# От переводчика

Если вы заметите какие либо не точности в переводе или знаете как можно улучшить перевод, то пишите мне на email - krasun.net@gmail.com. Также если заметите, что руководство было обновлено, сообщите мне об этом.

Только вместе мы сможем сделать перевод лучше.

# Приступая к работе. XML-версия.

Doctrine 2 - это проект, цель которого управление хранением предметной области. Сердцем проекта является шаблон проектирования Data Mapper, который полностью отделяет бизнес-логику или предметную область от хранения ее системах управления реляционными базами данных. Программисты получают выгоду от использования Doctrine 2, так как они могут полностью сосредоточиться на решении проблем бизнес-логики, откладываю задачу хранения доменных объектов на второй план. Но это не значит, что хранение не является важным аспектом Doctrine, мы просто считаем, что отделение задачи хранения от взаимодействия доменных объектов между собой, дает значительные преимущества.

## Что такое сущности?

Сущности - это легковесные PHP объекты, которые не нуждаются в наследовании какого-либо абстрактного класса или интерфейсов. Класс сущности не должен быть final или содержать final методы. В дополнение ко всему, эти классы также не нуждаются в реализации методов ни __clone, ни __wakeup или же это нужно делать это осторожно.

Сущность содержит хранимые свойства. Хранимое свойство - это переменная сущности, которая хранит данные сохраняемые и извлекаемые с помощью возможностей Doctrine.

## Пример модели: Bug Tracker

В этому руководстве мы реализуем доменную модель: Bug Tracker из документации по

Zend_Db_Table (http://framework.zend.com/manual/en/zend.db.table.html). Прочитав данную документацию, мы можем извлечь из нее требования к доменной модели:

- У ошибок есть свойства: описание, дата создания, статус, репортер и инженер.
- Ошибка может принадлежать разным продуктам.
- Репортер ошибок и инженер - это пользователи системы.
- Пользователь может создать новую ошибку.
- Инженер отвечающий за ошибку может ее закрыть.
- Ошибки можно просматривать в постраничном режиме.

**Внимание! Данное руководство будете постепенно увеличивать ваши знания о Doctrine 2, для того, что бы показать распространенные подводные камни отображения сущностей на базу данных. Не копируйте код - он не подходит для производства без понимания некоторых аспектов, которым учит данное руководство.**

## Первый прототип

Первый упрощенный дизайн доменной модели будет выглядеть как следующий набор классов:

```
<?php

class Bug
{
    public $id;
    public $description;
    public $created;
    public $status;
    public $products = array();
    public $reporter;
    public $engineer;
}
class Product
{
    public $id;
    public $name;
}
class User
{
    public $id;
    public $name;
    public $reportedBugs = array();
    public $assignedBugs = array();
}
```

**Внимание! Это всего лишь прототип, пожалуйста, не используйте публичные свойства с Doctrine 2 на полную, в разделе "Запросы используемые в приложении" вы узнаете почему. Комбинация заместителей и публичных свойств может привести к красивым и трудноуловимым ошибкам.**

Так как мы сосредоточимся на аспекте преобразования данных из БД в объекты, то в данный момент, не нужно прикладывать усилия для инкапсуляции бизнес-логики. Все хранимые свойства - доступны публично. Позже мы увидим, что это не лучшее решение при использования Doctrine 2, так как она одна уже заставит вас скрывать бизнес-логику. Для сохранения Doctrine 2 применяет рефлексию, которая позволяет "достучаться" до значений ваших свойств сущностей.

Многие поля - это обычные скалярные значения, например, 3 ID поля сущностей, ихние имена, описания, статусы и даты изменений. Doctrine 2 может легко обрабатывать эти поля, как и любая другая ORM. С точки зрения нашей доменный модели, мы уже можем использовать данные сущности, а как они преобразуется и сохраняются в БД мы увидим чуть позже.

В нашей доменной модели существует несколько связей, семантика которых будет обсуждаться ниже в каждом конкретном случае, для того, что бы объяснить как Doctrine 2 их обрабатывает. В общем случае связь один-ко-одному или многие-ко-одному в БД заменяются образцами родственных объектов в доменной модели. Связь один-ко-многим или многие-ко-многим заменяются образцами коллекций в доменной модели.

Если вы поняли о чем идет речь, то вы заметите, что Doctrine 2 загрузит всю БД в память, когда вы обратитесь к одному объекту. Однако по умолчанию Doctrine 2 генерирует заместители, которые лениво подгружают сущности или коллекции, которые еще не были получены из БД.

Для того, что бы использовать ленивую загрузку коллекции, необходимо заменить обычные PHP массивы общим интерфейсом коллекций Doctrine\Common\Collections\Collection, которые работают как массивы, используя следующие интерфейсы: ArrayAccess, IteratorAggregate, Countable. Класс Doctrine\Common\Collections\ArrayCollection - это самая простая реализация данного интерфейса.

Теперь когда, мы знаем, что это, мы можем прояснить нашу доменную модель используя предположения о связанных коллекциях:

```
<?php
```

```
use Doctrine\Common\Collections\ArrayCollection;

class Bug
{
    public $products = null;

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

class User
{
    public $reportedBugs = null;
    public $assignedBugs = null;

    public function __construct()
    {
        $this->reportedBugs = new ArrayCollection();
        $this->assignedBugs = new ArrayCollection();
    }
}
```

Всякий раз когда сущность создаеться из БД, реализация коллекции типа Doctrine\ORM\PersistantCollection замещает массив. По сравнению с Doctrine\Common\Collections\ArrayCollection данная реализация помогает Doctrine ORM понять какие изменения происходят в коллекции, так как это очень важно для правильного сохранения состояния.

**Внимание! Лениво подгружающие заместители хранят объект Doctrine - EntityManager и все его зависимости. Поэтому var_dump(), скорее всего, будет пытаться вывести дамп очень большой рекурсивной структуры, которую не возможно вывести и прочитать. Вы должны использовать Doctrine\Common\Util\Debug::dump() для того, что бы ограничит вывод дампа до читабельного, человеком, уровня. Кроме того, вы должны знать вывод дампа Entity в браузер, может занять несколько минут и метод Debug::dump() просто игнорирует любые вхождения EntityManager в заместителях.**

В следствии того, что мы работаем с коллекциями для ссылок, мы должны быть предельно осторожны, когда реализуем двух-сторонние связи в доменной модели. Идея о владеющий стороне или обратной стороне является центральной идеей и должна быть постонно на уме. Следующие предположения о связях должны учитываться для того, что бы Doctrine 2 корректно работала. Эти предположения не уникальны для Doctrine 2, но являются

лучшими практиками при обработке связей в БД и ORM (Object-Relational Mapping).

- Изменения в коллекциях сохраняются или обновляются, когда владеющая сторона коллекции обновляется или сохраняется.
- Сохранение Сущности на обратной стороне отношения никогда не запускает операцию сохранения изменений в коллекции.
- В связи один-к-одному сущность содержащая внешний ключ связанной сущности в собственной таблице, всегда является владеющей стороной.
- В связи многие-ко-многим любая сторона может быть владеющей. Однако в двунаправленных связях только одна сторона может быть владеющей.
- В связи многие-ко-одному по умолчания сторона “многих” владеющая так как она содержит внешний ключ.
- Связь один-ко-многим инвертирована по умолчанию, пока внешний ключ содержит сторона “многих”. В связи “один-ко-многим” сторона “одного” может быть владеющей только в том случае, если связь реализована как “многие-ко-многим” с использованием соединения таблиц и сторона “одного” содержит только уникальные значения.

**Важно! Поддержание двух-сторонних связей в актуальном состоянии на обратной стороне лежит на ответственности пользовательского кода. Doctrine не может магически обновить твои коллекции для поддержания актуального состояния.**

Применительно к сущностям User и сущностям Bug у нас имеются двусторонние ссылки. Мы должны обновить код для обеспечения консистентности двусторонних связей:

```
<?php

class Bug
{
    protected $engineer;
    protected $reporter;

    public function setEngineer($engineer)
    {
        $engineer->assignedToBug($this);
        $this->engineer = $engineer;
    }

    public function setReporter($reporter)
    {
        $reporter->addReportedBug($this);
        $this->reporter = $reporter;
    }

    public function getEngineer()
```

```
    {
        return $this->engineer;
    }

    public function getReporter()
    {
        return $this->reporter;
    }
}
class User
{
    public function addReportedBug($bug)
    {
        $this->reportedBugs[] = $bug;
    }

    public function assignedToBug($bug)
    {
        $this->assignedBugs[] = $bug;
    }
}
```

Я решил назвать “обратные” методы в прошедшем времени, что бы было наглядно видно, что связывание уже произошло и данные методы служат лишь для поддержания консистентности. ссылок. Вы видите, что из исходя из User::addReportedBug() и User::assignedToBug() использование этих методов по одиночке в пользовательском коде не добавит сущность Bug к коллекции владеющей стороны Bug::$reporter или Bug::$engineer. Использование этих методов и вызов Doctrine для сохранения изменений не будет обновлять отображение коллекции в базе данных.

Только использование Bug::setEngineer() или Bug::setReporter() позволит корректно сохранить целостность связей. Мы также оставили коллекции защищенными (protected) начиная с версии PHP 5.3 для Doctrine не составит труда работать с приватными или защищенными свойствами используя рефлексию (Reflection).

Свойства Bug::$reporter и Bug::$engineer являются связями многие-ко-многим, которые указывают на сущность User. В нормализованной реляционной модели внешний ключ сохраняется в таблице сущностей Bug, следовательно в нашей объектно-реляционной модели Bug владеющая сторона. Вы всегда должны быть уверены, что сценарии использовании вашей доменной модели руководствуются тем какая сторона владеющая, а какая обратная при Doctrine маппинге (Doctrine mapping). В нашем примере, где бы ни сохранялась новая сущность Bug или сущность Engineer связывалась бы с сущностью Bug, мы не хотим обновлять сущность User для сохранения ссылки. Все дело в сущности Bug - владеющей стороне связи.

Сущности Bug ссылаются на сущности Product в одном направлении связи многие-ко-многим в базе данных - это указывается тем, что сущности Bug принадлежат сущностям Product.

```
<?php

class Bug
{
    protected $products = null; // Set protected for encapsulation

    public function assignToProduct($product)
    {
        $this->products[] = $product;
    }

    public function getProducts()
    {
        return $this->products;
    }
}
```

Теперь мы покончили с доменной моделью заданной по требованиям. Начиная с доменной модели с публичными свойствами мы проделали немного работы, что бы инкапсулировать связи между объектами, что бы быть уверенными, что мы не нарушаем состояние консистентности, когда используем Doctrine.

Однако до сих пор, предположения введенные Doctrine о наших бизнес объектах не ограничивали нас в возможностях моделирования домена. На самом деле мы будем инкапсулировать доступ ко всем свойствам используя лучшие практики ООП.

## Отображение метаданных для наших Сущностей

До текущего момента, мы реализовали наши сущности в виде структур данных и Doctrine ничего не знает о том как их хранить. Если бы существовали базы данных, которые хранят все в оперативной памяти, то мы могли бы уже закончить проект, так как выполнили все технические требования. Однако мир не совершенен и нам необходимо сохранять наши сущности в каком-то хранилище, что бы мы были уверены, что не потеряем ихнее состояние. На данный момент Doctrine работает с РСУБД. Но в будущем мы собираемся так же поддерживать NoSQL базы данных, такие как CouchDb или  MongoDb.

Следующий шаг для хранения с Dotrine - это описание структуры нашей доменной модели при помощи некоторого языка метаданных. Язык метаданных описывает, то как наши сущности, их атрибуты, связи между ними будут хранится в БД, а также ограничения

применительно к ним.

Метаданные для сущностей загружаются при помощи реализации Doctrine\ORM\Mapping\Driver\Driver и Dotrine 2 поставляется с драйверами для работы с XML, YAML и нотациями. В этом руководстве я буду использовать XML драйвер. Я думаю что XML бьет YAML благодаря проверки схемы на корректность и моя любимая IDE предлагает мне авто-завершение для XML файлов и это нечто ибо вам теперь не нужно постоянно возвращаться к документации.

Так как мы не описывали наши сущности в пределах какого-либо пространства имен мы должны реализовать файлы Bug.dcm.xml, Product.dcm.xml и User.dcm.xml и положить их каталог, где будут хранится настройки отображения.

Первое обсуждаемое определение будет для Product, пока это самое легкое описание:

```
<doctrine-mapping
      xmlns = "http://doctrine-project.org/schemas/orm/doctrine-
mapping"
      xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation =
                "http://doctrine-project.org/schemas/orm/doctrine-
mapping                    http://doctrine-project.org/schemas/orm/doctrine-
mapping.xsd">

      <entity name="Product" table="zf_products">
                <id name="id" type="integer" column="product_id">
                    <generator strategy="AUTO" />
            </id>
              <field name="name" column="product_name" type="string" />
</entity>

</doctrine-mapping>
```

Тег самого высокого уровня entity о классе и имени таблицы. Примитивный тип Product::$name определен как атрибут. Свойство Id определено в Id теге. Id имеет тег generator, вложенный внутрь, который определяет, что генерация первичного ключа автоматически возлагается на родные механизмы БД, например, AUTO INCREMENT, если это MySQL и Sequences если это PostgreSQL или Oracle.

Продолжим определение отображений. Переходим к определению отображения сущности Bug:

```
<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/doctri   ne-mapping

http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

      <entity name="Bug" table="zf_bugs">
        <id name="id" type="integer" column="bug_id">
            <generator strategy="AUTO" />
        </id>

        <field name="description" column="bug_description"
           type="text" />
        <field name="created" column="bug_created"
           type="datetime" />
        <field name="status" column="bug_status"
           type="string" />

        <many-to-one target-entity="User" field="reporter"
             inversed-by="reportedBugs">
            <join-column name="reporter_id"
                 referenced-column-name="account_id" />
        </many-to-one>

        <many-to-one target-entity="User" field="engineer"
             inversed-by="assignedBugs">
            <join-column name="engineer_id"
                 referenced-column-name="account_id" />
        </many-to-one>

        <many-to-many target-entity="Product" field="products">
            <join-table name="zf_bugs_products">
                <join-columns>
                    <join-column name="bug_id"
                 referenced-column-name="bug_id" />
            </join-columns>
                <inverse-join-columns>
                   <join-column name="product_id"
                referenced-column-name="product_id" />
                </inverse-join-columns>
            </join-table>
        </many-to-many>

      </entity>

      </doctrine-mapping>
```

Теперь опять мы имеем определение сущности, идентификатор и примитивного типа. Имя столбцов используется из Zend_Db_Table примеров и отличается от имен атрибутов в классе Bug. Дополнительно для поля created мы указываем, что оно принадлежит к типу

DateTime, которые преобразует формат YYYY-MM-DD из БД в объекты PHP DateTime и обратно.

После описания полей следуют две условные ссылки на сущность User. Они создаются при помощи many-to-one тега. Имя класса связанной сущности было определено target-entity атрибутом и этого достаточно для того, что бы преобразователь БД мог получить доступ к внешней таблице. Теги join-column используется для указания как называются внешние и связанные столбцы - информация которая необходима для Doctrine, что бы правильно построить связи между этими двумя сущностями. Так как репортер и инженер являются владельцами в двусторонней связи, то мы должны также определить inversed-by атрибут. Они должны указывать на поля обратной стороны связи.

Последнее упущенное свойство - коллекция Bug::$products. Оно содержит все продукты, где конкретный баг проявлялся. Снова вы должен определить атрибуты target-entity и field в теге many-to-many. Кроме того, вы должны определить детали связующей таблицы многие-ко-многим и ее внешние столбцы. Определение не много сложнее, но я доверился авто-завершению, так как я все время забываю детали схемы.

Осталось описать сущность User:

```
<doctrine-mapping
      xmlns =
        "http://doctrine-project.org/schemas/orm/doctrine-mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
          "http://doctrine-project.org/schemas/orm/doctrine-mapping
          http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

        <entity name="User" table="zf_accounts">
            <id name="id" type="integer" column="account_id">
                <generator strategy="AUTO" />
            </id>

            <field name="name" column="account_name" type="string" />

            <one-to-many target-entity="Bug" field="reportedBugs"
            mapped-by="reporter" />
            <one-to-many target-entity="Bug" field="assignedBugs"
            mapped-by="engineer" />

        </entity>

  </doctrine-mapping>
```

Есть несколько вещей которые необходимо упомянуть о one-to-many тегах. Помните,

что мы обсуждали обратные и владеющие стороны. Поэтому мы можем указать только на свойство в классе Bug, которое содержит владеющие стороны.

Пример хорошо показывает базовые возможности языка описания метаданных.

## Получение EntityManager'a

Публичный интерфейс Doctrine EntityManager предоставляет точку доступа к полному циклу управления вашими сущностями, а также трансформации сущностей для сохранения и обратно. Вы должны настроить EntityManager и создать его для того, что бы использовать ваши сущности с помощью Doctrine 2. Я покажу настройку по шагам и каждый из них опишу:

```
<?php

// Setup Autoloader (1)
require '/path/to/lib/Doctrine/Common/ClassLoader.php';
$loader = new Doctrine\Common\ClassLoader(
      "Doctrine", '/path/to/Doctrine/trunk/lib/');
$loader->register();

$config = new Doctrine\ORM\Configuration(); // (2)

// Proxy Configuration (3)
$config->setProxyDir(__DIR__.'/lib/MyProject/Proxies');
$config->setProxyNamespace('MyProject\Proxies');
$config->setAutoGenerateProxyClasses(
      (APPLICATION_ENV == "development"));

// Mapping Configuration (4)
$driverImpl = new Doctrine\ORM\Mapping\Driver\XmlDriver(
      __DIR__."/config/mappings");
$config->setMetadataDriverImpl($driverImpl);

// Caching Configuration (5)
if (APPLICATION_ENV == "development") {
    $cache = new \Doctrine\Common\Cache\ArrayCache();
} else {
    $cache = new \Doctrine\Common\Cache\ApcCache();
}
$config->setMetadataCacheImpl($cache);
$config->setQueryCacheImpl($cache);

// database configuration parameters (6)
$conn = array(
```

```
    'driver' => 'pdo_sqlite',
    'path' => __DIR__ . '/db.sqlite',
);

// obtaining the entity manager (7)
$evm = new Doctrine\Common\EventManager()
$entityManager = \Doctrine\ORM\EntityManager::create(
      $conn, $config, $evm);
```

Первый блок настраивает возможности автозагрузки Doctrine. Я регистрирую пространства имен Doctrine по заданным путям. Для того, что бы добавить свои собственные пространства имен, вы можете создать новый экземпляр класса ClassLoader с разными пространствами имен и разными путями. Необязательно использовать Doctrine ClassLoader для ваших нужд автозагрузки, вы можете использовать, то что вы считаете лучшим.

Следующий блок содержит создание объекта конфигурации ORM. Прежде чем конфигурация будет показана в следующих блоках, здесь есть еще несколько моментов, которые подробно описаны в разделе конфигурации в руководстве (прим. переводчика: информация, которая приводится по ссылке на английском языке).

Конфигурация заместителя необходимый блок вашего приложения, вы должны указать куда Doctrine будет писать код для генерации заместителей. Заместители это потомки ваших сущностей созданные при помощи Doctrine для “типо-безопасной” ленивой загрузки. В следующих главах мы увидим точно как это работает. Помимо пути к заместителям мы также указываем пространство имен в котором они будут находится, а также флаг autoGenerateProxyClasses указывающий должны ли заместители каждый раз создаваться заново при каждом запросе, что рекомендуются при разработке. В рабочей версии следуют предотвратить это, так как эта операция стоит достаточно дорого.

Четвертый блок кода содержит детали описания драйвера отображения. Мы используем XML отображение, следовательно мы настраиваем экземпляр XmlDriver указывая путь к каталогу с настройками отображения, где мы положили Bug.dcm.xml, Product.dcm.xml и User.dcm.xml.

В 5-ом блоке настраивается параметры кэширования. При разработке мы используем ArrayCache для одно запроса, а в рабочей версии буквально требуется использовать APC, Memcache или XCache, что бы получить полную скорость использования Doctrine. Внутри Doctrine на полную использует кэширование для метаданных и DQL языка запросов так, что удостоверьтесь, что вы используете механизм кэширования.

6-ой блок показывает какие параметры конфигурации необходимы для соединения с БД, в моем случае это основанная на файлах sqlite база данных. Все конфигурационные

параметры всех поставляемых драйверов вы можете найти в секции руководства - настройка DBAL (прим. переводчика: информация, которая приводится по ссылке на английском языке).

В последнем блоке показано как мы получаем EntityManager при помощи фабричного метода. Здесь мы также передаем экземпляр EventManager, который является не обязательным. Однако используя EventManager вы можете внедрятся в жизненный цикл ваших сущностей, что очень часто требуется, так что теперь вы знаете как это сделать.

## Генерация схемы базы данных

Теперь когда мы определили метаданные отображения и загрузили EntityManager, мы хотим создать схему для реляционной базы данных из него (EntityManager). Doctrine имеет интерфейс командной строки, который предоставляет вам доступ к SchemaTool - компонент позволяющий генерировать необходимые таблицы для работы с метаданными.

Для работы с инструментами командной строки - cli-config.php должен быть расположен в корне проекта, в котором будут запускаться команды Doctrine. Это очень простой файл:

```
<?php
      $helperSet = new \Symfony\Components\Console\Helper\HelperSet(
            array('em' =>
            new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper(
                  $entityManager)
      ));
      $cli->setHelperSet($helperSet);
```

Потом вы можете изменить каталог, на каталог проекта и использовать инструментарий командной строки Doctrine:

```
doctrine@my-desktop> cd myproject/
doctrine@my-desktop> doctrine orm:schema-tool:create
```

**Команда doctrine будет существовать только в случае, если вы устанавливали doctrine из PEAR. В противном случае вы должны копнуть в коде bin/doctrine.php для настройки вашего клиент командной строки Doctrine. Просмотрите раздел инструментария в руководстве (прим. переводчика: информация, которая приводится по ссылке на английском языке) для того, что бы настроить правильно Doctrine консоль.**

На протяжении разработки вам, вероятно, необходимо будет пересоздать базу данных несколько раз, когда вы изменяете метаданные сущностей. Вы можете это сделать следующим

разом:

```
doctrine@my-desktop> doctrine orm:schema-tool:drop
doctrine@my-desktop> doctrine orm:schema-tool:create
```

Или используйте функциональность обновления:

```
doctrine@my-desktop> doctrine orm:schema-tool:update
```

Обновление базы данных использует алгоритм разности для заданной схемы БД - краеугольный камень пакета Doctrine\DBAL, который может быть использован без пакета Doctrine ORM. Однако это недоступно в SQLite, поскольку он не поддерживает ALTER TABLE.

## Запись сущностей в базу данных

После того, как мы создали схему, мы можем начать сохранять наши сущности в базе дынных. Для новичков мы должны создать пользовательские сценарии:

```
<?php

$newUsername = "beberlei";

$user = new User();
$user->name = $newUsername;

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

Также мы можем создать Product:

```
<?php

$newProductName = "My Product";

$product = new Product();
$product->name = $newProductName;

$entityManager->persist($product);
$entityManager->flush();
```

Так, что же происходит в этих двух фрагментах? В обоих фрагментах, работа с классами происходит как обычно, что здесь интересно, так это общение с EntityManager. Для сообщения EntityManager того, что новая сущность должна быть записана в базу данных, вы должны вызвать метод persist(). Однако, на самом деле EntityManager ничего не сделает, это

просто уведомление. Вы должны явно вызвать метод flush(), что бы изменения были записаны в базу данных.

Вам наверное, интересно, почему существуют различия между уведомлением о сохранении persist() и сохранении изменений flus()? Doctrine 2 использует шаблон проектирования UnitOfWork для того, что бы агрегировать все операции (INSERT, UPDATE, DELETE) в одну быструю транзакцию, когда вызывается метод flush(). Использование такого подхода символически быстрее, чем запись сущностей по отдельности. В более сложных сценариев, чем два предложенных выше вы свободны запрашивать обновления для разных сущностей и сохранять изменения за раз.

Doctrine UnitOfWork определяет сущности, которые изменились после считывания их с базы данных, так что вам остается следить за сущностями, которые вы создаете и удаляете и передавать их в EntityManager#persist() и EntityManager#remove() соответственно. Данное сравнение изменения в сущностях использует эффективный алгоритм и не требует дополнительной памяти, так что вы сохраните вычислительную мощность, так как будут обновляться только изменившиеся столбцы.

```
<?php

$reporter = $entityManager->find("User", $theReporterId);
$engineer = $entityManager->find("User", $theDefaultEngineerId);

$bug = new Bug();
$bug->description = "Something does not work!";
$bug->created = new DateTime("now");
$bug->status = "NEW";

foreach ($productIds AS $productId) {
    $product = $entityManager->find("Product", $productId);
    $bug->assignToProduct($product);
}

$bug->setReporter($reporter);
$bug->setEngineer($engineer);

$entityManager->persist($bug);
$entityManager->flush();

echo "Your new Bug Id: ".$bug->id."\n";
```

Это первый контакт с API чтения EntityManager, который показывает, что EntityManager#find($name, $id) возвращает один экземпляр сущности запрошенной по первичному ключу. Кроме этого мы снова видим использование persist() + flush() шаблона для сохранения ошибки в базу данных.

Смотрите как легко связывается ошибка, репортер, инженер и программное обеспечение сделанные при использовании обсуждавшихся методов в разделе “Первый прототип”. UnitOfWork обнаружит связи, когда flush() будет вызван и надлежащим образом сохранит их в базе данных.

## Примеры запросов используемых в приложении

### Список ошибок

Используя предыдущие примеры мы можем немного заполнить базу данных, однако сейчас мы должны обсудить, как получить необходимые виды представления используя основные отражатели. В запущенном приложении ошибки, могут просматриваться в постраничном виде и это будет первый сценарий чтения:

```
<?php

$dql = "SELECT b, e, r FROM Bug b
        JOIN b.engineer e
        JOIN b.reporter r
        ORDER BY b.created DESC";

$query = $entityManager->createQuery($dql);
$query->setMaxResults(30);
$bugs = $query->getResult();

foreach($bugs AS $bug) {
    echo $bug->description." - ".
         $bug->created->format('d.m.Y')."\n";
    echo "    Reported by: ".
         $bug->getReporter()->name."\n";
    echo "    Assigned to: ".$bug->getEngineer()->name."\n";
    foreach($bug->getProducts() AS $product) {
        echo "    Platform: ".$product->name."\n";
    }
    echo "\n";
}
```

Данный DQL запрос считывает 30 только, что добавленных ошибок с их соответствующими инженерами и репортерами за один обычный SQL запрос. Вывод в консоль будет следующим:

```
Something does not work! - 02.04.2010
    Reported by: beberlei
```

```
Assigned to: beberlei
Platform: My Product
```

**DQL - это не SQL. Вы наверное удивлены почему мы начали писать SQL в начале этого сценария. Разве мы не используем ORM, что бы избавиться от написания SQL вручную? Doctrine представляет DQL, который лучше описывается как язык запроса объектов с диалектом OQL и  похожим на HQL или JPQL. Этот язык запросов не знает ничего о концепции столбцов и таблиц, но знает о классах сущностей и ихних свойствах. Используя метаданные, которые мы до этого определили DQL позволяет писать очень характерные и мощные запросы.**

**Главная причина использования DQL как предпочтительного API чтения большинства ORM - это его легкость по сравнению с SQL. DQL позволяет создавать такие конструкции, которые большинство ORM не могут, GROUP BY даже с HAVING, под-запросы, соединения нескольких классов, смешанные результаты с сущности и скалярными типами данных такие как результаты COUNT() и подобных. Используя DQL у вас очень редко будет возникать желание забросить ORM потому, что он не поддерживает самые мощные концепции SQL**

**Кроме написания DQL вручную, вы можете использовать QueryBuilder полученный из EntityManager#createQueryBuilder(), который является объектом запроса поверх яызка DQL.**

**В крайнем случае вы можете использовать SQL и описания результирующей выборки для получения сущностей из базы данных. Сам DQL сводиться к обычному SQL выражению и к экземпляру ResutSetMapping. Используя SQL вы также можете использовать хранимые процедуры для получения данных или использовать продвинутые не переносимые запросы к база данных, например, такие как в PostgreSQL рекурсивные запросы.**

### Преобразование списка ошибок в массив

В предыдущем сценарии мы получили результирующий набор как соответствующие экземпляры объектов. Однако, мы не ограничены получением только объектов от Doctrine. Для обычного представления списка нам достаточно доступа на чтение наших сущностей и по этому мы можем переключиться на получение обычных PHP массивов вместо объектов. Это позволит получить на выходе более производительности для простых запросов чтения.

Для реализация представления списка в виде обычного PHP массива, мы можем переписать наш код следующим образом:

```
<?php

$dql = "SELECT b, e, r, p FROM Bug b JOIN b.engineer e ".
       "JOIN b.reporter r JOIN b.products p ORDER BY b.created DESC";
$query = $em->createQuery($dql);
$bugs = $query->getArrayResult();

foreach ($bugs AS $bug) {

echo $bug['description'] . " - " . $bug['created']-
>format('d.m.Y')."\n";
    echo "    Reported by: ".$bug['reporter']['name']."\n";
    echo "    Assigned to: ".$bug['engineer']['name']."\n";
    foreach($bug['products'] AS $product) {
        echo "    Platform: ".$product['name']."\n";
    }
    echo "\n";
}
```

Однако есть одно значительное отличие в DQL запросе, мы должны добавить дополнительную связь с продуктами присоединенными к ошибке. Результирующий SQL запрос будет намного больше, но с точки зрения производительности - это более выгодные решение.

**Поиск по первичному ключу**

Следующий сценарий показывает получение ошибки по первичному ключу. Это может быть сделано при помощи использования DQL как в предыдущем примере, однако есть один удобные метод в EntityManager, которые позволяет обрабатывать загрузку по первичному ключу, которую мы видели в сценариях записи:

```
<?php

$bug = $entityManager->find("Bug", (int)$theBugId);
```

Однако далее мы увидим другую проблему при использовании такого подхода. Попытаемся вывести на экран имя инженера:

```
<?php

echo "Bug: ".$bug->description."\n";
echo "Engineer: ".$bug->getEngineer()->name."\n";
```

Оно будет равным null! Что произошло? Данный подход работал в предыдущем

примере, значит проблем с сохранением в Doctrine - быть не может. Так почему же это произошло? Вы попали в ловушку публичных свойств.

Когда мы получили ошибку по первичному ключу, инженер и репортер не загружались мгновенно из базы данных, но были заменены лениво загружающими заместителями. Фрагмент кода заместителя может быть найден в указанном каталоге заместителей и выглядит следующим образом:

```
<?php

namespace MyProject\Proxies;

/**
 * THIS CLASS WAS GENERATED BY THE DOCTRINE ORM. DO NOT EDIT THIS FILE.
 */
class UserProxy extends \User implements \Doctrine\ORM\Proxy\Proxy
{
    // .. lazy load code here

    public function addReportedBug($bug)
    {
        $this->_load();
        return parent::addReportedBug($bug);
    }

    public function assignedToBug($bug)
    {
        $this->_load();
        return parent::assignedToBug($bug);
    }
}
```

Вы видите как при каждом вызове метода заметистетили лениво загружаються из базы данных? Однако использование публичных свойств не позволяет Dotrine улавливать доступ к этим свойств и производить ленивую загрузку. Мы должны переписать наши сущности, сделав свойства приватными или защищенными и добавить геттеры и сеттеры, что бы получить работающий пример:

```
<?php

echo "Bug: ".$bug->getDescription()."\n";
echo "Engineer: ".$bug->getEngineer()->getName()."\n";

/**
Bug: Something does not work!
Engineer: beberlei
*/
```

Обязательное описание свойств как приватных или защищенных в Doctrine 2, также заставляет вас инкапсулировать ваши объекты, что является хорошей практикой объектно-ориентированного программирования.

## Просмотр сущностей User

В следующем сценарии мы хотим получить всех пользователей, которые были связаны с ошибками или сами сообщали об ошибках. Мы достигнем этого результата при использовании DQL снова, в этот раз с использованием некоторых WHERE условий и граничных параметров.

```
<?php

$dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter
r ".
       "WHERE b.status = 'OPEN' AND e.id = ?1 OR r.id = ?1
        ORDER BY b.created DESC";

$myBugs = $entityManager->createQuery($dql)
                        ->setParameter(1, $theUserId)
                        ->setMaxResults(15)
                        ->getResult();

foreach ($myBugs AS $bug) {
    echo $bug->getDescription()."\n";
}
```

Для сценариев чтения этого примера, мы будем опускать из виду то, что иженеры могут закрывать ошибки.

## Количество багов

До сих пор мы рассматривали только получение сущностей или их представления в вида массива.  Doctrine также поддерживает получение не только сущностей при помощи DQL. Эти значения называются “результирующие скалярные значения” и они должны быть значениями, которые получаются с использованием COUNT, SUM, MIN, MAX и AVG функциями.

```
<?php

$dql = "SELECT p.id, p.name, count(b.id) AS openBugs FROM Bug b ".
       "JOIN b.products p WHERE b.status = 'OPEN' GROUP BY p.id";
$productBugs = $em->createQuery($dql)->getScalarResult();
```

```
foreach($productBugs as $productBug) {
    echo $productBug['name']." has " .
         $productBug['openBugs'] . " open bugs!\n";
}
```

## Обновление сущностей

Просто сценарий опущенный из требований - инженеры могут закрывать ошибки. Это выглядит следующим образом:

```
<?php

$bug = $entityManager->find("Bug", (int)$theBugId);
$bug->close();

$entityManager->flush();
```

Когда мы получаем ошибку из базы данных она сохраняется в IndentityMap внутри UnitOfWork. Это обозначает, что будет существовать только одна ошибка с указанным id и не важно, сколько раз вы вызовете EntityManage#find(). UnitOfWork также обнаруживает сущности которые получены при помощи DQL и также существуют в IdentityMap.

Когда вызывается сохранение всех изменений flush() EntityManager обходит IdentityMap и выполняет сравнение между оригинальными значениями полученными из базы данных и значениями, которые сущности имеют на данный момент. Если хоть одно из свойств сущности изменилось, сущность будет обновлена в базе данных. Только измененные свойства будут сохранены в базе данных - это очень производительное решение по сравнению с обновлением всех свойств сущности.

На этом все, надеюсь вам понравилось. Руководство будет обновляться постепенно включая следующие темы:

- Хранилища сущностей
- Больше о ассоциативных отображениях
- События жизненного цикла возникающие в UnitOfWork
- Сортировка коллекций

Дополнительные детали всех тем, которые здесь обсуждались могут быть найдены в соответствующих разделах документации.