->persist($user);
$em->flush();
```

Здесь мы не задавали транзакции явным образом, поэтому **EntityManager#flush()** сам сделает* commit* или **rollback**. Подобное поведение достигается за счет агрегации *DML* операций движком *ORM*, этого достаточно если все манипулирование данными работает под эгидой *Unit of Work* и происходит в контексте модели предметной области и, следовательно, *ORM*.

12.1.2. Подход 2: явный

Явный способ работы с транзакциями заключается в использовании напрямую *API Doctrine\DBAL\Connection*. Код будет выглядеть следующим образом:

```
<?php

// $em это экземпляр EntityManager
$em->getConnection()->beginTransaction(); // auto-commit произойдет автоматически
try {
    //... какой-то код
    $user = new User;
    $user->setName('George');
    $em->persist($user);
    $em->flush();
    $em->getConnection()->commit();
} catch (Exception $e) {
    $em->getConnection()->rollback();
    $em->close();
    throw $e;
}
```

Явное управление транзакциями необходимо когда вам нужно включить дополнительные операции DBAL в модуль работы (Unit of Work). Или другой вариант: вам нужно использовать какие-то методы менеджера сущностей, которые требуют активной транзакции. Такие методы будут выдавать исключение типа *TransactionRequiredException* чтобы информировать вас об этом требовании.

В качестве более удобной альтернативы явному разграничению транзакций можно использовать механизм **control abstractions** в форме методов **Connection#transactional($func)** и **EntityManager#transactional($func)**. Использование таких абстракций, помимо очевидного сокращения объема кода гарантирует, что вы не забудете сделать **rollback** транзакции или закрыть *EntityManager*. Следующий пример является полным эквивалентом предыдущего кода:

```
<?php
```

```
    // $em это экземпляр EntityManager
    $em->transactional(function($em) {
    //... какой-то код
    $user = new User;
    $user->setName('George');
    $em->persist($user);
});
```

Разница между **Connection#transactional($func)** и **EntityManager#transactional($func)** заключается в том, что последний перед коммитом делает **EntityManager#flush()**, а в случае возникновения исключений корректно закрывает *EntityManager* (в дополнение к откату транзакции).

12.1.3. Обработка исключений

Если при вызове **EntityManager#flush()** возникнет исключение, транзакция автоматически откатится, а *EntityManager*закроется.

Как было показано на предыдущем примере, при явном разграничении транзакций когда оная терпит неудачу вследствие, например, исключения, происходит откат транзакции, а менеджер сущностей закрывается путем вызова **EntityManager#close()**и работа с ним прекращается. И как было показано ранее, более элегантно это можно сделать при помощи **control abstractions**. Заметьте, после того как вы поймали исключение, нужно будет заново выбросить его. Если вы все же хотите корректно обрабатывать некоторые исключения, нужно явно поймать их в более ранних catch-блоках (но не забудьте в том же месте откатить транзакцию и закрыть *EntityManager*). Все остальные методики обработки исключений стоит применять подобным же образом **(i.e. either log or re-throw, not both, etc.)**

Как результат подобного поведения, все экземпляры сущностей которые имели статус **MANAGED** или **REMOVED** станут**DETACHED**. Отсоединенные объекты получат состояние, которое они имели на момент откат транзакции. **The state of the objects is in no way rolled back and thus the objects are now out of synch with the database**. Приложения может работать с такими отсоединенными объектами, но стоит иметь ввиду, что их состояние может быть неточным.

Если после того как произошло исключение вы захотите начать новый блок (unit of work), это нужно будет делать с новым экземпляром *EntityManager*.

12.2. Блокировки

Doctrine 2 имеет встроенную поддержку блокировок двух типов: пессимистичной и оптимистичной. Это позволит весьма точно контролировать вопрос того какой вид блокирования больше подходит сущностям именно вашего приложения.

12.2.1. Оптимистичная блокировка (Optimistic locking)

Что же, транзакции отлично работают при распараллеливании в одиночном запросе. Однако, транзакция не должна охватывать несколько запросов, это так называемое “время раздумья” (“user think time”). Поэтому длительная “бизнес-транзакция”, которая охватывает несколько запросов должна включать в себя несколько обычных транзакций. По этой причине, одиночные транзакции больше не могут управлять параллелизмом в рамках более длительных бизнес-транзакций. Ответственность за это частично ложиться на само приложение.

Doctrine имеет встроенную поддержку оптимистичной блокировки, реализованной с помощью поля **version**. При таком подходе сущность, которую нужно защитить от одновременного изменения в рамках длительной транзакции получает специальное поле**version**, представляющее собой либо число (тип *integer*) либо временную метку (типа *datetime*). Затем когда в конце длительной операции эта сущность сохраняется происходит сравнение ее текущей версии и версии из базе данных, и, если они не совпадают будет выброшено исключение типа **OptimisticLockException**, которое будет говорить о том, что сущность была изменения кем-то еще.

Назначить версию для сущности можно следующим образом. В примере мы будем использовать целочисленный тип *integer*.

```
<?php

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

В качестве альтернативы можно использовать тип *datetime* (он отображается на *SQL* типы *timestamp* или *datetime*):

```php
<?php

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

Целочесленные версии являются более предпочтительным вариантом по сравнению с временными метками, т.к. они менее подвержены конфликтам в высоконагруженной среде, в противоположность временным меткам, которые зависят от разрешающей способности типа *timestamp* конкретной СУБД.

Если при вызове* EntityManager#flush()* произойдет конфликт версии, будет выброшено исключение **OptimisticLockException**и произойдет откат активной транзакции (либо она будет поставлена в очередь для отката). Это исключение может быть соответствующим образом поймано и обработано. Возможными реакциями на исключения **OptimisticLockException** могут быть, например, выдача сообщения о конфликте пользователю или попытка обновить и загрузить объекты в новую транзакцию и попытаться повторно ее исполнить.

Время между показом html-формы и фактическим внесением изменений в сущность в самом худшем варианте может достигать времени жизни самой сессии. И если в течение этого периода в сущность были внесены какие-то изменения, то непосредственно при получении сущности будет выброшенго исключение **optimistic locking exception**: (*Оригинал:* ***If changes happen to the entity in that time frame you want to know directly when retrieving the entity that you will hit an optimistic locking exception****:*)

Вы можете сами проверить версию сущности с помощью **EntityManager#find()**:

```php
<?php

use Doctrine\DBAL\LockMode;
use Doctrine\ORM\OptimisticLockException;

$theEntityId = 1;
$expectedVersion = 184;

try {
```

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

    // какой-то код

    $em->flush();
} catch(OptimisticLockException $e) {
    echo "Не в огорчение будет сказано, но кто-то уже изменил эту сущность.
 Примените изменения еще раз!";
}
```

Или же это можно сделать с помощью метода **EntityManager#lock()**:

```
<?php

use Doctrine\DBAL\LockMode;
use Doctrine\ORM\OptimisticLockException;

$theEntityId = 1;
$expectedVersion = 184;

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

try {
    // assert version
    $em->lock($entity, LockMode::OPTIMISTIC, $expectedVersion);

} catch(OptimisticLockException $e) {
    echo "Не в огорчение будет сказано, но кто-то уже изменил эту сущность.
 Примените изменения еще раз!";
}
```

12.2.2. Важные замечания по реализации

Достаточно просто запороть процесс работы с оптимистичной блокировкой если некорректно проводить сравнение версий. Предположим, что есть два пользователя — Алиса и Боб, пытающиеся получить доступ к банковскому аккаунту:

- Алиса читает заголовок статьи в блоге, равный **“Foo”**, для версии блокировки с номером* 1* (GET запрос)
- Боб также читает заголовок статьи в блоге, равный **“Foo”**, для версии блокировки равной **1** (GET запрос).
- Боб меняет заголовок на **“Bar”**, тем самым обновляя версию на **2** (POST запрос с формы)

- Алиса меняет заголовок на **“Baz”**, … (POST запрос с формы)

На последнем шаге этого сценария статья блога еще раз будет считана из базы данных до того как будет применено изменения заголовка от Алисы. На этом шаге нужно проверить, что статья все еще имеет версию **1** (этого нет в сценарии)

При правильной работе с оптимистичной блокировкой необходимо добавить поле со значением версии в качестве скрытого поля формы (или, для большей безопасности, в сессию). В противном случае вы не сможете удостовериться в том, что версия та же самая, что и была при чтении из базы когда Алиса выполняла свой GET запрос. Если это случится, вы потеряете часть обновлений; произойдет то, чего вы пытались избежать с помощью оптимистичной блокировки.

Рассмотрим пример. Есть форма (GET запрос):

```
<?php

$post = $em->find('BlogPost', 123456);

echo 'getId() . '" />';
echo 'getCurrentVersion() . '" />';
```

А вот код изменения заголовка (POST запрос):

```
<?php

$postId = (int)$_GET['id'];
$postVersion = (int)$_GET['version'];

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

12.2.3. Пессимистичная блокировка (Pessimistic Locking)

В Doctrine 2 пессимистичная блокировка поддерживается на уровне СУБД. Не предпринималось никаких попыток реализовать этот функционал внутри самой *Doctrine*, правильнее использовать возможности конкретной СУБД и команды ANSI-SQL реализации блокировки на уровне строк. Любая сущность может работать в рамках такой блокировки, для этого не нужно определять никаких специальных параметров.

Однако, для того, чтобы пессимистичная блокировка заработала как положено вам нужно отключить режим авто-коммитов в СУБД, а код, ориентированный на

использование такой блокировки должен быть обернут в транзакцию, как это описано в разделе 10.1.2. Подход 2: явный[1] данной главы. Если вы попробуете использовать пессимистичную блокировки вне транзакции*Doctrine 2* выбросит исключение.

В настоящий момент поддерживаются 2 режима блокировки:

- **Pessimistic Write** (Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE), блокирует строки базы данных от параллельных операций чтения и записи.
- **Pessimistic Read** (Doctrine\DBAL\LockMode::PESSIMISTIC_READ), блокирует иные параллельные запросы, которые пытаются обновить строки или залочить их в режиме записи.

Использовать пессимистичные блокировки можно тремя различными способами:

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

[1] http://odiszapc.ru/doctrine/transactions-and-concurrency/#1012_2

Глава 13. Пакетная обработка

Эта глава покажет как наиболее эффективно можно осуществлять массовую вставку, обновление и удаление данных в*Doctrine*. Основная проблема при работе с массовыми операциями это повышенное потребление памяти. Есть несколько стратегий, которые могут помочь вам в этом вопросе.

> ORM, вообще говоря, не предназначена для таких операций. Для этого в каждой СУБД есть свой собственный гораздо более эффективный механизм, так что если описанные стратегии вам не подойдут, мы рекомендуем воспользоваться инструментами конкретной СУБД, предназначенными для таких операций.

13.1. Массовые вставки

В *Doctrine* массовые вставки лучше всего выполнять партиями, используя возможности транзакций *EntityManager*. Следующий пример показывает как вставлять 10000 объектов партиями по 20 штук. Для достижения оптимального результата имеет смысл поэкспериментировать с размером партии. Больший размер означает бо'льшие возможности для повторного использования внутренних операций, но и требует больше работы при операции **flush()**.

```
<?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(); // Detaches all objects from Doctrine!
    }
}
```

13.2. Массовые обновления

В *Doctrine* есть два способа осуществления массовых обновлений.

13.2.1. DQL UPDATE

На сегодняшний день наиболее эффективным способом осуществления массовых обновлений является *DQL* запрос *UPDATE:*

```
<?php

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

13.2.2. Итерация по результирующему набору

Другой подход заключается в использовании метода **Query#iterate()** для итерации по результирующему набору без необходимости загрузки полного набора данных в память. Следующий пример показывает как это можно сделать, комбинируя итерацию и пакетную обработку:

```
<?php

$batchSize = 20;
$i = 0;
$q = $em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
foreach($iterableResult AS $row) {
    $user = $row[0];
    $user->increaseCredit();
    $user->calculateNewBonuses();
    if (($i % $batchSize) == 0) {
        $em->flush(); // Выполняет обновления
        $em->clear(); // Отсоединяет все объекты от Doctrine
    }
    ++$i;
}
```

> Нельзя осуществлять итерацию в запросах, которые подсоединяют связи-коллекции **(collection-valued association)**. Природа таких результируюзих наборов не подходит для инкрементой гидрации.
>
> Оригинал: **Iterating results is not possible with queries that fetch-join a collection-valued association. The nature of such SQL result sets is not suitable for incremental hydration.**

13.3. Массовое удаление

В *Doctrine* есть два способа осуществления массовых удалений. Можно либо запустить одиночный *DQL* запрос *DELETE* или же проитерировать набор данных, удаляя каждый из элементов по отдельности.

13.3.1. DQL DELETE

На сегодняшний день наиболее эффективным способом для массовых удалений является *DQL _запрос _DELETE*:

```
<?php

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

13.3.2. Итерация по результирующему набору

Другой подход заключается в использовании метода **Query#iterate()** для итерации по результирующему набору без необходимости загрузки полного набора данных в память. Следующий пример показывает как это можно сделать, комбинируя итерацию и пакетную обработку:

```
<?php

$batchSize = 20;
$i = 0;
$q = $em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
while (($row = $iterableResult->next()) !== false) {
    $em->remove($row[0]);
    if (($i % $batchSize) == 0) {
        $em->flush(); // Выполняет удаления
        $em->clear(); // Отсоединяет все объекты от Doctrine
    }
    ++$i;
}
```

> Нельзя осуществлять итерацию в запросах, которые подсоединяют связи-коллекции **(collection-valued association)**. Природа таких результируюзих наборов не подходит для инкрементой гидрации.

Оригинал: **Iterating results is not possible with queries that fetch-join a collection-valued association. The nature of such SQL result sets is not suitable for incremental hydration.**

13.4. Iterating Large Results for Data-Processing

Если не требуется делать *UPDATE* или *DELETE*, то для итерации можно использовать метод **iterate()**. Экземпляр *IterableResult*, возвращаемый методом **$query→iterate()** определяет интерфейс *Iterator*, поэтому можно обрабатывать большой результирующий набор при минимальных затратах памяти:

```
<?php

$q = $this->_em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
foreach ($iterableResult AS $row) {
    // Что-то делаем с данными в строке, $row[0] всегда является объектом

    // Отсоединяем от Doctrine, так что сразу будет запущена сборка мусора
    $this->_em->detach($row[0]);
}
```

Нельзя осуществлять итерацию в запросах, которые подсоединяют связи-коллекции **(collection-valued association)**. Природа таких результируюзих наборов не подходит для инкрементой гидрации.

Оригинал: **Iterating results is not possible with queries that fetch-join a collection-valued association. The nature of such SQL result sets is not suitable for incremental hydration.**

Глава 14. Язык DQL – Doctrine Query Language

Doctrine Query Language (DQL) – _данное семейство запросов производное от языка _Object Query Language, который в свою очередь чем-то напоминает такие грамматики как *Hibernate Query Language (HQL)* и *Java Persistence Query Language (JPQL)*.

С помощью DQL можно строить довольно мощные запросы к существующим объектным моделям. Представьте себе, что все объекты у вас хранятся в некотором хранилище (что-то вроде объектной базы данных), и с помощью DQL запросов вы обращаетесь к этому хранилищу с целью получить необходимое вам подмножество объектов.

> Типичная ошибка новичков состоит в том, что они думают о DQL как об очередной форме SQL, пытаясь вставлять в запросы имена таблиц и столбцов, или же JOIN'ить таблицы друг с другом. Так что имейте ввиду, DQL — это язык запросов только для объектной модели, для реляционных движняков он не подходит.

DQL не чувствителен к регистру символов за исключением пространств имен, названий классов и их полей.

14.1. Типы DQL запросов

В DQL присутствуют такие конструкции как **SELECT, UPDATE и DELETE**, они аналогичны своим собратьям из мира SQL. Операция **INSERT** отсутствуют, потому что для согласованности объектной модели все сущности и связи заводятся под управление ORM через вызов **EntityManager#persist()** и способа вставлять из в базу напрямую не предусмотрено, да он и не нужен.

Запрос **SELECT** умеет вытаскивать какие-то определенные куски из вашей доменной модели, к которым нельзя получить доступ при помощь связей. В дополнение к этому, такие запросы позволяют запрашивать сущности вместе с полным набором связей с помощью единственного SQL запроса, что не может не радовать.

С помощью запросов **UPDATE** и **DELETE** можно выполнять пакетные обновления и удаления сущностей из доменной модели. Это бывает весьма полезно, ведь не всегда есть возможность загрузить в память полный набор сущностей для их последующего изменения.

14.2. Запросы SELECT

14.2.1. DQL SELECT

Выражение SELECT определяет какие данные появятся в результатах запроса (кто-бы мог подумать, епта). Композиция различных выражений в запросе SELECT также может влиять на природу результатов запроса.

Пример ниже производит выборку пользователей старше 20 лет:

```
<?php

$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.age > 20');
$users = $query->getResult();
```

Давайте рассмотрим это запрос:

- **u** — это синоним, указывающий на класс *MyProject\Model\User*. Помещая этот алиас в выражение *SELECT* мы тем самым указываем, что нам нужно получить все экземпляры именно класса User.
- За ключевым словом FROM всегда следует полное имя класса, за которым в свою очередь следует алиас для этого класса. Класс это своего рода корень запроса, от которого далее можно перемещаться с помощью JOIN'ов (будет описано позднее) и различных путевых выражений **(path expressions)**.
- Выражение **u.age** в блоке WHERE это и есть путевое выражение. Их легко найти по оператору '.', используемого для формирования путей. Выражение **u.age** указывает на поле age класса User.

Результатом этого запроса будет список объектов User, все пользователи в котором старше 20 лет.

Внутри выражения SELECT можно указывать как ключевые поля класса для загрузки всей сущности, так и лишь некоторые из них с помощью синтаксиса **u.name**. Можно комбинировать эти способы, а также применять к ним функции агрегации DQL. Числовые поля также могут использоваться в математических операциях. Для дополнительной информации смотрите разделы Функции, операторы и агрегации[1].

[1] http://odiszapc.ru/doctrine/dql-doctrine-query-language/#135

14.2.2. JOIN

Запрос SELECT может содержать JOIN'ы двух типов: "Regular" и "Fetch".

Regular Joins: используются с целью фильтрации результатов запросов, а также вычисления агрегированных значений.

Fetch Joins: Похожи на обычные JOIN'ы, но дополнительно вытаскивают из базы все связанные сущности и включают их в результат запроса.

Не существует какого-то ключевого слова, которое бы говорило о том какой из JOIN'ов использовать. JOIN, будь то INNER JOIN или OUTER JOIN, будет трактоваться как "Fetch JOIN" если поля подключаемой через JOIN сущности появятся в SELECT-части DQL запроса вне какой-либо агрегатной функции. В противном случае это будет обычный (regular) JOIN.

Пример обычного JOIN-а по полю адреса:

```
<?php

$query = $em->createQuery("SELECT u FROM User u JOIN u.address a WHERE a.city =
 'Berlin'");
$users = $query->getResult();
```

Пример fetch-JOIN-а по адресу:

```
<?php

$query = $em->createQuery("SELECT u, a FROM User u JOIN u.address a WHERE a.city =
 'Berlin'");
$users = $query->getResult();
```

Когда Doctrine строит результат для запроса с Fetch-JOIN'ами, на первом уровне результирующего массива будет расположен класс из выражения FROM. В предыдущем примере будет возвращен массив экземпляров класса User, при этом в каждый экземпляр будет добавлена переменная **User#address**. Когда вы обратитесь к этой переменной Doctrine не нужно будет дополнительно подгружать всю связь с помощью другого запроса, нет необходимости использоваться здесь *Lazy loading*.

> Тем не менее Doctrine дает возможность ссылаться на любые доступные связи между объектами доменной модели. Объекты, которые не были загружены из БД заменяются на экземпляры прокси-классов, для их

загрузки будет использоваться “ленивая загрузка”. С коллекциями все аналогично — они будут загружены с помощью lazy-loading при первом доступе к ним. Помните, использование lazy-loading ведет к том, что база данных будет бомбардироваться кучей мелких запросов, что безусловно может отрицательно сказаться на производительности приложения. И Fetch-JOIN’ы как раз таки и позволяют избежать этого — они загрузят всю нужную вам ветку сущностей с помощью единственного SELECT запроса.

14.2.3. Именованные и позиционные параметры

DQL поддерживает два типа параметров: именованные и позиционные. Однако, в отличие от многих SQL-грамматик, здесь позиционные параметры задаются с помощью чисел, например ”?1”, ”?2” и т.д. Именованные параметры задаются в виде ”:name1”, ”:name2” и т.д.

Когда нужно сослаться на параметр в методе **Query#setParameter($param, $value)**, то оба типа параметров нужно указывать без префиксов.

14.2.4. Примеры DQL SELECT

Этот раздел содержит большой набор различных DQL запросов с комментариями. Фактический результат также зависит от режима **hydrations** (черт побьери, кто-нибудь знает адекватный аналог этого слова на русском).

Запросив все сущности класса **User**:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u');
$users = $query->getResult(); // массив объектов класса User
```

Получение первичных ключей (ID) всех CmsUsers:

```
<?php

$query = $em->createQuery('SELECT u.id FROM CmsUser u');
$ids = $query->getResult(); // массив идентификаторов CmsUser
```

Получение идентификаторов всею юзеров, написавших хотя бы одну статью:

```
<?php
```

```
$query = $em->createQuery('SELECT DISTINCT u.id FROM CmsArticle a JOIN a.user u');
$ids = $query->getResult(); // массив идентификаторов CmsUser
```

Получение всех статей, отсортированных по имени автора:

```
<?php

$query = $em->createQuery('SELECT a FROM CmsArticle a JOIN a.user u ORDER BY u.name
 ASC');
$articles = $query->getResult(); // массих объектов CmsArticle
```

Получение полей Username и Name класса CmsUser:

```
<?php

$query = $em->createQuery('SELECT u.username, u.name FROM CmsUser u');
$users = $query->getResults(); // массив значение полей username и name класса
 CmsUser
echo $users[0]['username'];
```

Получение объектов ForumUser и связанной с ними сущности:

```
<?php

$query = $em->createQuery('SELECT u, a FROM ForumUser u JOIN u.avatar a');
$users = $query->getResult(); // массив объектов ForumUser с загруженной связью
 avatar
echo get_class($users[0]->getAvatar());
```

Получение объекта CmsUser с полной загрузкой всех его телефонных номеров:

```
<?php

$query = $em->createQuery('SELECT u, p FROM CmsUser u JOIN u.phonenumbers p');
$users = $query->getResult(); // массив объектов CmsUser с загруженной связью
 phonenumbers
$phonenumbers = $users[0]->getPhonenumbers();
```

Сортировка по возрастанию:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id ASC');
```

```
$users = $query->getResult(); // массив объектов ForumUser
```

Сортировка по убыванию:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id DESC');
$users = $query->getResult(); // массив объектов ForumUser
```

Аггрегатные функции:

```
<?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();
```

Выражение WHERE и позиционные параметры:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$users = $query->getResult(); // массив объектов ForumUser
```

Предложение WHERE и именованные параметры:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.username = :name');
$query->setParameter('name', 'Bob');
$users = $query->getResult(); // array of ForumUser objects
```

Вложенные условия в предложении WHERE:

```
<?php

$query = $em->createQuery('SELECT u from ForumUser u WHERE (u.username = :name OR
 u.username = :name2) AND u.id = :id');
$query->setParameters(array(
    'name' => 'Bob',
```

```
    'name2' => 'Alice',
    'id' => 321,
));
$users = $query->getResult(); // массив объектов ForumUser
```

COUNT DISTINCT:

```
<?php

$query = $em->createQuery('SELECT COUNT(DISTINCT u.name) FROM CmsUser');
$users = $query->getResult(); // массив объектов ForumUser
```

Арифметическое выражение:

```
<?php

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE ((u.id + 5000) * u.id + 3)
 < 10000000');
$users = $query->getResult(); // массив объектов ForumUser
```

Получение идентификаторов пользователей и статей, если они есть при помощи LEFT JOIN:

```
<?php

$query = $em->createQuery('SELECT u.id, a.id as article_id FROM CmsUser u LEFT JOIN
 u.articles a');
$results = $query->getResult(); // array of user ids and every article_id for each
 user
```

Дополнительные условия в JOIN:

```
<?php

$query = $em->createQuery("SELECT u FROM CmsUser u LEFT JOIN u.articles a WITH
 a.topic LIKE '%foo%'");
$users = $query->getResult();
```

Использование нескольких Fetch-JOIN'ов:

```
<?php
```

```
$query = $em->createQuery('SELECT u, a, p, c FROM CmsUser u JOIN u.articles a JOIN
 u.phonenumbers p JOIN a.comments c');
$users = $query->getResult();
```

Выражение BETWEEN:

```
<?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();
```

Использование функций DQL в выражении WHERE:

```
<?php

$query = $em->createQuery("SELECT u.name FROM CmsUser u WHERE TRIM(u.name) =
 'someone'");
$usernames = $query->getResult();
```

Выражение 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();
```

Функция CONCAT():

```
<?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();
```

Ключевое слово EXISTS и связанный с ним подзапрос:

```
<?php

$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE EXISTS (SELECT
 p.phonenumber FROM CmsPhonenumber p WHERE p.user = u.id)');
$ids = $query->getResult();
```

Получение всех пользователей, являющихся членами группы **$group**:

```
<?php

$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE :groupId MEMBER OF
 u.groups');
$query->setParameter('groupId', $group);
$ids = $query->getResult();
```

Получение всех пользователей, имеющих более одного телефонного номера:

```
<?php

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE SIZE(u.phonenumbers) >
 1');
$users = $query->getResult();
```

Пользователи, не имеющие ни одного номера:

```
<?php

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.phonenumbers IS EMPTY');
$users = $query->getResult();
```

Выборка с учетом иерархии наследования, в примере ниже показано получение экземпляров заданного класса, являющихся потомками другого класса:

```
<?php
```

```
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company
\CompanyPerson u WHERE u INSTANCE OF Doctrine\Tests\Models\Company
\CompanyEmployee');
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company
\CompanyPerson u WHERE u INSTANCE OF ?1');
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company
\CompanyPerson u WHERE u NOT INSTANCE OF ?1');
```

СИНТАКСИС PARTIAL ОБЪЕКТОВ

Обычно когда вы запрашиваете не все, а только какие-то определенные поля, нет необходимости вытаскивать весь объект. Вместо этого, можно запросить только массив в виде обычного плоского набора, аналогично тому как если бы для получения данных вы использовали напрямую язык SQL вместе с JOIN.

Когда нужно получить **partial** объекты, нужно использовать одноименное ключевое слово:

```
<?php

$query = $em->createQuery('SELECT partial u.{id, username} FROM CmsUser u');
$users = $query->getResult(); // массив неполных объектов CmsUser
```

Аналогично это работает и с JOIN'ами:

```
<?php

$query = $em->createQuery('SELECT partial u.{id, username}, partial a.{id, name}
 FROM CmsUser u JOIN u.articles a');
$users = $query->getResult(); // массив неполных объектов CmsUser
```

14.2.5. Использование INDEX BY

Конструкция INDEX BY никак не транслируется в SQL, она затрагивает лишь объекты и генерацию результирующего набора. Что именно имеется ввиду? После предложение FROM и JOIN можно указать по какому полю будет индексироваться этот класс в результирующем наборе. По умолчанию в качестве ключей выступают целочисленные инкрементные значения начиная с нуля. Но с помощью INDEX BY можно назначить любую колонку в качестве ключа, хотя делать это имеет смысл только для первичных или уникальных ключей:

```
SELECT u.id, u.STATUS, UPPER(u.name) nameUpper FROM USER u INDEX BY u.id
```

```
JOIN u.phonenumbers p INDEX BY p.phonenumber
```

Такой запрос возвратит набор, индексированный сразу по **user.id** и **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)
```

14.3. Запросы UPDATE

DQL умеет не только получать данные, но и менять их (кто бы мог подумать). Работа оператора UPDATE полностью предсказуема и работает как показано в примере ниже: UPDATE MyProject\Model\USER u SET u.password = 'new' WHERE u.id IN (1, 2, 3) Ссылаться на связанные сущность можно только в предложении WHERE или используя под-запросы.

> DQL UPDATE транслируется напрямую в SQL UPDATE и, таким образом, обходит любые схемы блокировки, события и не увеличивает номер версии. Сущности, которые были ранее загружены из базы и являются PERSISTED-сущностями, не будут автоматически синхронизированы с актуальными данными в базе. Чтобы сделать это рекомендуется каждый раз вызывать метод **EntityManager#clear()** и заново получать экземпляры затронутой сущности.

14.4. Запросы DELETE

Запросы DELETE имеют такой же простой синтаксис как и UPDATE:

```
DELETE MyProject\Model\USER u WHERE u.id = 4
```

На связанные сущности накладывается такие же ограничения.

> DQL запросы DELETE транслируются напрямую в одноименный SQL, исключая, т.о., реакцию на события и выполнение проверки для столбца с версией, если они не были явно добавлены в предложение WHERE. Кроме того, удаление не распространяется каскадно на связанные сущности, даже если это явно указано в метаданных.

14.5. Функции, операторы и аггрегации

14.5.1. Функции в DQL

В предложениях SELECT, WHERE и HAVING поддерживаются следующие функции:

- ABS(arithmetic_expression)
- CONCAT(str1, str2)
- CURRENT_DATE() – текущая дата
- CURRENT_TIME() – текущее время
- CURRENT_TIMESTAMP()
- LENGTH(str) – длина строки
- LOCATE(needle, haystack [, offset]) – позиция первого вхождения подстроки
- LOWER(str) – перевод в нижний регистр.
- MOD(a, b) – остаток от деления a на b.
- SIZE(collection) – количество элементов в коллекции
- SQRT(q) – квадратный корень.
- SUBSTRING(str, start [, length]) – подстрока.
- TRIM([LEADING | TRAILING | BOTH] ['trchar' FROM] str) – Удалении оконечных пробелов.
- UPPER(str) – перевод в верхний регистр.

- DATE_ADD(date, days, unit) – добавляет к дате заданное количество дней (доступные единицы измерения: DAY, MONTH)
- DATE_SUB(date, days, unit) – вычитание дней из даты
- DATE_DIFF(date1, date2) – Разница в днях между двумя датами

14.5.2. Арифметические операторы

В DQL допускает использование арифметических выражений:

```
SELECT person.salary * 1.5 FROM CompanyPerson person WHERE person.salary < 100000
```

14.5.3. Агрегатные функции

В предложениях SELECT и GROUP BY можно использовать следующие функции: AVG, COUNT, MIN, MAX, SUM

14.5.4. Другие выражения

Помимо всего вышеперечисленного, в DQL есть довольно широкий набор различных выражений, пришедший из SQL, например:

- **ALL/ANY/SOME** – при использовании в выражении WHERE сразу после подзапроса работает как и его эквивалент в SQL.
- **BETWEEN a AND b** и **NOT BETWEEN a AND b** для проверки попадания значения в заданный интервал.
- **IN (x1, x2, …)** и **NOT IN (x1, x2, ..)** для проверки вхождения значения в заданный набор.
- **LIKE** .. и **NOT LIKE** .. сравнение строк.
- **IS NULL** и **IS NOT NULL** проверка на NULL
- **EXISTS** и **NOT EXISTS** в связке с подзапросами

14.5.5. Создание пользовательских функций

По умолчанию DQL имеет в своем арсенале общий набор функций, поддерживаемых многими СУБД. Однако, чаще всего база выбирается раз и навсегда. В таком

случае можно расширить синтаксис DQL функциями, ориентированными на конкретную платформу.

Регистрация пользовательских функций осуществляется через объект Configuration:

```
<?php

$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($name, $class);
$config->addCustomNumericFunction($name, $class);
$config->addCustomDatetimeFunction($name, $class);

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

В зависимости от типа функции она может возвращать строку, число или дату и время. Давайте в качестве примера добавим специфичную для MySQL функцию FLOOR(). Все классы нужно наследовать от базового:

```
<?php

namespace MyProject\Query\AST;

use \Doctrine\ORM\Query\AST\Functions\FunctionNode;
use \Doctrine\ORM\Query\Lexer;

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 $parser)
    {
        $lexer = $parser->getLexer();

        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);

        $this->simpleArithmeticExpression = $parser->SimpleArithmeticExpression();
```

```
        $parser->match(Lexer::T_CLOSE_PARENTHESIS);
    }
}
```

Зарегистрируем функцию, после чего она станет доступна прямо в DQL запросе:

```
<?php

\Doctrine\ORM\Query\Parser::registerNumericFunction('FLOOR', 'MyProject\Query
\MysqlFloor');
$dql = "SELECT FLOOR(person.salary * 1.75) FROM CompanyPerson person";
```

14.6. Запросы к унаследованным классам

В этой главе рассказывается как строить запросы к унаследованным классам и какой результат при этом ожидать.

14.6.1. Одиночная таблица

Стратегия Single Table Inheritance[2] заключается в том, что все классы в иерархии соответствуют одной единственной таблице. И чтобы различать какая запись какому классу соответствует используется так называемый **столбец дискриминатора**.

Чтобы показать как это работает давайте для начала подготовим набор сущностей. Возьмем сущности Person и Employee:

```
<?php

namespace Entities;

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

[2] http://martinfowler.com/eaaCatalog/singleTableInheritance.html

```
     * @GeneratedValue
     */
    protected $id;

    /*
     * @Column(type="string", length=50)
     */
    protected $name;

    // ...
}

/*
 * @Entity
 */
class Employee extends Person
{
    /*
     * @Column(type="string", length=50)
     */
    private $department;

    // ...
}
```

Обратите внимание как будет выглядеть SQL запрос на создание таблиц для этих сущностей, а таблица-то всего одна:

```
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
)
```

И теперь каждый раз когда будет сохраняться экземпляр сущности Employee, автоматом будет заполняться столбец дискриминатора:

```
<?php

$employee = new \Entities\Employee();
$employee->setName('test');
$employee->setDepartment('testing');
$em->persist($employee);
$em->flush();
```

Теперь напишем запрос, который достанет только что сохраненную сущность Employee из базы:

```
SELECT e FROM Entities\Employee e WHERE e.name = 'test'
```

Если посмотреть на нативный SQL запрос, можно заметить специальное условие, которое гарантирует, что из базы будет возвращена именно сущность типа Employee:

```
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')
```

14.6.2. Class Table Inheritance

Стратегия Class Table Inheritance[3] справедлива когда каждый класс в иерархии соответствует нескольким таблицам: его собственной и таблицам его родительских классов. Таким образом, таблица дочернего класса будет связана с таблицей родительского класса посредством внешнего ключа. В Doctrine 2 имплементирует эту стратегию посредством использования столбца дискриминатора у самой верхней таблицы в иерархии, потому что в контексте этой стратегии это наипростейший из способов для возможности работы полиморфных запросов.

Пример ниже аналогичен наследования от единственной таблицы, нужно только поменять тип наследования с **SINGLE_TABLE** на **JOINED**:

```
<?php

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

Посмотрите на SQL запрос, который создает таблицы, обратите внимание на различия с предыдущим примером:

[3] http://martinfowler.com/eaaCatalog/classTableInheritance.html

```
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
```

- Данные бьются на две таблицы.
- Таблицы связаны внешним ключом

Теперь если попробовать сохранить ту же сущность Employee как мы делали для SINGLE_TABLE, а затем запросить ее из базы, то итоговый SQL запрос будет выглядеть иначе, в нем будет автоматически подсоединена таблица, несущая в себе информацию о типе Person:

```
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 = ?
```

14.7. Класс Query

Любой запрос DQL всегда будет представлен экземпляром класса *Doctrine\ORM \Query*. Этот экземпляр создается при вызове**EntityManager#createQuery($dql)** куда вы передаете строку с запросом. Либо можно создать пустой экземпляр Query, а затем вызвать метод **Query#setDql($dql)**. Пару примеров:

```
<?php

// $em это эклемпляр EntityManager

// example1: Передача строки с DQL
$q = $em->createQuery('select u from MyProject\Model\User u');

// example2: то же при помощи setDql
$q = $em->createQuery();
```

```
$q->setDql('select u from MyProject\Model\User u');
```

14.7.1. Форматы результата запросов

Формат в котором будет возвращен результат DQL запроса SELECT может быть определен с помощью так называемого режима гидрации (**hidration mode**). Режим гидрации определяет каким способом будет подготовлен SQL запрос. Для каждого типа гидрации есть свой отдельный метода в классе Query.

Вот они:

Query#getResult(): Возвращает коллекцию объектов. Результатом может быть либо коллекция объектов (простой) либо массив, в котором объекты вложены в строки результатов запроса (смешанный).

Query#getSingleResult(): Возвращает один объект. Если в результате запроса содержится более одного обхекте или объект отсутствует, будет выброшено исключение. Нет разницы простой это результат или смешанный.

Query#getOneOrNullResult(): Возвращает один объект Если объект отсутствует будет возвращено значение NULL.

Query#getArrayResult(): Возвращает массив графов (вложенный масив), который в значительной степени взаимозаменяем с графом объектов, возвращаемых методом **Query#getResult()**, но только для чтения.

> В некоторых случаях граф массивов может отличаться от соответствующего графа объектов из-за отличия в семантике между массивами и объектами.

Query#getScalarResult(): Возвращает плоский/прямоугольный результирующий набор, который может содержать повторяющиеся данные. Нет разницы простой это результат или смешанный.

Query#getSingleScalarResult(): Возвращает единственное скалярное значение из результата, возвращаемого СУБД. Если результат содержит более одного такого значения, будет выброшено исключение. Нет разницы простой это результат или смешанный.

Вместо этих методом можно воспользоваться универсальным методом* Query#execute(array $params = array(), $hydrationMode = Query::HYDRATE_OBJECT)**.**

В нем можно явно указать метод гидрации. Фактически, все вышеприведенные методы это лишь сокращения для метода *Query#execute. Например, **Query#getResult()** внутри себя вызывает**Query#execute**, передавая Query::HYDRATE_OBJECT в качестве метода гидрации.

С целью удобства лучше применять вышеприведенные методы вместо execute.

14.7.2. Простые (Pure) и смешанные (Mixed) результаты

DQL запрос SELECT, вызыванный с помощью методов **Query#getResult()** и **Query#getArrayResult()** может возвращать результат в двух формах: простой (pure) и смешанной (mixed). В предыдущих примерах вы уже видели простую форму — это просто массив объектов. По умолчанию, результат возвращается в простой форме, но если в предложении SELECT будут присутствовать скалярные значения, не относящиеся к сущности, такие как агрегации и т.д., результат будет представлен в смешанной форме. Смешанный результат имеет иную структуру, чтобы вмещать в себя скалярные значение.

Простой результат обычно выглядит так:

```
$dql = "SELECT u FROM User u";

array
    [0] => Object
    [1] => Object
    [2] => Object
    ...
```

Смешанный результат имеет структуру следующего формата:

```
$dql = "SELECT u, 'some scalar string', count(u.groups) AS num FROM User u JOIN
 u.groups g GROUP BY u.id";

array
    [0]
        [0] => Object
        [1] => "some scalar string"
        ['num'] => 42
        // ... здесь идут другие скалярные значение, индексируемые числовым
 способом или по имени
    [1]
        [0] => Object
        [1] => "some scalar string"
```

```
        ['num'] => 42
        // ... здесь идут другие скалярные значение, индексируемые числовым
 способом или по имени
```

Чтобы лучше понять суть смешанных результатов рассмотрим следующий DQL-запрос:

```
SELECT u, UPPER(u.name) nameUpper FROM MyProject\Model\USER u
```

В запросе используется функция UPPER, которая возвращает скалярное значение. Таким образом, в предложении SELECT будет присотствовать не только сущность но и скалярное знаение, поэтому результат будет смешанным.

Несколько нюансов о смешанных результатах:

- Объект, запрашиваемый в предложении FROM всегда будет доступен по ключу ’0′.
- Каждое скалярное значение без имени будет пронумеровано по порядку его нахождения в запросе, начиная с единицы.
- Скалярные значения, имеющие псевдоним будут доступны по ключу, значением которого является этот псевдоним. Регистр имени сохраняется.
- Если в предложении FROM указано несколько объектов они будут чередоваться в каждой строке.

Ниже показано как может выглядеть результат запроса:

```
array
    array
        [0] => User (Object)
        ['nameUpper'] => "ROMAN"
    array
        [0] => User (Object)
        ['nameUpper'] => "JONATHAN"
    ...
```

И как получить доступ к его элементам из PHP кода:

```
<?php

foreach ($results as $row) {
    echo "Name: " . $row[0]->getName();
    echo "Name UPPER: " . $row['nameUpper'];
}
```

14.7.3. Несколько сущностей в предложени FROM

В случае если вы делаете выборку сразу нескольких сущностей, указывая их в предложении FROM, в результирующий набор сущности будут чередоваться по строкам. Вот как это выглядит:

```
$dql = "SELECT u, g FROM User u, Group g";

array
    [0] => Object (User)
    [1] => Object (Group)
    [2] => Object (User)
    [3] => Object (Group)
```

14.7.4. Методы гидрации

Каждый из режимов гидрации делает предположение о том, в каком виде результат должен быть возвращен конечному пользователю. Чтобы умет ьвыбирать нужный формат для результата запроса следует понимать все детали гидрации:

Каждый режим представлен соответствующей константов:

- Query::HYDRATE_OBJECT
- Query::HYDRATE_ARRAY
- Query::HYDRATE_SCALAR
- Query::HYDRATE_SINGLE_SCALAR

OBJECT HYDRATION

Object hydration оформляет результирующий набор в виде графа объектов:

```
<?php

$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_OBJECT);
```

ARRAY HYDRATION

Примерно тоже самое, результатом будет тот же граф объектов, представленный в виде массива:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_ARRAY);
```

Тоже самое можно сделать одним коротким вызовом метода getArrayResult():

```
<?php

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

SCALAR HYDRATION

Возвращает простой прямоугольный набор вместо графа:

```
<?php

$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_SCALAR);
echo $users[0]['u_id'];
```

при скалярной гидрации делаются следующие предположения относительно выбираемых полей: К полям классов будет добавлен префикс в виде псевдонима DQL. Результирующий набор для запроса вида ‘SELECT u.name ..’ будет содержать ключ ‘u_name’.

SINGLE SCALAR HYDRATION

Для случая когда запрос возвращает единственное скалярное значение:

```
<?php

$query = $em->createQuery('SELECT COUNT(a.id) FROM CmsUser u LEFT JOIN u.articles a
 WHERE u.username = ?1 GROUP BY u.id');
$query->setParameter(1, 'jwage');
$numArticles = $query->getResult(Query::HYDRATE_SINGLE_SCALAR);
```

В качестве сокращения можно использовать метод getSingleScalarResult():

```
<?php

$numArticles = $query->getSingleScalarResult();
```

ПОЛЬЗОВАТЕЛЬСКИЕ РЕЖИМЫ ГИДРАЦИИ

Для создание своих методов гидрации нужно создать класс, унаследовав его от *AbstractHydrator*:

```
<?php

namespace MyProject\Hydrators;

use Doctrine\ORM\Internal\Hydration\AbstractHydrator;

class CustomHydrator extends AbstractHydrator
{
    protected function _hydrateAll()
    {
        return $this->_stmt->fetchAll(PDO::FETCH_ASSOC);
    }
}
```

Затем нужно добавить этот класс в конфикурацию ORM:

```
<?php

$em->getConfiguration()->addCustomHydrationMode('CustomHydrator', 'MyProject
\Hydrators\CustomHydrator');
```

Теперь гидратор доступен для использования в запросах:

```
<?php

$query = $em->createQuery('SELECT u FROM CmsUser u');
$results = $query->getResult('CustomHydrator');
```

14.7.5. Итерирование по огромным результирующим наборам

Иногда бывает, что запрос возвращает такой неебически большой объем данных, что его нельзя просто взять и обработать. Какой бы режим гидрации вы не использовали, все они загружают весь результирующий набор целиком в память, что может быть недопустимо при работе с большим объемом данных. В главе Пакетная обработка[4] описано как работать с большими массивами данных.

[4] http://odiszapc.ru/doctrine/batch-processing

14.7.6. Функции

У класса AbstractQuery, от которого наследуются Query и NativeQuery есть следующие методы.

ПАРАМЕТРЫ

Выражения, использующие именованные wildcards, для выполнения требуют дополнительных параметров. Передать парамтеры в запрос можно с помощью следующих методов:

- AbstractQuery::setParameter($param, $value) – Устанавливает значение для численного или именованного wildcard.
- AbstractQuery::setParameters(array $params) – Устанавливает параметры, получая массив а виде пар ключ-значение.
- AbstractQuery::getParameter($param)
- AbstractQuery::getParameters()

Парадавать именованные и позиционные параметры в эти методы нужно без префиксов **?** или **:**.

API ДЛЯ УПРАВЛЕНИЕ КЕШЕМ

Кешировать результаты запросов можно как на основе переменных (SQL, режим гидрации, параметры, Hints) или пользовательских ключей. По умолчанию результаты запроса не кешируются. Включить кеш можно персонально для каждого запроса. В следующем примере показано как работать с Result Cache API:

```
<?php

$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.id = ?1');
$query->setParameter(1, 12);

$query->setResultCacheDriver(new ApcCache());

$query->useResultCache(true)
      ->setResultCacheLifeTime($seconds = 3600);

$result = $query->getResult(); // промах
```

```
$query->expireResultCache(true);
$result = $query->getResult(); // игнорирование кеша, снова промах

$query->setResultCacheId('my_query_result');
$result = $query->getResult(); // результат сохранен под ключом 'my_query_result'

// Либо можно вызвать useResultCache() со всем параметрами:
$query->useResultCache(true, $seconds = 3600, 'my_query_result');
$result = $query->getResult(); // cache hit!

// Интроспекция
$queryCacheProfile = $query->getQueryCacheProfile();
$cacheDriver = $query->getResultCacheDriver();
$lifetime = $query->getLifetime();
$key = $query->getCacheKey();
```

> Установить драйвер кеша можно глобально в экземпляре класса Doctrine\ORM\Configuration, т.о. он будет передан во все экземпляры Query и NativeQuery.

ПОДСКАЗКИ

С помощью метода **AbstractQuery::setHint($name, $value)** в гидраторы и парсер запроса можно передавать подсказки (hints). Сейчас в основном существуют внтуренние подсказки, которые не используются пользователями библиотеки. Однако, следующие несколько можно применять:

- Query::HINT_FORCE_PARTIAL_LOAD – позволяет производить гидрацию даже если из базы данных были получены не все столбцы. Этот хинт помогает снизить потребление памяти для больших массивов данных. В Doctrine нет функционала для неявной перезагрузки таких данных. Чтобы заново подгрузить такие объекты из базы данных их нужно передать методу EntityManager::refresh().
- Query::HINT_REFRESH – этот хинт используется внутри метода EntityManager::refresh(), но также может быть использован и в обычном режиме. Как он работает: когда вы загружаете из БД данные сущности, которая уже находится под управлением UnitOfWork, то поля этой сущности будут обновлены. В обычном режиме предпочтение отдается данным существующей сущности, а результирующий набор отклоняется.
- Query::HINT_CUSTOM_TREE_WALKERS – Массив дополнительных эклемпляров Doctrine\ORM\Query\TreeWalker, подключаемых к процессу синтаксического разбора DQL запроса.

КЕШ ЗАПРОСОВ (ТОЛЬКО ДЛЯ DQL ЗАПРОСОВ)

Думаю всем очевидно, что разбор DQL запросов и последующая их трансформация в SQL несут в себе ряд издержек по сравнению с выполнением обычных SQL запросов. Поэтому мы существует специальный кеш, где хранится результат синтакцсического разбора каждого DQL запроса. Если использовать wildcards в запросах, то синтаксический анализ можно вообще свести на нет – все будет браться из кеша.

Каждому экземпляру *Doctrine\ORM\Query* драйвер кеша запросов передается из экземпляра *Doctrine\ORM\Configuration instance to each Doctrine\ORM\Query*, кроме того он включен по умолчанию. Так что обычно не стоит заморачиваться с опциями этого кеша, однако, если все же захотите, то следующие методы помогут вам:

- Query::setQueryCacheDriver($driver) – позволяет установить экземпляр драйвера кеша.
- Query::setQueryCacheLifeTime($seconds = 3600) – устанавливает время жизни кеша.
- Query::expireQueryCache($bool) – если установлен в TRUE принудительно инвалидирует кеш, отключая его.
- Query::getExpireQueryCache()
- Query::getQueryCacheDriver()
- Query::getQueryCacheLifeTime()

ПЕРВЫЙ И МАКСИМАЛЬНЫЙ ЭЛЕМЕНТЫ В РЕЗУЛЬТИРУЮЩЕМ НАБОРЕ (ТОЛЬКО ДЛЯ DQL)

Можно делать срез по результататм DQL запросов (limit и offset):

- Query::setMaxResults($maxResults)
- Query::setFirstResult($offset)

> Если в запросе присутствует подключаемая fetch-joined коллекция, вышеприведенные методы будут работать не так как ожидается. setMaxResults просто ограничивает число строк результата, в то время как при использовании fetch-joined коллекций одна и та же ведущая сущность может появляться в различных строках, финальный результат после гидрации будет меньше заданного числа строк.

ВРЕМЕННОЕ ИЗМЕНЕНИЕ РЕЖИМА ВЫБОРКИ В DQL

Обычно все связи помечены ка к lazy или extra lazy, однако в некоторых случаях из за высокой стоимость операции JOIN не нужно включать в результирующий набор остальные сущности через fetch join. Поэтому такие связи Many-To-One или One-To-One можно пометить соответствующим образом для пакетной обработки с помощью конструкции WHERE .. IN.

```
<?php

$query = $em->createQuery("SELECT u FROM MyProject\User u");
$query->setFetchMode("MyProject\User", "address", "EAGER");
$query->execute();
```

Допустим в БД лежат 10 пользователей и соответствующие им адреса, тогда запрос будет выглядеть так:

```
SELECT * FROM users;
SELECT * FROM address WHERE id IN (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

14.8. EBNF

Следующая контекстно-свободная грамматика, представленная в форме EBNF описывает язык DQL. К ней можно обращаться если вам не очевидны те или иные стороны DQL или непонятен синтаксис того или иного запроса.

14.8.1. Document syntax:

- non-terminals begin with an upper case character
- terminals begin with a lower case character
- parentheses (…) are used for grouping
- square brackets […] are used for defining an optional part, e.g. zero or one time
- curly brackets {…} are used for repetition, e.g. zero or more times
- double quotation marks "…" define a terminal string a vertical bar | represents an alternative

14.8.2. Terminals

- identifier (name, email, …)

- string (‘foo’, ‘bar’’s house’, ‘%ninja%’, …)
- char (‘/’, ‘\’, ‘ ‘, …)
- integer (-1, 0, 1, 34, …)
- float (-0.23, 0.007, 1.245342E+8, …)
- boolean (false, true)

14.8.3. Query Language

```
QueryLanguage ::= SelectStatement | UpdateStatement | DeleteStatement
```

14.8.4. Statements

```
SelectStatement ::= SelectClause FromClause [WhereClause] [GroupByClause]
 [HavingClause] [OrderByClause]
UpdateStatement ::= UpdateClause [WhereClause]
DeleteStatement ::= DeleteClause [WhereClause]
```

14.8.5. Identifiers

```
/* 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 simple field */
FieldIdentificationVariable ::= identifier
```

```
/* identifier that must be a collection-valued association field (to-many) (the
 "Phonenumbers" of "u.Phonenumbers") */
```

```
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 to a single field (not mapping to a relation) */
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
```

14.8.6. Path Expressions

```
/* "u.Group" or "u.Phonenumbers" declarations */
JoinAssociationPathExpression             ::= IdentificationVariable
 "." (CollectionValuedAssociationField | SingleValuedAssociationField)
```

```
/* "u.Group" or "u.Phonenumbers" usages */
AssociationPathExpression                 ::= CollectionValuedPathExpression |
 SingleValuedAssociationPathExpression
```

```
/* "u.name" or "u.Group" */
SingleValuedPathExpression                ::= StateFieldPathExpression |
 SingleValuedAssociationPathExpression
```

```
/* "u.name" or "u.Group.name" */
```

```
StateFieldPathExpression                  ::= IdentificationVariable "." StateField
 | SingleValuedAssociationPathExpression "." StateField
```

```
/* "u.Group" */
SingleValuedAssociationPathExpression     ::= IdentificationVariable "."
 SingleValuedAssociationField
/* "u.Group.Permissions" */
CollectionValuedPathExpression            ::= IdentificationVariable
 "." {SingleValuedAssociationField "."}* CollectionValuedAssociationField
```

```
/* "name" */
StateField                                ::= {EmbeddedClassStateField "."}*
 SimpleStateField
```

```
/* "u.name" or "u.address.zip" (address = EmbeddedClassStateField) */
SimpleStateFieldPathExpression            ::= IdentificationVariable "." StateField
```

14.8.7. Clauses

```
SelectClause        ::= "SELECT" ["DISTINCT"] SelectExpression {","
 SelectExpression}*
SimpleSelectClause  ::= "SELECT" ["DISTINCT"] SimpleSelectExpression
UpdateClause        ::= "UPDATE" AbstractSchemaName ["AS"]
 AliasIdentificationVariable "SET" UpdateItem {"," UpdateItem}*
DeleteClause        ::= "DELETE" ["FROM"] AbstractSchemaName ["AS"]
 AliasIdentificationVariable
FromClause          ::= "FROM" IdentificationVariableDeclaration {","
 IdentificationVariableDeclaration}*
SubselectFromClause ::= "FROM" SubselectIdentificationVariableDeclaration {","
 SubselectIdentificationVariableDeclaration}*
WhereClause         ::= "WHERE" ConditionalExpression
HavingClause        ::= "HAVING" ConditionalExpression
GroupByClause       ::= "GROUP" "BY" GroupByItem {"," GroupByItem}*
OrderByClause       ::= "ORDER" "BY" OrderByItem {"," OrderByItem}*
Subselect           ::= SimpleSelectClause SubselectFromClause [WhereClause]
 [GroupByClause] [HavingClause] [OrderByClause]
```

14.8.8. Items

```
UpdateItem  ::= IdentificationVariable "." (StateField |
 SingleValuedAssociationField) "=" NewValue
```

```
OrderByItem ::= (ResultVariable | SingleValuedPathExpression) ["ASC" | "DESC"]
GroupByItem ::= IdentificationVariable | SingleValuedPathExpression
NewValue    ::= ScalarExpression | SimpleEntityExpression | "NULL"
```

14.8.9. From, Join and Index by

```
IdentificationVariableDeclaration          ::= RangeVariableDeclaration [IndexBy]
 {JoinVariableDeclaration}*
SubselectIdentificationVariableDeclaration ::= IdentificationVariableDeclaration |
 (AssociationPathExpression ["AS"] AliasIdentificationVariable)
JoinVariableDeclaration                    ::= Join [IndexBy]
RangeVariableDeclaration                   ::= AbstractSchemaName ["AS"]
 AliasIdentificationVariable
Join                                       ::= ["LEFT" ["OUTER"] | "INNER"] "JOIN"
 JoinAssociationPathExpression
                                               ["AS"] AliasIdentificationVariable
 ["WITH" ConditionalExpression]
IndexBy                                    ::= "INDEX" "BY"
 SimpleStateFieldPathExpression
```

14.8.10. Select Expressions

```
SelectExpression        ::= IdentificationVariable | PartialObjectExpression
 | (AggregateExpression | "(" Subselect ")"  | FunctionDeclaration |
 ScalarExpression) [["AS"] AliasResultVariable]
SimpleSelectExpression  ::= ScalarExpression | IdentificationVariable |
                            (AggregateExpression [["AS"] AliasResultVariable])
PartialObjectExpression ::= "PARTIAL" IdentificationVariable "." PartialFieldSet
PartialFieldSet         ::= "{" SimpleStateField {"," SimpleStateField}* "}"
```

14.8.11. Conditional Expressions

```
ConditionalExpression       ::= ConditionalTerm {"OR" ConditionalTerm}*
ConditionalTerm             ::= ConditionalFactor {"AND" ConditionalFactor}*
ConditionalFactor           ::= ["NOT"] ConditionalPrimary
ConditionalPrimary          ::= SimpleConditionalExpression | "("
 ConditionalExpression ")"
SimpleConditionalExpression ::= ComparisonExpression | BetweenExpression |
 LikeExpression |
                                InExpression | NullComparisonExpression |
 ExistsExpression |
                                EmptyCollectionComparisonExpression |
 CollectionMemberExpression |
```

```
                                InstanceOfExpression
```

14.8.12. Collection Expressions

```
EmptyCollectionComparisonExpression ::= CollectionValuedPathExpression "IS" ["NOT"]
 "EMPTY"
CollectionMemberExpression          ::= EntityExpression ["NOT"] "MEMBER" ["OF"]
 CollectionValuedPathExpression
```

14.8.13. Literal Values

```
Literal     ::= string | char | integer | float | boolean
InParameter ::= Literal | InputParameter
```

14.8.14. Input Parameter

```
InputParameter      ::= PositionalParameter | NamedParameter
PositionalParameter ::= "?" integer
NamedParameter      ::= ":" string
```

14.8.15. Arithmetic Expressions

```
ArithmeticExpression       ::= SimpleArithmeticExpression | "(" Subselect ")"
SimpleArithmeticExpression ::= ArithmeticTerm {("+" | "-") ArithmeticTerm}*
ArithmeticTerm             ::= ArithmeticFactor {("*" | "/") ArithmeticFactor}*
ArithmeticFactor           ::= [("+" | "-")] ArithmeticPrimary
ArithmeticPrimary          ::= SingleValuedPathExpression | Literal | "("
 SimpleArithmeticExpression ")"
                               | FunctionsReturningNumerics | AggregateExpression |
 FunctionsReturningStrings
                               | FunctionsReturningDatetime |
 IdentificationVariable | InputParameter | CaseExpression
```

14.8.16. Scalar and Type Expressions

```
ScalarExpression       ::= SimpleArithmeticExpression | StringPrimary |
 DateTimePrimary | StateFieldPathExpression
                           BooleanPrimary | EntityTypeExpression | CaseExpression
StringExpression       ::= StringPrimary | "(" Subselect ")"
```

```
StringPrimary          ::= StateFieldPathExpression | string | InputParameter |
 FunctionsReturningStrings | AggregateExpression | CaseExpression
BooleanExpression      ::= BooleanPrimary | "(" Subselect ")"
BooleanPrimary         ::= StateFieldPathExpression | boolean | InputParameter
EntityExpression       ::= SingleValuedAssociationPathExpression |
 SimpleEntityExpression
SimpleEntityExpression ::= IdentificationVariable | InputParameter
DatetimeExpression     ::= DatetimePrimary | "(" Subselect ")"
DatetimePrimary        ::= StateFieldPathExpression | InputParameter |
 FunctionsReturningDatetime | AggregateExpression
```

Parts of CASE expressions are not yet implemented.

14.8.17. Aggregate Expressions

```
AggregateExpression ::= ("AVG" | "MAX" | "MIN" | "SUM") "(" ["DISTINCT"]
 StateFieldPathExpression ")" |
                        "COUNT" "(" ["DISTINCT"] (IdentificationVariable |
 SingleValuedPathExpression) ")"
```

14.8.18. Условия

```
CaseExpression        ::= GeneralCaseExpression | SimpleCaseExpression |
 CoalesceExpression | NullifExpression
GeneralCaseExpression ::= "CASE" WhenClause {WhenClause}* "ELSE" ScalarExpression
 "END"
WhenClause            ::= "WHEN" ConditionalExpression "THEN" ScalarExpression
SimpleCaseExpression  ::= "CASE" CaseOperand SimpleWhenClause {SimpleWhenClause}*
 "ELSE" ScalarExpression "END"
CaseOperand           ::= StateFieldPathExpression | TypeDiscriminator
SimpleWhenClause      ::= "WHEN" ScalarExpression "THEN" ScalarExpression
CoalesceExpression    ::= "COALESCE" "(" ScalarExpression {"," ScalarExpression}*
 ")"
NullifExpression      ::= "NULLIF" "(" ScalarExpression "," ScalarExpression ")"
```

14.8.19. Другие выражения

```
QUANTIFIED/BETWEEN/COMPARISON/LIKE/NULL/EXISTS
QuantifiedExpression     ::= ("ALL" | "ANY" | "SOME") "(" Subselect ")"
BetweenExpression        ::= ArithmeticExpression ["NOT"] "BETWEEN"
 ArithmeticExpression "AND" ArithmeticExpression
ComparisonExpression     ::= ArithmeticExpression ComparisonOperator
 ( QuantifiedExpression | ArithmeticExpression )
```

```
InExpression             ::= StateFieldPathExpression ["NOT"] "IN" "(" (InParameter
 {"," InParameter}* | Subselect) ")"
InstanceOfExpression     ::= IdentificationVariable ["NOT"] "INSTANCE" ["OF"]
 (InstanceOfParameter | "(" InstanceOfParameter {"," InstanceOfParameter}* ")")
InstanceOfParameter      ::= AbstractSchemaName | InputParameter
LikeExpression           ::= StringExpression ["NOT"] "LIKE" string ["ESCAPE" char]
NullComparisonExpression ::= (SingleValuedPathExpression | InputParameter)
 "IS" ["NOT"] "NULL"
ExistsExpression         ::= ["NOT"] "EXISTS" "(" Subselect ")"
ComparisonOperator       ::= "=" | "<" | "<=" | "<>" | ">" | ">=" | "!="
```

14.8.20. Функции

```
FunctionDeclaration ::= FunctionsReturningStrings | FunctionsReturningNumerics |
 FunctionsReturningDateTime
```

```
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 ")"
```

Глава 15. Создание запросов с помощью QueryBuilder

QueryBuilder — это API, который позволяет в несколько шагов создать практически любой DQL запрос.

В составе QueryBuilder входит набор классов и методов для программного построения запросов и весьма гибкий API. Использовать ли построитель или писать запросы вручную решать вам.

15.1. Создание объекта QueryBuilder

Точно так же как вы создавали обычный запрос, создается и объект QueryBuilder. Сразу пример:

```
<?php

// $em -- экземпляр EntityManager

// Пример 1: создание экземпляра QueryBuilder
$qb = $em->createQueryBuilder();
```

Экземпляр QueryBuilder имеет набор функций, названия которых говорят сами за себя. Так, например, можно узнать тип объекта QueryBuilder:

```
<?php

// $qb -- экземпляр QueryBuilder

// Пример 2: retrieving type of QueryBuilder
echo $qb->getType(); // Выведет: 0
```

Объект может иметь один из трех возможных типов:

- **QueryBuilder::SELECT**, which returns value 0
- **QueryBuilder::DELETE**, returning value 1
- **QueryBuilder::UPDATE**, which returns value 2

После того как запрос построен можно получить экземпляр менеджера сущностей, текст DQL запроса и сам объект этого запроса:

```
<?php

// $qb -- экземпляр QueryBuilder

// Пример 3: получение экземпляра EntityManager
$em = $qb->getEntityManager();

// Пример 4: получение текста DQL запроса, представленного экземпляром QueryBuilder
$dql = $qb->getDql();

// Пример 5: получение экземпляра объекта Query
$q = $qb->getQuery();
```

Для увеличения производительности в QueryBuilder используется кеш DQL. Любое вносимое в объект запроса изменение, которое может повлиять на текст результирующего запроса переводит QueryBuilder в состояние, которое мы называем**STATE_DIRTY**. Итого, любой экземпляр QueryBuilder может находиться в одном из двух состояний:

- **QueryBuilder::STATE_CLEAN** означает, что DQL запрос находится в актуальном состоянии, объект принимает это состояние сразу после создания
- **QueryBuilder::STATE_DIRTY** означает, что DQL запрос был изменен и должен быть перестроен

15.2. Работа с QueryBuilder

Все вспомогательные методы объекта QueryBuilder являются сокращениями для метода **add()**, который и отвечает за построение DQL запроса. Он принимает три параметра: **$dqlPartName**, **$dqlPart** и **$append** (по умолчанию равен *false*)

- **$dqlPartName**: Определяет место, в котором будет размещен **$dqlPart**. Возможные значения: select, from, where, groupBy, having, orderBy
- **$dqlPart**: Что должно быть размещено в **$dqlPartName**. Принимает строку или любой экземпляр*Doctrine\ORM\Query\Expr**
- **$append**: Необязательный аргумент (по умолчанию равен *false*) должен ли **$dqlPart** переопределять все заданные до этого элементы в **$dqlPartName**

```
<?php
```

```
// $qb -- экземпляр QueryBuilder

// Пример 6: создание запроса "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name
 ASC" с помощью QueryBuilder
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = ?1')
   ->add('orderBy', 'u.name ASC');
```

15.2.1. Параметры

В Doctrine есть возможность привязки параметров к объекту построителя, подобно тому как это делалось при ручном написании запросов (см. предыдущую главу). Можно использовать как строковые так и цифровые метки, хотя синтаксис их немного отличается. Так или иначе, нужно выбрать что-то одно: не допускается смешивать оба стиля. Привязать параметры можно следующим образом:

```
<?php

// $qb - экземпляр QueryBuilder

// Пример 6: определяем запрос: "SELECT u FROM User u WHERE u.id = ? ORDER BY
 u.name ASC" с помощью QueryBuilder
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = ?1')
   ->add('orderBy', 'u.name ASC');
   ->setParameter(1, 100); // Устанавливает параметр ?1 в 100, т.о. будет запрошен
 пользователь, имеющий u.id = 100
```

Не обязательно использовать именно цифровые параметры, есть и другой способ:

```
<?php

// $qb -- экземпляр QueryBuilder

// Пример 6: определяем запрос: "SELECT u FROM User u WHERE u.id = ? ORDER BY
 u.name ASC" с помощью QueryBuilder
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = :identifier')
   ->add('orderBy', 'u.name ASC');
   ->setParameter('identifier', 100); // Устанавливает параметр :identifier в 100,
 т.о. будет запрошен пользователь с u.id = 100
```

Обратите внимание на то, что имена цифровых параметров начинаются со знака '**?**', за которым следует число, а именованных со знака '**:**', за которым следует имя.

Для привязки сразу нескольких параметров можно использовать метод **setParameters()**:

```
<?php

// $qb -- экземпляр QueryBuilder

// Query here...
$qb->setParameters(array(1 => 'value for ?1', 2 => 'value for ?2'));
```

Привязанные ранее параметры можно получить с помощью методов “**getParameter()**” или “**getParameters()**”:

```
<?php

// $qb -- экземпляр QueryBuilder

// Первый способ
$params = $qb->getParameters(array(1, 2));
// Второй способ
$param  = array($qb->getParameter(1), $qb->getParameter(2));
```

При попытка обращения к несуществующему параметру метод **getParameter()** вернет NULL.

15.2.2. Ограничения

Для наложения ограничений на результат запроса можно использовать методы, аналогичные используемым в объекте Query, который можно получит с помощью метода **EntityManager#createQuery()**.

```
<?php

// $qb - экземпляр 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 );
```

15.2.3. Выполнение запроса

QueryBuilder всего лишь строит объект, это вовсе не означает выполнение получившегося запроса. Кроме того, такие вещи как, например, подсказки нельзя задать непосредственно в самом билдере, для этого сначала нужно получить объект Query:

```
<?php

// $qb -- экземпляр QueryBuilder
$query = $qb->getQuery();

// Установка дополнительных параметров
$query->setQueryHint('foo', 'bar');
$query->useResultCache('my_cache_id');

// Выполнение запроса
$result = $query->getResult();
$single = $query->getSingleResult();
$array = $query->getArrayResult();
$scalar = $query->getScalarResult();
$singleScalar = $query->getSingleScalarResult();
```

15.2.4. Классы Expr*

Когда вы вызываете метод add(), передавая ему строковой параметр, в действительность происходит создание экземпляра класса **Doctrine\ORM\Query\Expr \Expr***. Следующий пример демонстрирует это:

```
<?php

// $qb -- экземпляр QueryBuilder

// Приер 7: задаем запрос: "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name
 ASC" с помощью QueryBuilder, используя экземпляры Expr\*
$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'));
```

Конечно, нифига не удобно строить запросы таким хитржопым способом. Поэтому для упрощения есть специальный вспомогательный класс Expr.

15.2.5. Класс Expr

Чтобы обойти различного рода проблемы при использовании метода **add()** в Doctrine есть вспомогательный класс для построения выражений — Expr, для этих целей он содержит ряд полезных методов:

```
<?php

// $qb -- экземпляр QueryBuilder

// Пример 8: QueryBuilder-реализация запроса "SELECT u FROM User u WHERE u.id = ?
 OR u.nickname LIKE ? ORDER BY u.surname DESC" с использованием класса Expr
$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'));
```

Да, выглядит это все еще громоздко, но в этом и есть основная фича класса Expr — дать возможность программно создавать условные выражения. Полный список методов этого класса приведен ниже:

```
<?php

class Expr
{
    /* Условия */

    // Пример -- $qb->expr()->andX($cond1 [, $condN])->add(...)->...
    public function andX($x = null); // Возвращает экземпляр Expr\AndX instance

    // Пример -- $qb->expr()->orX($cond1 [, $condN])->add(...)->...
    public function orX($x = null); // Возвращает экземпляр Expr\OrX instance

    /* Сравнение */

    // Пример -- $qb->expr()->eq('u.id', '?1') => u.id = ?1
    public function eq($x, $y); // Возвращает экземпляр Expr\Comparison

    // Пример -- $qb->expr()->neq('u.id', '?1') => u.id <> ?1
    public function neq($x, $y); // Возвращает экземпляр Expr\Comparison

    // Пример -- $qb->expr()->lt('u.id', '?1') => u.id < ?1
```

```
public function lt($x, $y); // Возвращает экземпляр Expr\Comparison

// Пример -- $qb->expr()->lte('u.id', '?1') => u.id <= ?1
public function lte($x, $y); // Возвращает экземпляр Expr\Comparison

// Пример -- $qb->expr()->gt('u.id', '?1') => u.id > ?1
public function gt($x, $y); // Возвращает экземпляр Expr\Comparison

// Пример -- $qb->expr()->gte('u.id', '?1') => u.id >= ?1
public function gte($x, $y); // Возвращает экземпляр Expr\Comparison

// Пример -- $qb->expr()->isNull('u.id') => u.id IS NULL
public function isNull($x); // Возвращает строку

// Пример -- $qb->expr()->isNotNull('u.id') => u.id IS NOT NULL
public function isNotNull($x); // Возвращает строку

/* Арифметика */

// Пример -- $qb->expr()->prod('u.id', '2') => u.id * 2
public function prod($x, $y); // Возвращает экземпляр Expr\Math

// Пример -- $qb->expr()->diff('u.id', '2') => u.id - 2
public function diff($x, $y); // Возвращает экземпляр Expr\Math

// Пример -- $qb->expr()->sum('u.id', '2') => u.id + 2
public function sum($x, $y); // Возвращает экземпляр Expr\Math

// Пример -- $qb->expr()->quot('u.id', '2') => u.id / 2
public function quot($x, $y); // Возвращает экземпляр Expr\Math

/* Псевдо-функции */

// Пример -- $qb->expr()->exists($qb2->getDql())
public function exists($subquery); // Возвращает экземпляр Expr\Func

// Пример -- $qb->expr()->all($qb2->getDql())
public function all($subquery); // Возвращает экземпляр Expr\Func

// Пример -- $qb->expr()->some($qb2->getDql())
public function some($subquery); // Возвращает экземпляр Expr\Func

// Пример -- $qb->expr()->any($qb2->getDql())
public function any($subquery); // Возвращает экземпляр Expr\Func

// Пример -- $qb->expr()->not($qb->expr()->eq('u.id', '?1'))
public function not($restriction); // Возвращает экземпляр Expr\Func
```

```
    // Пример -- $qb->expr()->in('u.id', array(1, 2, 3))
    // Нельзя вставлять значения напрямую: $qb->expr()->in('value',
 array('stringvalue')), это приведет к выбросу исключения.
    // Вместо этого используйте параметризацию: $qb->expr()->in('value', array('?
1')), затем назначьте параметр ?1 (см. предыдущий раздел)
    public function in($x, $y); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->notIn('u.id', '2')
    public function notIn($x, $y); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->like('u.firstname', $qb->expr()->literal('Gui%'))
    public function like($x, $y); // Возвращает экземпляр Expr\Comparison

    // Пример -- $qb->expr()->between('u.id', '1', '10')
    public function between($val, $x, $y); // Возвращает экземпляр Expr\Func

    /* Функции */

    // Пример -- $qb->expr()->trim('u.firstname')
    public function trim($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->concat('u.firstname', $qb->expr()->concat(' ',
 'u.lastname'))
    public function concat($x, $y); // Возвращает экземпляр Expr\Func

    // Пример- $qb->expr()->substr('u.firstname', 0, 1)
    public function substr($x, $from, $len); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->lower('u.firstname')
    public function lower($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->upper('u.firstname')
    public function upper($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->length('u.firstname')
    public function length($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->avg('u.age')
    public function avg($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->max('u.age')
    public function max($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->min('u.age')
    public function min($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->abs('u.currentBalance')
    public function abs($x); // Возвращает экземпляр Expr\Func
```

```
    // Пример -- $qb->expr()->sqrt('u.currentBalance')
    public function sqrt($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->count('u.firstname')
    public function count($x); // Возвращает экземпляр Expr\Func

    // Пример -- $qb->expr()->countDistinct('u.surname')
    public function countDistinct($x); // Возвращает экземпляр Expr\Func
}
```

15.2.6. Вспомогательные методы

До текущего момента запросы создавались самым сложным низкоуровневым способом. Безусловно, когда дело касается оптимизации это полезно, но в большинстве случаев рекомендуется использовать готовые абстракции. Итак, чтобы сделать построение запросов еще проще можно воспользоваться преимуществом Helper-методов. Давайте рассмотрим их, ниже приведен Пример 6, переписанный с использованием хелперов класса QueryBuilder:

```
<?php

// $qb -- экземпляр QueryBuilder

// Пример 9: создание запроса "SELECT u FROM User u WHERE u.id = ?1 ORDER BY u.name
 ASC" с использованием псмомогательных методов
$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name', 'ASC');
```

Хелперы — это стандартный способ создания запросов. Вообще говоря, старайтесь избегать ручного написания запросов с помощью строк и не слишком вдохновляйтесь методами **$qb→expr()→**. Ниже приведен рефакторинг Примера 8 с использованием описанной парадигмы:

```
<?php

// $qb -- экземпляр QueryBuilder

// Пример 8: QueryBuilder-аналог запроса "SELECT u FROM User u WHERE u.id = ?1 OR
 u.nickname LIKE ?2 ORDER BY u.surname DESC" с использованием хелперов
$qb->select(array('u')) // строка 'u' будет самостоятельно конвертирована в массив
```

```
    ->from('User', 'u')
    ->where($qb->expr()->orX(
        $qb->expr()->eq('u.id', '?1'),
        $qb->expr()->like('u.nickname', '?2')
    ))
    ->orderBy('u.surname', 'ASC'));
```

Полный список полезных вспомогательных методов класса QueryBuilder:

```
<?php

class QueryBuilder
{
    // Пример -- $qb->select('u')
    // Пример -- $qb->select(array('u', 'p'))
    // Пример -- $qb->select($qb->expr()->select('u', 'p'))
    public function select($select = null);

    // Пример -- $qb->delete('User', 'u')
    public function delete($delete = null, $alias = null);

    // Пример -- $qb->update('Group', 'g')
    public function update($update = null, $alias = null);

    // Пример -- $qb->set('u.firstName', $qb->expr()->literal('Arnold'))
    // Пример -- $qb->set('u.numChilds', 'u.numChilds + ?1')
    // Пример -- $qb->set('u.numChilds', $qb->expr()->sum('u.numChilds', '?1'))
    public function set($key, $value);

    // Пример -- $qb->from('Phonenumber', 'p')
    public function from($from, $alias = null);

    // Пример -- $qb->innerJoin('u.Group', 'g', Expr\Join::WITH, $qb->expr()-
>eq('u.status_id', '?1'))
    // Пример -- $qb->innerJoin('u.Group', 'g', 'WITH', 'u.status = ?1')
    public function innerJoin($join, $alias = null, $conditionType = null,
 $condition = null);

    // Пример -- $qb->leftJoin('u.Phonenumbers', 'p', Expr\Join::WITH, $qb->expr()-
>eq('p.area_code', 55))
    // Пример -- $qb->leftJoin('u.Phonenumbers', 'p', 'WITH', 'p.area_code = 55')
    public function leftJoin($join, $alias = null, $conditionType = null,
 $condition = null);

    // Важно: ->where() затирает заданные ранее условия
    //
    // Пример -- $qb->where('u.firstName = ?1', $qb->expr()->eq('u.surname', '?2'))
```

```
    // Пример -- $qb->where($qb->expr()->andX($qb->expr()->eq('u.firstName', '?1'),
 $qb->expr()->eq('u.surname', '?2')))
    // Пример -- $qb->where('u.firstName = ?1 AND u.surname = ?2')
    public function where($where);

    // Пример -- $qb->andWhere($qb->expr()->orX($qb->expr()->lte('u.age', 40),
 'u.numChild = 0'))
    public function andWhere($where);

    // Пример -- $qb->orWhere($qb->expr()->between('u.id', 1, 10));
    public function orWhere($where);

    // Важно: -> groupBy() затирает заданные ранее схемы группировки
    //
    // Пример -- $qb->groupBy('u.id')
    public function groupBy($groupBy);

    // Пример -- $qb->addGroupBy('g.name')
    public function addGroupBy($groupBy);

    // Важно: -> having() также затирает все ранее заданные им условия
    //
    // Пример -- $qb->having('u.salary >= ?1')
    // Пример -- $qb->having($qb->expr()->gte('u.salary', '?1'))
    public function having($having);

    // Пример- $qb->andHaving($qb->expr()->gt($qb->expr()->count('u.numChild'), 0))
    public function andHaving($having);

    // Пример -- $qb->orHaving($qb->expr()->lte('g.managerLevel', '100'))
    public function orHaving($having);

    // Важно: -> orderBy() затирает ранее заданные правила сортировки
    //
    // Пример- $qb->orderBy('u.surname', 'DESC')
    public function orderBy($sort, $order = null);

    // Пример- $qb->addOrderBy('u.firstName')
    public function addOrderBy($sort, $order = null); // По умолчанию $order =
 'ASC'
}
```

Глава 16. Нативный SQL

Итак, Доктрина умеет выполнять самый настоящий SQL, а затем создавать из результата такого запроса объектную модель. Это очень круто. С помощью класса NativeQuery можно выполнять низкоуровневые запросы SELECT, и отображать их результат на необходимые объекты. Описание того, как именно будет произведено отображение представлено классом ResultSetMapping. Она описывает соответствие полей базы данных полям объекта. Что это дает: можно использовать супер-оптимизированные запросы или хранимые процедуры и при этом не терять преимуществ ORM. Одна из красивейших возможностей Doctrine.

> Запросы DELETE, UPDATE or INSERT использовать с помощью NativeSQL API не получится. Для этого подойдет метод **EntityManager#getConnection()**, он вернет ссылку на объект подключение к базе данных, с помощью нее можно вызвать метод executeUpdate().

16.1. Класс NativeQuery

Экземпляр NativeQuery создается с помощью метода **EntityManager#createNativeQuery($sql, $resultSetMapping)**. Он принимает параметра: текста запроса и объект **ResultSetMapping**, который описывает правила отображения результатов.

После создания экземпляра **NativeQuery**, к нему можно привязать параметры, а затем выполнить.

16.2. The ResultSetMapping

Понимание класса ResultSetMapping является ключ к успешной работе с NativeQuery. Результат запроса в Doctrine может содержать следующие компоненты:

- **Entity results**. Корневые элементы в запросе.
- **Joined entity results**. Сущности из связей от вышеприведенных.
- **Field results**. Колонка, представляющая собой поле сущности. Она всегда связана или с **entity result** или **joined entity result**.

- **Scalar results**. Обычные скалярные значения, они будут находится в каждой строке результата. Если результирующий набор содержит сущности, то после добавления к нему скалярных значений он станет смешанным **(mixed)**.
- **Meta results**. Колонки, содержащие мета-информацию, такую как внешние ключи или столбцы дискриминатора. При запросе объектов (getResult()) все мета-столбцы корневых и связанных сущностей должны быть представлены в SQL запросе и соответственно отображены с помощью **ResultSetMapping#addMetaResult**.

> Не будет сюрпризом, что при создании DQL запросов Doctrine внутри себя как раз и использует **ResultSetMapping**. Когда запрос будет разобран синтаксическим анализаторов и преобразован в SQL, Doctrine заполнит ResultSetMapping информацией о том, как запрос должен быть обработан при гидрации.

Ниже будет подробно рассмотрен каждый из описанных выше типов.

16.2.1. Entity results

Entity result описывает тип сущности, которая будет корневым элементом в результате запроса. Entity result можно добавить с помощью **ResultSetMapping#addEntityResult()**. Сигнатура метода выглядит следующим образом:

```
<?php

/*
 * Добавляет entity result к текущему ResultSetMapping.
 *
 * @param string $class Имя класса сущности.
 * @param string $alias Синоним для класса. Должен быть уникальным среди всех
 entity
 *                      results или joined entity results в данном
 ResultSetMapping.
 */
public function addEntityResult($class, $alias)
```

Первый параметр это полное имя класса. Второй — синоним, который должен быть уникальным в рамках ResultSetMapping. Этот синоним используется для прикрепления **field results** к соответствующему **entity result**. Это аналоги идентификационной переменной, используемой в DQL в качестве алиаса для классов или связей.

Но одного entity result недостаточно для формирования корректного ResultSetMapping. Как **entity result** так и **joined entity result** всегда требуют дополнительного набор **field results**, который мы вскоре рассмотрим.

16.2.2. Joined entity results

Этот тип результата описывает связь, в результате запроса она будет связана собственно с элементом **entity result**. Этот типа можно добавить с помощью метода **ResultSetMapping#addJoinedEntityResult()**. Сигнатура методы выглядит так:

```
<?php

/*
 * Добавляет joined entity result.
 *
 * @param string $class Имя класса данной joined entity.
 * @param string $alias Алиас.
 * @param string $parentAlias Алиас для родительской entity result.
 * @param object $relation Имя связи, соединяющей родительский entity result с
 joined entity result.
 */
public function addJoinedEntityResult($class, $alias, $parentAlias, $relation)
```

Первый параметр это просто имя класса связанной сущности. Второй — уникальный синоним, он будет нужен для подключения **field results**. Третий параметр это алиас родительской **entity result** для данной связи. Последним параметром задается имя поля родительского **entity result**, которое и будет являться ссылкой на связь.

16.2.3. Field results

Тут все просто: **field result** описывает какому полю сущности соответствует тот или иной столбец результата SQL запроса. Таким образом, этот тип связан с **entity results**. Добавляется он с помощью метода **ResultSetMapping#addFieldResult()**. Сигнатура метода:

```
<?php

/*
 * Добавляет field result к entity result или joined entity result.
 *
 * @param string $alias Алиас entity result или joined entity result, к которому
 будет добавлено поле.
```

```
 * @param string $columnName Имя колонки в результирующем набора SQL запроса.
 * @param string $fieldName Имя поле сущности.
 */
public function addFieldResult($alias, $columnName, $fieldName)
```

Первый параметр это алиаса для **entity result**, к которому будет привязано данный **field result**. Второй параметр это имя столбца из SQL запроса. Имейте ввиду, что имя столбца зависит от регистра. Последний параметр это имя поля, которому и будет назначено соответствие.

16.2.4. Scalar results

Тип scalar result описывает соответствие колонки из SQL запроса скалярному значению в результате Doctrine. Обычно скалярные результаты используются для хранения результатов агрегатных функций, тем не менее сюда можно назначить абсолютно любой столбец. Добавляется с помощью метода **ResultSetMapping#addScalarResult()**:

```
<?php

/*
 * Добавляет соответствие для scalar result.
 *
 * @param string $columnName Имя колонки в результирующем наборе SQL запроса.
 * @param string $alias Синоним под которым значение этой колонки будет хранится в
 итоговом результирующем наборе.
 */
public function addScalarResult($columnName, $alias)
```

16.2.5. Meta results

Meta result описывает единственный столбец в результирующем наборе SQL запроса, который может быть внешним ключом или столбцом дискриминатора. Эти колонки имеют фундаментальное значение для Doctrine, т.к. с их помощью происходит создание объектов из результатов SQL запроса (гидрация, мать ее так.). Для добавления используется метод**ResultSetMapping#addMetaResult()**, имеющий следующую сигнатуру:

```
<?php

/*
 * Добавляет внешний ключ или столбец дискриминатора.
 *
```

```
     * @param string $alias
     * @param string $columnAlias
     * @param string $columnName
     */
    public function addMetaResult($alias, $columnAlias, $columnName)
```

Первый параметр это алиас **entity result**, которой этот столбец соответствует. Столбец с мета-информацией (внешний ключ или столбец дискриминатора) всегда указывает на entity result. Второй параметр это имя или алиас столбца из SQL запроса. Третий параметр это имя используемой для отображения колонки.

16.2.6. Столбец дискриминатора

При подключении дерева наследования Doctrine нужно дать подсказку по поводу того, какая meta-column в этом дереве является столбцом дискриминатора.

```
<?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)
```

16.2.7. Примеры

Чтобы лучше понять как работает **ResultSetMapping** давайте разберем несколько примеров.

Первый пример описывает отображение для одной сущности.

```
<?php

// Эквивалентный DQL запрос: "select u from User u where u.name=?1"
// User не имеет связей.
```

```
$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();
```

Результат запроса будет выглядеть следующим образом:

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

Обратите внимание, если сущность имеет больше полей, чем представлено в примере выше, то сформированный объект будет являться неполным. Что такое неполные объекты будет описано в 17 главе. Строка, передаваемая методу **createNativeQuery** есть не что иное как нативный SQL запрос, он будет выполнен как есть без каких-либо преобразований со стороны Doctrine.

В предыдущем примере User не имел никаких связей, поэтому таблица была задействована без внешний ключей. В следующем примере предполагается, что User имеет одностороннюю или двустороннюю связь “один к одному” с сущностью CmsAddress, где User является владеющей стороной с внешним ключом.

```
<?php

// Эквивалентный DQL Запрос: "select u from User u where u.name=?1"
// User владеет связью Address, но Address не будет загружен запросом.
$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();
```

Внешние ключи используются Доктриной для ленивой загрузки. П вышеприведенном примере у каждого объекта User в результирующем набор будет свой прокси-объект представляющий Address (ключ address_id). И при запросе этого прокси произойдет фактическая загрузка объекта, представленного этим ключом.

Следовательно, для fetch-joined связи не обязательно иметь внешний ключи в SQL запросе, это нужно только для lazy-loading связей.

```
<?php

// Эквивалентный DQL запрос: "select u from User u join u.address a WHERE u.name
 = ?1"
// User владеет связью Address, которая будет загружена запросом.
$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();
```

В этом примере вложенная сущность Address регистрируется с помощью метода **ResultSetMapping#addJoinedEntityResult**, который уведомляет Doctrine о том, что эта сущность не будет находится на нижнем уровне в наборе, а будет представлена как joined-сущность и располагаться где-то внутри графа объектов. В этом случае третьим параметром мы указываем алиас ‘u’ и “address” четвертым параметром, это означает, что Address будет соответствовать полю User::$address property.

Если связанная сущность является частью иерархии наследования, которой нужен столбец дискриминатора, то этот столбец должен присутствовать в результирующем наборе в виде мета-столбца. Эта ситуация представлена в следующем примере, здесь предполагается, что у User есть один или несколько подклассов и для отображения иерархии используется Class Table Inheritance или Single Table Inheritance (в обоих случаях используется столбец дискриминатора).

```
<?php

// Эквивалентный DQL запрос: "select u from User u where u.name=?1"
// User является базовым классом. У User нет связей.
$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();
```

В случае Class Table Inheritance пример выше породит неполные объекты в случае если любые объекты в результирующем набор являются подтипами User. При использовании DQL, Doctrine автоматически подключает необходимые связи, при использовании нативного SQL это ваша ответственность.

16.3. ResultSetMappingBuilder

У запросов, создаваемый с помощью нативного SQL есть минусы. Основной минус заключается в том, что при изменении имен колонок нужно будет править правила отображения. В DQL это происходит автоматически.

Чтобы избежать этих проблем есть специальный класс **ResultSetMappingBuilder**. Он позволяет добавить все колонки заданной сущности в правила отображения. Чтобы избежать конфликтов можно переименовать необходимые колонки как в примере ниже:

```
<?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";

$rsm = new ResultSetMappingBuilder($em);
$rsm->addRootEntityFromClassMetadata('MyProject\User', 'u');
$rsm->addJoinedEntityFromClassMetadata('MyProject
\Address', 'a', 'u', 'address', array('id' => 'address_id'));
```

Для сущностей с множеством полей использование билдера весьма удобно. Он наследует класс **ResultSetMapping** и имеет ту же функциональностьl. В настоящий момент **ResultSetMappingBuilder** не умеет работать с наследованием классов сущностей.

Глава 17. Отслеживание изменений

Отслеживание изменений это процесс определения что именно было изменено в MANAGED-сущностях с последнего момента как они были синхронизированы с базой данных.

В Doctrine доступны 3 политики отслеживания изменений, каждая со своими достоинствами и недостатками. Политику слежения можно задать как для класса так и для иерархии классов.

17.1. Deferred Implicit

“deferred implicit” — политика, заданная по умолчанию и как водится наиболее подходящая в большинстве случаев. Doctrine обнаруживает изменения, сравнивая каждое поле во время коммита, кроме того она умеет находить изменения в сущностях, на которые ссылаются другие MANAGED-сущности (“persistence by reachability”). Несмотря на удобство, у этой политики есть и минусы, связанные с производительностью при работе с большими Unit of Work. (см. параграф Unit of Work[1]). Это связано с тем, что когда Doctrine не знает что изменилось, он вынуждена перебирать все MANAGED-сущности при каждом вызове **EntityManager#flush()**, что делает эту операции весьма дорогой.

17.2. Deferred Explicit

Политика “deferred explicit” чем-то напоминает ”deferred implicit”, у нее такой же принцип отслеживания изменений: во время коммита проверяется каждое поле. Разница в том, что при вызове **EntityManager#persist(entity)** или каскадном сохранении Doctrine будет просматривать только сущности, которые были явно помечены для этого. Все остальные сущности будут опущены. Если вы готовы пожертвовать автоматизацией в пользу производительности, это политика вам подойдет.

Из всего этого следует, что в рамках этой политики вызов flush() гораздо более дешев. Минус здесь следующий. Если у вас большое приложение и бизнес логика такова, что сущности в процессе их обработки бизнес правилами должны пройти несколько

[1] http://odiszapc.ru/doctrine/architecture/#222_Unit_of_Work

слоев, то до того момента как они будут преданы **EntityManager#persist()** вам придется отслеживать изменения самостоятельно.

Политика настраивается следующим образом:

```
<?php

/*
 * @Entity
 * @ChangeTrackingPolicy("DEFERRED_EXPLICIT")
 */
class User
{
    // ...
}
```

17.3. Notify

Эта политика основана на том, что сущности сами сообщают своим слушателям о происходящих изменениях. Класс, использующий эту политику, должен реализовывать интерфейс **NotifyPropertyChanged *из пространства имен Doctrine (*Doctrine \Common\NotifyPropertyChanged**). Типичная реализация выглядит следующим образом:

```
<?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;
    }
```

```
}
```

Теперь в каждом сеттере для текущего или наследуемого класса нужно уведомлять все экземпляры **PropertyChangedListener**. Чтобы показать как это работает, добавим метод в сущность **MyEntity**:

```
<?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;
        }
    }
}
```

В каждом методе, который изменяет состояние объекта **MyEntity *следует вызывать *_onPropertyChanged**.

Проверка на предмет того, отличается ли новое значение от предшествующего, не обязательно, но рекомендуется. Так или иначе, у вас полный контроль над процессом, хотите проверяйте, хотите — нет.

Минус применения этой политики очевиден: нужно реализовывать интерфейс и писать кучу связующего кода. Но заметьте, мы стараемся оставить логику нотификации максимально абстрактной. Строго говоря, нотификация не затрагивает уровень persistence и компоненты Doctrine ORM или DBAL. Так что эта политика может оказаться

весьма полезной в соответствующих сценариях. Как уже было отмечено, пространство имен Doctrine/Common состоит исключительно из маленьких классов и интерфейсов, у которых практически отсутствуют внешние зависимости (нет зависимостей от DBAL и ORM компонентов). Можно легко переключиться на другой persistence layer, ничего не поломав.

Положительный момент и главное преимущество этой политики в ее эффективности. Среди всех рассмотренных политик она самая быстрая. Отлично подходит для Unit of Work большого размера, а вызов **flush()** при отсутствии изменения очень дешев.

Глава 18. Неполные объекты

С их помощью можно запросить из базы только часть объекта. В общем случае, они являются обрезанными, применяются только для целей оптимизации и работать с ними надо осторожно.

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.

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.

Use of partial objects is tricky. Fields that are not retrieved from the database will not be updated by the UnitOfWork even if they get changed in your objects. You can only promote a partial object to a fully-loaded object by calling `EntityManager#refresh()` or a DQL query with the refresh flag.

18.1. 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");
```

You can also get a partial reference instead of a proxy reference by calling:

```
<?php

$reference = $em->getPartialReference('MyApp\Domain\User', 1);
```

Partial references are objects with only the identifiers set as they are passed to the second argument of the `getPartialReference()` method. All other fields are null.

18.2. 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.

Глава 19. XML Mapping

The XML mapping driver enables you to provide the ORM metadata in form of XML documents.

The XML driver is backed by an XML Schema document that describes the structure of a mapping document. The most recent version of the XML Schema document is available online at 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
                  https://raw.github.com/doctrine/doctrine2/master/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. In order to work, this requires certain conventions:

- Each entity/mapped superclass must get its own dedicated XML mapping document.
- The name of the mapping document must consist of the fully qualified name of the class, where namespace separators are replaced by dots (.). 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. This is more of a convention and you are not forced to do this. You can change the file extension easily enough.

```
<?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);
```

Warning

Note that Doctrine ORM does not modify any settings for `libxml`, therefore, external XML entities may or may not be enabled or configured correctly. XML mappings are not XXE/XEE attack vectors since they are not related with user input, but it is recommended that you do not use external XML entities in your mapping files to avoid running into unexpected behaviour.

19.1. Simplified XML Driver

The Symfony project sponsored a driver that simplifies usage of the XML Driver. The changes between the original driver are:

1. File Extension is .orm.xml
2. Filenames are shortened, "MyProjectEntitiesUser" will become User.orm.xml
3. You can add a global file and add multiple entities in this file.

Configuration of this client works a little bit different:

```
<?php
$namespaces = array(
    '/path/to/files1' => 'MyProject\Entities',
    '/path/to/files2' => 'OtherProject\Entities'
);
$driver = new \Doctrine\ORM\Mapping\Driver\SimplifiedXmlDriver($namespaces);
$driver->setGlobalBasename('global'); // global.orm.xml
```

19.1.1. Example

As a quick start, here is a small example document that makes use of several common elements:

```
// 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://raw.github.com/doctrine/doctrine2/master/doctrine-
mapping.xsd">
        <entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">
        <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>

        <generator strategy="AUTO"/>
            <sequence-generator sequence-name="tablename_seq" allocation-
size="100" initial-value="1" />


  <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="CASCADE"/>
        </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="false" />
                </join-columns>

                    <join-column name="group_id" referenced-column-
name="id" column-definition="INT NULL" />

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

Be aware that class-names specified in the XML files should be fully qualified.

19.1.2. 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.

19.2. 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="MyProject
\User" table="cms_users" schema="schema_name" repository-class="MyProject
\UserRepository">
        <!-- definition here -->
    </entity>
</doctrine-mapping>
```

Required attributes:

- name - The fully qualified class-name of the entity.

Optional attributes:

- **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** - The type of inheritance, defaults to 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.
- **schema** - (>= 2.5) The schema the table lies in, for platforms that support schemas

19.3. Defining Fields

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 `` element.

```
<entity name="MyProject\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" />
    <field name="login_count" type="integer" nullable="false">
        <options>
            <option name="comment">The number of times the user has logged in.</
option>
            <option name="default">0</option>
        </options>
    </field>
</entity>
```

Required attributes:

- name - The name of the Property/Field on the given Entity PHP class.

Optional attributes:

- 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? Defaults to false.
- nullable - Should this field allow NULL as a value? Defaults to false.
- 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.
- options - Array of additional options:
  - default - The default value to set for the column if no value is supplied.
  - unsigned - Boolean value to determine if the column should be capable of representing only non-negative integers (applies only for integer column and might not be supported by all vendors).
  - fixed - Boolean value to determine if the specified length of a string column should be fixed or varying (applies only for string/binary column and might not be supported by all vendors).
  - comment - The comment of the column in the schema (might not be supported by all vendors).
  - customSchemaOptions - Array of additional schema options which are mostly vendor specific.
- 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.

> For more detailed information on each attribute, please refer to the DBAL `Schema-Representation` documentation.

19.4. 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="MyProject\User">
```

```
</entity>
```

Required attributes:

- name - The name of the Property/Field on the given Entity PHP class.
- type - The `Doctrine\DBAL\Types\Type` name, preferably "string" or "integer".

Optional attributes:

- 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="MyProject\User">

        <generator strategy="AUTO" />

</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="MyProject\User">

        <generator strategy="SEQUENCE" />
```

```
        <sequence-generator sequence-name="user_seq" allocation-size="5" initial-
value="1" />

</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.

```
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.
```

19.5. 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="MyProject\BaseClass">
        <field name="created" type="datetime" />
        <field name="updated" type="datetime" />
    </mapped-superclass>
</doctrine-mapping>
```

Required attributes:

- 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.

19.6. 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="MyProject\Animal" inheritance-type="JOINED">
    <discriminator-column name="discr" type="string" />
    <discriminator-map>
        <discriminator-mapping value="cat" class="MyProject\Cat" />
        <discriminator-mapping value="dog" class="MyProject\Dog" />
        <discriminator-mapping value="mouse" class="MyProject\Mouse" />
    </discriminator-map>
</entity>
```

The allowed values for inheritance-type attribute are `JOINED` or `SINGLE_TABLE`.

> All inheritance related definitions have to be defined on the root entity of the hierarchy.

19.7. 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>
```

19.8. Defining One-To-One Relations

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="MyProject\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. *IMPORTANT:* No leading backslash!
- 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="MyProject\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. *IMPORTANT:* No leading backslash!

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. Defaults to 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="MyProject\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>
```

19.9. 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="MyProject\Article">
    <many-to-one field="author" target-entity="User" />
</entity>
```

Required attributes:

- 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. *IMPORTANT:* No leading backslash!

Optional attributes:

- 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. Defaults to false.
- 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="MyProject\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.

19.10. 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="MyProject\User">
    <one-to-many field="phonenumbers" target-entity="Phonenumber" mapped-by="user"
 />
</entity>
```

Required attributes:

- 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. *IMPORTANT:* No leading backslash!
- mapped-by - Name of the field on the owning side (here Phonenumber entity) that contains the owning side association.

Optional attributes:

- fetch - Either LAZY, EXTRA_LAZY or EAGER, defaults to LAZY.
- index-by: Index the collection by a field on the target entity.

19.11. 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="MyProject\User">
    <many-to-many field="groups" target-entity="Group" />
</entity>
```

Required attributes:

- 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. *IMPORTANT:* No leading backslash!

Optional attributes:

- 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="MyProject\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>

                <join-column name="group_id" referenced-column-name="id"/>

        </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.

19.12. 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="MyProject\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 />`

19.13. 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.

Required attributes:

- name - The column name of the foreign key.
- referenced-column-name - The column name of the associated entities primary key

Optional attributes:

- 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".

19.14. 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="MyProject\User">
    <many-to-many field="groups" target-entity="Group">
        <order-by>
            <order-by-field name="name" direction="ASC" />
        </order-by>
    </many-to-many>
</entity>
```

19.15. 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">
    <unique-constraints>
        <unique-constraint columns="name,user_email" name="search_idx" />
    </unique-constraints>
</entity>
```

You have to specify the column and not the entity-class field names in the index and unique-constraint definitions.

19.16. 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 `` 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://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
    <entity name="Application\Model\ArticleAttribute">
        <field name="value" type="string" />
        <many-to-one field="article" target-entity="Article" inversed-
by="attributes" />
     </entity>
</doctrine-mapping>
```

Глава 20. YAML Mapping

The YAML mapping driver enables you to provide the ORM metadata in form of YAML documents.

The YAML mapping document of a class is loaded on-demand the first time it is requested and subsequently stored in the metadata cache. In order to work, this requires certain conventions:

- Each entity/mapped superclass must get its own dedicated YAML mapping document.
- The name of the mapping document must consist of the fully qualified name of the class, where namespace separators are replaced by dots (.).
- All mapping documents should get the extension ".dcm.yml" to identify it as a Doctrine mapping file. This is more of a convention and you are not forced to do this. You can change the file extension easily enough.

```
<?php
$driver->setFileExtension('.yml');
```

It is recommended to put all YAML mapping documents in a single folder but you can spread the documents over several folders if you want to. In order to tell the YamlDriver where to look for your mapping documents, supply an array of paths as the first argument of the constructor, like this:

```
<?php
use Doctrine\ORM\Mapping\Driver\YamlDriver;

// $config instanceof Doctrine\ORM\Configuration
$driver = new YamlDriver(array('/path/to/files'));
$config->setMetadataDriverImpl($driver);
```

20.1. Simplified YAML Driver

The Symfony project sponsored a driver that simplifies usage of the YAML Driver. The changes between the original driver are:

- File Extension is .orm.yml
- Filenames are shortened, "MyProject\Entities\User" will become User.orm.yml

- You can add a global file and add multiple entities in this file.

Configuration of this client works a little bit different:

```
<?php
$namespaces = array(
    '/path/to/files1' => 'MyProject\Entities',
    '/path/to/files2' => 'OtherProject\Entities'
);
$driver = new \Doctrine\ORM\Mapping\Driver\SimplifiedYamlDriver($namespaces);
$driver->setGlobalBasename('global'); // global.orm.yml
```

20.2. Example

As a quick start, here is a small example document that makes use of several common elements:

```
# Doctrine.Tests.ORM.Mapping.User.dcm.yml
Doctrine\Tests\ORM\Mapping\User:
    type: entity
    repositoryClass: Doctrine\Tests\ORM\Mapping\UserRepository
    table: cms_users
    schema: schema_name # The schema the table lies in, for platforms that support
 schemas (Optional, >= 2.5)
    readOnly: true
    indexes:
        name_index:
            columns: [ name ]
    id:
        id:
            type: integer
            generator:
                strategy: AUTO
    fields:
        name:
            type: string
            length: 50
        email:
            type: string
            length: 32
            column: user_email
            unique: true
            options:
                fixed: true
                comment: User's email address
```

```
        loginCount:
            type: integer
            column: login_count
            nullable: false
            options:
                unsigned: true
                default: 0
    oneToOne:
        address:
            targetEntity: Address
            joinColumn:
                name: address_id
                referencedColumnName: id
                onDelete: CASCADE
    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 ]
```

Be aware that class-names specified in the YAML files should be fully qualified.

20.3. Reference

20.3.1. Unique Constraints

It is possible to define unique constraints by the following declaration:

```
# ECommerceProduct.orm.yml
ECommerceProduct:
    type: entity
```

```
    fields:
        # definition of some fields
    uniqueConstraints:
        search_idx:
            columns: [ name, email ]
```

Глава 21. Annotations Reference

You've probably used docblock annotations in some form already, most likely to provide documentation metadata for a tool like `PHPDocumentor` (@author, @link, …). Docblock annotations are a tool to embed metadata inside the documentation section which can then be processed by some tool. Doctrine 2 generalizes the concept of docblock annotations so that they can be used for any kind of metadata and so that it is easy to define new docblock annotations. In order to allow more involved annotation values and to reduce the chances of clashes with other docblock annotations, the Doctrine 2 docblock annotations feature an alternative syntax that is heavily inspired by the Annotation syntax introduced in Java 5.

The implementation of these enhanced docblock annotations is located in the `Doctrine\Common\Annotations` namespace and therefore part of the Common package. Doctrine 2 docblock annotations support namespaces and nested annotations among other things. The Doctrine 2 ORM defines its own set of docblock annotations for supplying object-relational mapping metadata.

> If you're not comfortable with the concept of docblock annotations, don't worry, as mentioned earlier Doctrine 2 provides XML and YAML alternatives and you could easily implement your own favourite mechanism for defining ORM metadata.

In this chapter a reference of every Doctrine 2 Annotation is given with short explanations on their context and usage.

21.1. @Column

Marks an annotated instance variable as "persistent". It has to be inside the instance variables PHP DocBlock comment. Any value hold inside this variable will be saved to and loaded from the database as part of the lifecycle of the instance variables entity-class.

Required attributes:

- **type**: Name of the Doctrine Type which is converted between PHP and Database representation.

Optional attributes:

- **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), which is the maximum number of digits that are stored for the values.
- **scale**: The scale for a decimal (exact numeric) column (applies only for decimal column), which represents the number of digits to the right of the decimal point and must not be greater than *precision*.
- **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.
- **options**: Array of additional options:
  - `default`: The default value to set for the column if no value is supplied.
  - `unsigned`: Boolean value to determine if the column should be capable of representing only non-negative integers (applies only for integer column and might not be supported by all vendors).
  - `fixed`: Boolean value to determine if the specified length of a string column should be fixed or varying (applies only for string/binary column and might not be supported by all vendors).
  - `comment`: The comment of the column in the schema (might not be supported by all vendors).
  - `collation`: The collation of the column (only supported by Drizzle, Mysql, PostgreSQL>=9.1, Sqlite and SQLServer).
- **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 atid73[:ref:`@JoinColumn <annref_joincolumn>`].

  For more detailed information on each attribute, please refer to the DBAL `Schema-Representation` documentation.

Examples:

```
<?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;

/**
 * @Column(type="string", length=2, options={"fixed":true, "comment":"Initial letters of
 first and last name"})
 */
protected $initials;

/**
 * @Column(type="integer", name="login_count" nullable=false, options={"unsigned":true,
 "default":0})
 */
protected $loginCount;
```

21.2. @ColumnResult

References name of a column in the SELECT clause of a SQL query. Scalar result types can be included in the query result by specifying this annotation in the metadata.

Required attributes:

- **name**: The name of a column in the SELECT clause of a SQL query

21.3. @Cache

Add caching strategy to a root entity or a collection.

Optional attributes:

- **usage**: One of `READ_ONLY`, `READ_WRITE` or `NONSTRICT_READ_WRITE`, By default this is `READ_ONLY`.
- **region**: An specific region name

21.4. @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 id75[:doc:`details on all the available change tracking policies <change-tracking-policies>`] can be found in the configuration section.

Example:

```
<?php
/**
 * @Entity
 * @ChangeTrackingPolicy("DEFERRED_IMPLICIT")
 * @ChangeTrackingPolicy("DEFERRED_EXPLICIT")
 * @ChangeTrackingPolicy("NOTIFY")
 */
class User {}
```

21.5. @DiscriminatorColumn

This annotation is an optional 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.

If this annotation is not specified, the discriminator column defaults to a string column of length 255 called `dtype`.

Required attributes:

- **name**: The column name of the discriminator. This name is also used during Array hydration as key to specify the class-name.

Optional attributes:

- **type**: By default this is string.
- **length**: By default this is 255.

21.6. @DiscriminatorMap

The discriminator map is a required annotation on the topmost/super class in an inheritance hierarchy. Its only argument is an array 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 on whether the classes are in the namespace or not.

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

21.7. @Entity

Required annotation to mark a PHP class as an entity. Doctrine manages the persistence of all classes marked as entities.

Optional attributes:

- **repositoryClass**: Specifies the FQCN of a subclass of the EntityRepository. 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.

Example:

```
<?php
/**
 * @Entity(repositoryClass="MyProject\UserRepository")
 */
```

```
class User
{
    //...
}
```

21.8. @EntityResult

References an entity in the SELECT clause of a SQL query. If this annotation is used, the SQL statement should select all of the columns that are mapped to the entity object. This should include foreign key columns to related entities. The results obtained when insufficient data is available are undefined.

Required attributes:

- **entityClass**: The class of the result.

Optional attributes:

- **fields**: Array of @FieldResult, Maps the columns specified in the SELECT list of the query to the properties or fields of the entity class.
- **discriminatorColumn**: Specifies the column name of the column in the SELECT list that is used to determine the type of the entity instance.

21.9. @FieldResult

Is used to map the columns specified in the SELECT list of the query to the properties or fields of the entity class.

Required attributes:

- **name**: Name of the persistent field or property of the class.

Optional attributes:

- **column**: Name of the column in the SELECT clause.

21.10. @GeneratedValue

Specifies which strategy is used for identifier generation for an instance variable which is annotated by id77[:ref:`@Id <annref_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.

Optional attributes:

- **strategy**: Set the name of the identifier generation strategy. Valid values are AUTO, SEQUENCE, TABLE, IDENTITY, UUID, CUSTOM and NONE. If not specified, default value is AUTO.

Example:

```
<?php
/**
 * @Id
 * @Column(type="integer")
 * @GeneratedValue(strategy="IDENTITY")
 */
protected $id = null;
```

21.11. @HasLifecycleCallbacks

Annotation which has to be set on the entity-class PHP DocBlock to notify Doctrine that this entity has entity lifecycle 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.

Example:

```
<?php
/**
 * @Entity
 * @HasLifecycleCallbacks
 */
class User
{
    /**
     * @PostPersist
     */
    public function sendOptinMail() {}
}
```

21.12. @Index

Annotation is used inside the id80[:ref:`@Table <annref_table>`] annotation on the entity-class level. It provides a hint to the SchemaTool to generate a database index on the specified table columns. It only has meaning in the SchemaTool schema generation context.

Required attributes:

- **name**: Name of the Index
- **columns**: Array of columns.

Optional attributes:

- **options**: Array of platform specific options:
- `where`: SQL WHERE condition to be used for partial indexes. It will only have effect on supported platforms.

Basic example:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",indexes={@Index(name="search_idx", columns={"name",
 "email"})})
 */
class ECommerceProduct
{
}
```

Example with partial indexes:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",indexes={@Index(name="search_idx", columns={"name",
 "email"}, options={"where": "(((id IS NOT NULL) AND (name IS NULL)) AND (email IS
 NULL))"})})
 */
class ECommerceProduct
{
}
```

21.13. @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.

Example:

```
<?php
/**
 * @Id
 * @Column(type="integer")
 */
protected $id = null;
```

21.14. @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 id82[:ref:`@DiscriminatorMap <annref_discriminatormap>`] andid84[:ref:`@DiscriminatorColumn <annref_discriminatorcolumn>`] annotations.

Examples:

```
<?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
{
    // ...
}
```

21.15. @JoinColumn

This annotation is used in the context of relations in id86[:ref:`@ManyToOne <annref_manytoone>`], id88[:ref:`@OneToOne <annref_onetoone>`] fields and in the Context of id90[:ref:`@JoinTable <annref_jointable>`] nested inside a @ManyToMany. This annotation is not required. If it is not specified the attributes *name* and *referencedColumnName* are inferred from the table and primary key names.

Required attributes:

- **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.

Optional attributes:

- **unique**: Determines whether this relation is exclusive between the affected entities and should be enforced as such on the database constraint level. Defaults to false.
- **nullable**: Determine whether 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 enables the 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 id92[:ref:`@Column <annref_column>`] also sets the related @JoinColumn's columnDefinition. This is necessary to make foreign keys work.

Example:

```
<?php
/**
 * @OneToOne(targetEntity="Customer")
 * @JoinColumn(name="customer_id", referencedColumnName="id")
 */
private $customer;
```

21.16. @JoinColumns

An array of @JoinColumn annotations for a id94[:ref:`@ManyToOne <annref_manytoone>`] or id96[:ref:`@OneToOne <annref_onetoone>`] relation with an entity that has multiple identifiers.

21.17. @JoinTable

Using id98[:ref:`@OneToMany <annref_onetomany>`] or id100[:ref:`@ManyToMany <annref_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.

Optional attributes:

- **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.

Example:

```
<?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;
```

21.18. @ManyToOne

Defines that the annotated instance variable holds a reference that describes a many-to-one relationship between two entities.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **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.

Example:

```
<?php
/**
 * @ManyToOne(targetEntity="Cart", cascade={"all"}, fetch="EAGER")
 */
private $cart;
```

21.19. @ManyToMany

Defines that the annotated instance variable holds a many-to-many relationship between two entities. id102[:ref:`@JoinTable <annref_jointable>`] is an additional, optional annotation that has reasonable default configuration values using the table and names of the two related entities.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **mappedBy**: This option specifies the property name on the targetEntity that is the owning side of this relation. It is a required attribute for the inverse side of a relationship.
- **inversedBy**: The inversedBy attribute designates the field 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.

> 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).

Example:

```
<?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;
```

21.20. @MappedSuperclass

A mapped superclass is an abstract or concrete class that provides persistent entity state and mapping information for its subclasses, but which is not itself an entity. 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 forid104[:doc:`more details on the restrictions of mapped superclasses `].

Optional attributes:

- **repositoryClass**: (>= 2.2) Specifies the FQCN of a subclass of the EntityRepository. That will be inherited for all subclasses of that Mapped Superclass.

Example:

```
<?php
/**
 * @MappedSuperclass
 */
class MappedSuperclassBase
{
    // ... fields and methods
}

/**
 * @Entity
 */
class EntitySubClassFoo extends MappedSuperclassBase
{
    // ... fields and methods
}
```

21.21. @NamedNativeQuery

Is used to specify a native SQL named query. The NamedNativeQuery annotation can be applied to an entity or mapped superclass.

Required attributes:

- **name**: The name used to refer to the query with the EntityManager methods that create query objects.
- **query**: The SQL query string.

Optional attributes:

- **resultClass**: The class of the result.

- **resultSetMapping**: The name of a SqlResultSetMapping, as defined in metadata. Example:

```
<?php
/**
 * @NamedNativeQueries({
 *      @NamedNativeQuery(
 *          name            = "fetchJoinedAddress",
 *          resultSetMapping= "mappingJoinedAddress",
 *          query           = "SELECT u.id, u.name, u.status, a.id AS a_id, a.country,
 a.zip, a.city FROM cms_users u INNER JOIN cms_addresses a ON u.id = a.user_id WHERE
 u.username = ?"
 *      ),
 * })
 * @SqlResultSetMappings({
 *      @SqlResultSetMapping(
 *          name    = "mappingJoinedAddress",
 *          entities= {
 *              @EntityResult(
 *                  entityClass = "__CLASS__",
 *                  fields      = {
 *                      @FieldResult(name = "id"),
 *                      @FieldResult(name = "name"),
 *                      @FieldResult(name = "status"),
 *                      @FieldResult(name = "address.zip"),
 *                      @FieldResult(name = "address.city"),
 *                      @FieldResult(name = "address.country"),
 *                      @FieldResult(name = "address.id", column = "a_id"),
 *                  }
 *              )
 *          }
 *      )
 * })
 */
class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;

    /** @Column(type="string", length=50, nullable=true) */
    public $status;

    /** @Column(type="string", length=255, unique=true) */
    public $username;
```

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

    /** @OneToOne(targetEntity="Address") */
    public $address;

    // ....
}
```

21.22. @OneToOne

The @OneToOne annotation works almost exactly as the id106[:ref:`@ManyToOne <annref_manytoone>`] with one additional option which can be specified. The configuration defaults for id108[:ref:`@JoinColumn <annref_joincolumn>`] using the target entity table and primary key column names apply here too.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **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. Defaults to false.
- **inversedBy**: The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.

Example:

```
<?php
/**
 * @OneToOne(targetEntity="Customer")
 * @JoinColumn(name="customer_id", referencedColumnName="id")
 */
private $customer;
```

21.23. @OneToMany

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **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. Defaults to 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.

Example:

```
<?php
/**
 * @OneToMany(targetEntity="Phonenumber", mappedBy="user", cascade={"persist", "remove",
 "merge"}, orphanRemoval=true)
 */
public $phonenumbers;
```

21.24. @OrderBy

Optional annotation that can be specified with a id110[:ref:`@ManyToMany <annref_manytomany>`] or id112[:ref:`@OneToMany <annref_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:

Example:

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

The DQL Snippet in OrderBy is only allowed to consist of unqualified, unquoted field names and of an optional ASC/DESC positional statement. Multiple Fields are separated by a comma (,). The referenced field names have to exist on the `targetEntity` class of the `@ManyToMany` or `@OneToMany` annotation.

21.25. @PostLoad

Marks a method on the entity to be called as a @PostLoad event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

21.26. @PostPersist

Marks a method on the entity to be called as a @PostPersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

21.27. @PostRemove

Marks a method on the entity to be called as a @PostRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

21.28. @PostUpdate

Marks a method on the entity to be called as a @PostUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

21.29. @PrePersist

Marks a method on the entity to be called as a @PrePersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

21.30. @PreRemove

Marks a method on the entity to be called as a @PreRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

21.31. @PreUpdate

Marks a method on the entity to be called as a @PreUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

21.32. @SequenceGenerator

For 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.

Required attributes:

- **sequenceName**: Name of the sequence

Optional attributes:

- **allocationSize**: Increment the sequence by the allocation size when its fetched. A value larger than 1 allows optimization for scenarios where you create more than one new entity per request. Defaults to 10
- **initialValue**: Where the sequence starts, defaults to 1.

Example:

```
<?php
/**
 * @Id
 * @GeneratedValue(strategy="SEQUENCE")
 * @Column(type="integer")
 * @SequenceGenerator(sequenceName="tablename_seq", initialValue=1, allocationSize=100)
 */
protected $id = null;
```

21.33. @SqlResultSetMapping

The SqlResultSetMapping annotation is used to specify the mapping of the result of a native SQL query. The SqlResultSetMapping annotation can be applied to an entity or mapped superclass.

Required attributes:

- **name**: The name given to the result set mapping, and used to refer to it in the methods of the Query API.

Optional attributes:

- **entities**: Array of @EntityResult, Specifies the result set mapping to entities.
- **columns**: Array of @ColumnResult, Specifies the result set mapping to scalar values.

Example:

```
<?php
/**
 * @NamedNativeQueries({
 *      @NamedNativeQuery(
 *          name            = "fetchUserPhonenumberCount",
 *          resultSetMapping= "mappingUserPhonenumberCount",
 *          query           = "SELECT id, name, status, COUNT(phonenumber) AS numphones FROM
 cms_users INNER JOIN cms_phonenumbers ON id = user_id WHERE username IN (?) GROUP BY id,
 name, status, username ORDER BY username"
 *      ),
 *      @NamedNativeQuery(
 *          name            = "fetchMultipleJoinsEntityResults",
 *          resultSetMapping= "mappingMultipleJoinsEntityResults",
 *          query           = "SELECT u.id AS u_id, u.name AS u_name, u.status AS u_status,
 a.id AS a_id, a.zip AS a_zip, a.country AS a_country, COUNT(p.phonenumber) AS numphones
 FROM cms_users u INNER JOIN cms_addresses a ON u.id = a.user_id INNER JOIN cms_phonenumbers
 p ON u.id = p.user_id GROUP BY u.id, u.name, u.status, u.username, a.id, a.zip, a.country
 ORDER BY u.username"
 *      ),
 * })
 * @SqlResultSetMappings({
 *      @SqlResultSetMapping(
 *          name    = "mappingUserPhonenumberCount",
 *          entities= {
 *              @EntityResult(
 *                  entityClass = "User",
 *                  fields      = {
 *                      @FieldResult(name = "id"),
 *                      @FieldResult(name = "name"),
 *                      @FieldResult(name = "status"),
 *                  }
 *              )
 *          },
 *          columns = {
 *              @ColumnResult("numphones")
 *          }
 *      ),
 *      @SqlResultSetMapping(
 *          name    = "mappingMultipleJoinsEntityResults",
 *          entities= {
 *              @EntityResult(
 *                  entityClass = "__CLASS__",
 *                  fields      = {
 *                      @FieldResult(name = "id",       column="u_id"),
```

```
 *                     @FieldResult(name = "name",     column="u_name"),
 *                     @FieldResult(name = "status",   column="u_status"),
 *                 }
 *             ),
 *             @EntityResult(
 *                 entityClass = "Address",
 *                 fields      = {
 *                     @FieldResult(name = "id",       column="a_id"),
 *                     @FieldResult(name = "zip",      column="a_zip"),
 *                     @FieldResult(name = "country",  column="a_country"),
 *                 }
 *             )
 *         },
 *         columns = {
 *             @ColumnResult("numphones")
 *         }
 *     )
 *})
 */
 class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;

    /** @Column(type="string", length=50, nullable=true) */
    public $status;

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

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

    /** @OneToMany(targetEntity="Phonenumber") */
    public $phonenumbers;

    /** @OneToOne(targetEntity="Address") */
    public $address;

    // ....
}
```

21.34. @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 entity's unqualified classname.

Required attributes:

- **name**: Name of the table

Optional attributes:

- **indexes**: Array of @Index annotations
- **uniqueConstraints**: Array of @UniqueConstraint annotations.
- **schema**: (>= 2.5) Name of the schema the table lies in.

Example:

```
<?php
/**
 * @Entity
 * @Table(name="user",
 *      uniqueConstraints={@UniqueConstraint(name="user_unique",columns={"username"})},
 *      indexes={@Index(name="user_idx", columns={"email"})}
 *      schema="schema_name"
 * )
 */
class User { }
```

21.35. @UniqueConstraint

Annotation is used inside the id114[:ref:`@Table <annref_table>`] annotation on the entity-class level. 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.

Required attributes:

- **name**: Name of the Index
- **columns**: Array of columns.

Optional attributes:

- **options**: Array of platform specific options:
- `where`: SQL WHERE condition to be used for partial indexes. It will only have effect on supported platforms.

Basic example:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx",
 columns={"name", "email"})})
 */
class ECommerceProduct
{
}
```

Example with partial indexes:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx",
 columns={"name", "email"}, options={"where": "(((id IS NOT NULL) AND (name IS NULL)) AND
 (email IS NULL))"})})
 */
class ECommerceProduct
{
}
```

21.36. @Version

Marker annotation that defines a specified column as version attribute used in an id116[:ref:`optimistic locking <transactions-and-concurrency_optimistic-locking>`] scenario. It only works on id118[:ref:`@Column <annref_column>`] annotations that have the type `integer` or `datetime`. Combining `@Version` with id120[:ref:`@Id <annref_id>`] is not supported.

Example:

```
<?php
/**
```

```
     * @Column(type="integer")
     * @Version
     */
    protected $version;
```