Занятие №6
ORM Doctrine

Обо мне

- **Плутахин Павел**
  - старший разработчик, тимлид команды Symfony
  - Web - отдел
  - В SimbirSoft с июня 2012 года
- **Образование**
  - Высшее, УлГТУ, радио-факультет, 2007 год
- **Технологии**
  - PHP, JavaScript, HTML5, CSS3
  - Symfony, Slim,ORM Doctrine, ORM Propel, Twig, Smarty, Phing, Composer, ElasticSearch
- **Участие в проектах**
  - ФГИС Росаккредитация
  - Портал госуслуг московской области
  - Российская социальная сеть
  - Социально-новостной портал

Где можно хранить данные?

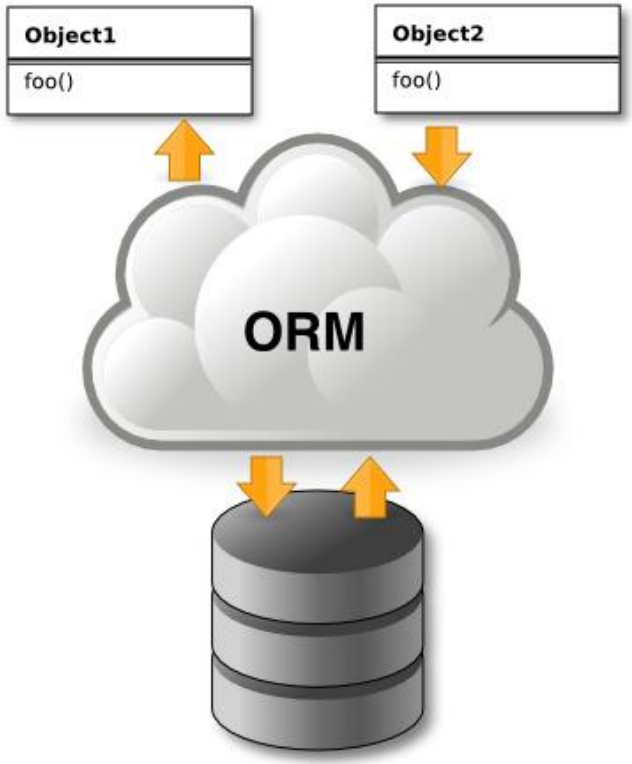
Где хранить данные?

- В файле
- В сессии
- На клиенте (куки)
- В памяти
- В базе данных
  - реляционная БД
  - NoSQL

Что такое ORM?

Object Relational Mapping

ORM - технология программирования, которая преобразует объекты структур в памяти приложения в форму, удобную для сохранения в реляционных и не реляционных базах данных, а также для решения обратной задачи — развертывания реляционной модели в объектную, с сохранением свойств объектов и отношений между ними.



Почему ORM?

Пример работы с БД на чистом PHP:

```php
$host = "localhost";
$user = "user";
$password = "password";

// Производим попытку подключения к серверу MySQL:
if (!mysql_connect($host, $user, $password)) {
    echo "MySQL Error!";
    exit;
}

// Выбираем базу данных:
mysql_select_db($db);

// SQL-запрос:
$q = mysql_query ("SELECT * FROM User");

// Выводим таблицу:
for ($c=0; $c<mysql_num_rows($q); $c++) {
    $f = mysql_fetch_array($q);
    echo $f['name'];
    echo $f['email'];
}
```

Почему ORM?

Пример работы с БД на ORM Doctrine:

```
$em = $this->getDoctrine()->getManager();
$users = $em->getRepository('MainBundle:User')->findAll();

foreach ($users as $user) {
    echo $user->getName();
    echo $user->getEmail();
}
```

Что такое ORM Doctrine?

ORM Doctrine занимается преобразованием записей из БД в классы PHP с необходимым набором свойств и наоборот.

Doctrine состоит из 3 основных библиотек:

1. **Common**
2. **DBAL** (включает в себя Common)

   слой абстракции доступа к БД
3. **ORM** (включает в себя как DBAL, так и Common)

   инструменты объектно-реляционного отображения

Common

- Загрузчик классов

Основан на широко используемых соглашениях о пространствах имен, имен классов и структуре каталогов

- Аннотации

Аннотации в стиле DocBlock к классам PHP

- Кеширование

Кеширование «из коробки» доступны: ApcCache, ArrayCache, FilesystemCache, MemcachedCache, PhpFileCache, RedisCache, WinCacheCache, XcacheCache, ZendDataCache

- События

DBAL

Database Abstraction Layer

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

- Конструктор запросов (**DQL**)
- Управление транзакциями
- Менеджер схемы БД
- Менеджер событий
- Обеспечение безопасности
- Поддержка различных БД
- Кеширование

ORM

Основные функции:

- Хранимые классы

Преобразования обьектов PHP в записи БД и наоборот

- Метаданные

Используется аннотации, XML, YAML или PHP.

- Связи, коллекции, наследование
- Репозиторий
- Менеджер обьектов (*EntityManager*)
- Язык DQL

Console

- Создание/обновление/валидация схем, таблиц в БД
- Миграции
- Интерактивный генератор классов моделей
- Генератор CRUD
- Фикстуры
- Выполнение запросов SQL/DQL
- Очистка кеша метаданных/запросов/результатов

Язык DQL

Doctrine Query Language

Собственный объектно-ориентированном язык запросов

- С помощью DQL можно строить довольно мощные запросы к существующим объектным моделям. По сути все объекты 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.
- Doctrine для работы с хранилищем предлагает Doctrine_Query API для создания запросов.

Entity

Сущность — легковесный хранимый в БД объект из предметной области вашего приложения.

Сущностью может являться любой PHP объект.

Entity

Пример сущности:

```
class Message
{
    private $id;
    private $text;
    private $postedAt;
}
```

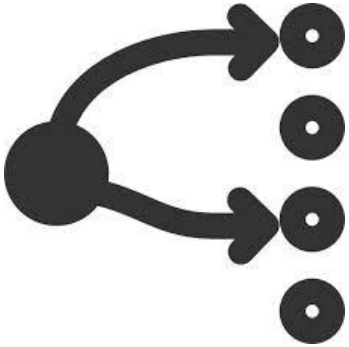
- Простой класс PHP
- Не наследуется от других специальных базовых классов
- Сущность не должна быть финальным (final) классом или иметь финальные методы.
- Все хранимые свойства класса сущности должны быть либо закрытыми (private), либо защищенными (protected), для работы "ленивой загрузки"
- Сущности поддерживают наследование, полиморфизм для своих связей и запросов. Сущностями могут быть как абстрактные, так и обычные классы.

EntityManager

Менеджер сущностей

UnitOfWork

Единица работы



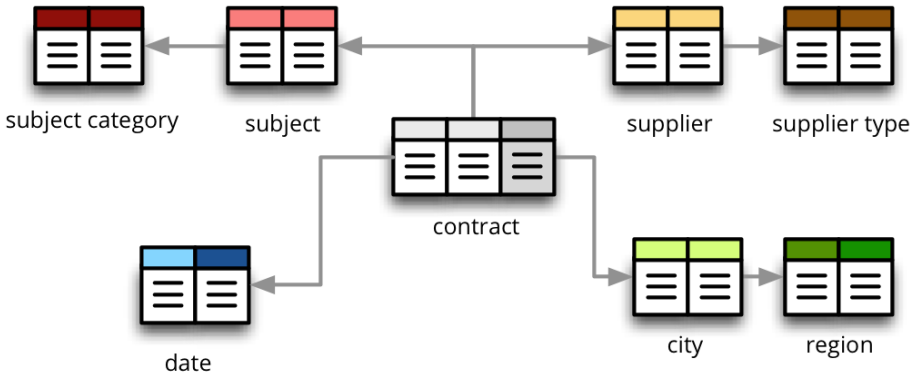
Repository

Хранилище сущностей

Mapping

Виды описания метаданных

- Docblock Annotations
- XML
- YAML
- PHP code



Mapping

Annotations

```php
/** @Entity */
class Message
{
    /** @Column(type="integer") */
    private $id;
    /** @Column(length=140) */
    private $text;
    /** @Column(type="datetime", name="posted_at") */
    private $postedAt;
}
```

XML

```xml
<doctrine-mapping>
  <entity name="Message">
    <field name="id" type="integer" />
    <field name="text" length="140" />
    <field name="postedAt" column="posted_at"
type="datetime" />
  </entity>
</doctrine-mapping>
```

YAML

```yaml
Message:
    type: entity
    fields:
        id:
            type: integer
        text:
            length: 140
        postedAt:
            type: datetime
            column: posted_at
```

PHP

```php
$metadata->mapField(array(
   'id' => true,
   'fieldName' => 'id',
   'type' => 'integer'
));

$metadata->mapField(array(
   'fieldName' => 'text',
   'type' => 'string',
));
 ...
```

Настройка Doctrine в Symfony

Перед тем как начать работу, необходимо настроить соединение с базой данных:

```
#app/config/parameters.yml
parameters:
    database_driver:   pdo_mysql
    database_host:     localhost
    database_name:   test_project
    database_user:     root
    database_password: password
```

Теперь, когда Doctrine знает о том куда подключаться, вы указываете ей создать БД:

```
php app/console doctrine:database:create
```

Создание сущности

Используем консольный генератор сущности:

php app/console doctrine:generate:entity

Welcome to the Doctrine2 entity generator

The Entity shortcut name:

MainBundle:Product

Configuration format (yml, xml, php, or annotation) [annotation]:

annotation

New field name (press <return> to stop adding fields):

name

Field type [string]:

string

Field length [255]:

255

Создание сущности

```php
// src/Bundle/MainBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;

        …
```

Обновление схемы БД

На текущий момент в нашей базе еще нет таблицы для хранения нашей сущности Product.

К счастью, Doctrine может автоматически создать все таблицы базы данных, необходимые для всех известных сущностей приложения:

php app/console doctrine:schema:update --force

Эта команда необычайно мощная. Она сравнивает как *должна* выглядеть база данных (основываясь на информации об отображении для сущностей) с тем, как она выглядит *на самом деле*, и создаёт SQL выражения, необходимые для *обновления* базы данных до того вида, какой она должна быть.

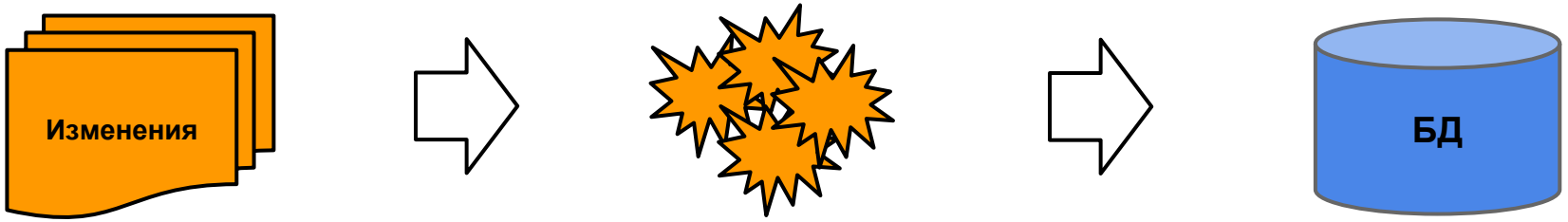
Обновление схемы БД

Какие проблемы могут возникнуть при обновлении схемы в БД?

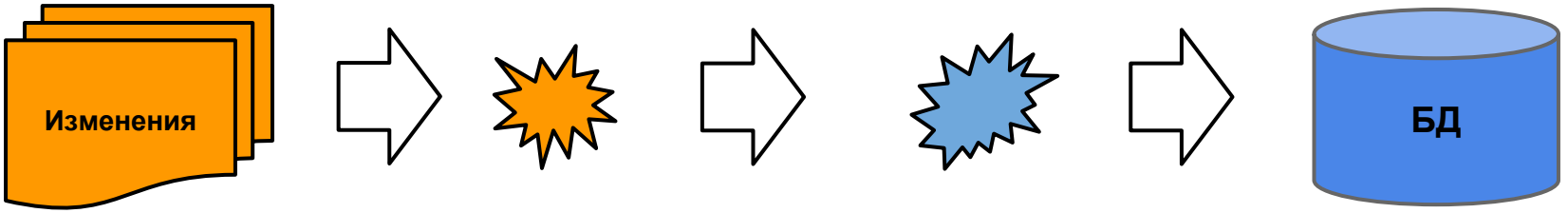
Миграции

это способ внесения изменений в структуру БД

Простой подход обновить схему БД



Атомарные, управляемые обновления БД (миграция)



Сохранение объектов в базу данных

```php
// src/Bundle/MainBundle/Controller/DefaultController.php
use Acme\StoreBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

public function createAction()
{
    $product = new Product();
    $product->setName('Notebook');
    $product->setPrice('319.99');
    $product->setDescription('Hi-tech product');

    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($product);
    $em->flush();

    ...
}
```

Работа EntityManager

- Метод persist() сообщает Doctrine команду на “управление” объектом $product. Она не вызывает создание запроса к базе данных (пока).
- Когда вызывается метод flush(), Doctrine просматривает все объекты, которыми она управляет, чтобы узнать, надо ли сохранить их в базу данных, и заносит изменения в БД.

Фактически, т.к. Doctrine знает обо всех управляемых сущностях, когда вызывается метод flush(), она прощитывает общий набор изменений и выполняет наиболее эффективный и возможный запрос или запросы.

Например, если сохраняется 100 объектов Product и впоследствии вызывается flush(), то Doctrine создаст *единственное* подготовленное выражение и повторно использует его для каждой вставки. Этот паттерн называется *Unit of Work,* это быстрый и эффективный способ.

Fixtures

Заранее подготовленные данные, для загрузки в БД

DoctrineFixturesBundle

Получение объектов из базы данных

```php
// src/Bundle/MainBundle/Controller/DefaultController.php

public function showAction($id)
{
    $em = $this->getDoctrine()->getEntityManager();
    $product = $em->getRepository('MainBundle:Product')->find($id);
     ...
}
```

Когда запрашивается объект определённого типа, всегда используется так называемый “репозиторий”.

Можно представить репозиторий как PHP класс, чья работа состоит в предоставлении помощи в получении сущностей определённого класса.

Обновление объекта

```
public function updateAction($id)
{
    $em = $this->getDoctrine()->getEntityManager();

    $product = $em->getRepository('MainBundle:Product')->find($id);
    $product->setName('New product name!');
    $em->flush();

    ...
}
```

Удаление объекта

```
public function deleteAction($id)
{
    $em = $this->getDoctrine()->getEntityManager();

    $product = $em->getRepository('MainBundle:Product')->find($id);
    $em->remove($product);
    $em->flush();

    ...
}
```

Работа с репозиторием

```php
$repository = $em->getRepository('MainBundle:Product');


// запрос по первичному ключу (обычно "id")
$product = $repository->find($id);

// динамические имена методов, использующиеся для поиска по значению столбцов
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');

// ищет *все* продукты
$products = $repository->findAll();

// ищет группу продуктов, основываясь на произвольном значении столбца
$products = $repository->findByPrice(19.99);

// запрос одного продукта, подходящего по заданным имени и цене
$product = $repository->findOneBy(array('name' => 'foo', 'price' => 19.99));
```

Связь репозитория и сущности

```php
/**
 * @ORM\Entity(repositoryClass="Bundle\MainBundle\Entity\ProductRepository")
 * @ORM\Table(name="product")
 */
class Product
{
    ...
}


/*********************************************************************************************/

class ProductRepository extends Doctrine\ORM\EntityRepository
{
    public function getById($id) {
        return $this->find($id);
    }
 }
```

Построение запросов

```
/** Запрос 1 элемента */

$repository->find($id); // запрос по Primary Key
$repository->findOneBy__($value);  // запрос по значению столбца
$repository->findOneBy(array( __ => $value, ...));  // запрос по значениям столбцов
// возвращает: класс сущности
```

```
/** Запрос множества элементов */

$repository->findAll(); // запрос всех сущностей
$repository->findBy__($value); // запрос по значению столбца
```

DQL

Использование языка DQL

- SELECT
- UPDATE
- DELETE
- JOIN
- ...

```
$query = $em->createQuery('SELECT p FROM Product p WHERE p.price > 19');
$products = $query->getResult();
```

Пример запроса на обновление сущности:

```
UPDATE MyProject\Model\Product p SET p.price = 20 WHERE p.id IN (1, 2, 3)
```

Пример запроса на удаление сущности:

```
DELETE MyProject\Model\Product p WHERE p.id = 4
```

Форматы результатов запроса

Hidration mode

- Query#**getResult**(): Возвращает коллекцию объектов, либо массив (смешанный).
- Query#**getSingleResult**(): Возвращает один объект. Если в результате запроса содержится более одного объекта или объект отсутствует, будет выброшено исключение.
- Query#**getOneOrNullResult**(): Возвращает один объект Если объект отсутствует будет возвращено значение NULL.
- Query#**getArrayResult**(): Возвращает массив графов (вложенный масив).
- Query#**getScalarResult**(): Возвращает плоский результирующий набор скалярных значений, который может содержать повторяющиеся данные.
- Query#**getSingleScalarResult**(): Возвращает единственное скалярное значение из результата запроса. Если результат содержит более одного такого значения, будет выброшено исключение.

Форматы результатов запроса

Пример:

```
$query = $em->createQuery('SELECT COUNT(p.id)
FROM Product p WHERE p.price > 20 GROUP BY p.id');

$count = $query->getSingleScalarResult();
    Или:
$count = $query->getResult(Query::HYDRATE_SINGLE_SCALAR);


echo $count; // Вывод: 12
```

Doctrine QueryBuilder:

Объект-ориентированный интерфейс запросов

Получение QB:

```
$queryBuilder = $repository->createQueryBuilder('alias');
```

Построение запроса с QB:

```
$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();

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

Связи/объединения сущностей

Прямая и обратная стороны связи

- 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 связи определяет какие изменения в существующем отношении попадут в базу данных.

Связи/объединения сущностей

Виды отношений:

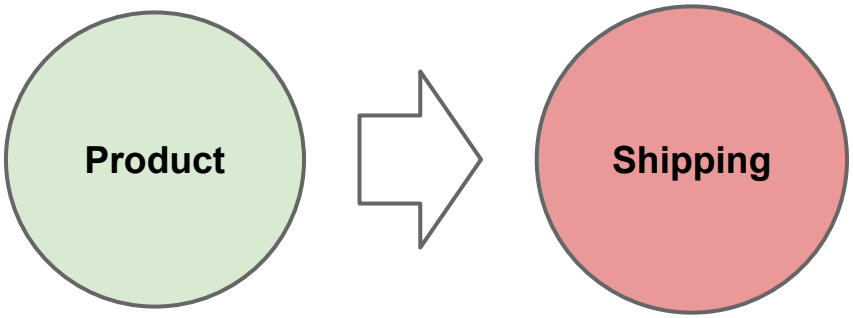
- Отношения “один к одному”, односторонние
- Отношения “один к одному”, двусторонние
- Отношения “один к одному” со ссылкой на себя же
- Отношения “один ко многим”, односторонние
- Отношения “многие к одному”, односторонние
- Отношения “один ко многим”, двусторонние
- Отношения “один ко многим” со ссылкой на себя
- Отношения “многие ко многим”, односторонние
- Отношения “многие ко многим”, двусторонние

Связи/объединения сущностей

Отношения “один к одному”, односторонние

Product (товар) имеет один объект *Shipping* (отгрузка товара). При этом в *Shipping* нет ссылки обратно на *Product*, поэтому отношение и называется односторонним: *Product -> Shipping*

```
** @Entity */
class Product {
    /**
    * @OneToOne(targetEntity="Shipping")
    */
    private $shipping;
 }

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

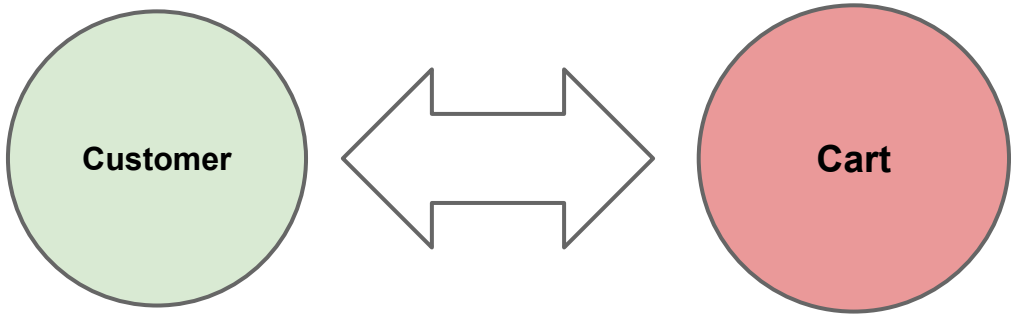


Связи/объединения сущностей

Отношения “один к одному”, двусторонние

Customer (заказчик) и *Cart* (корзина). У *Cart* есть обратная ссылка на *Customer*, поэтому эта связь является двусторонней:

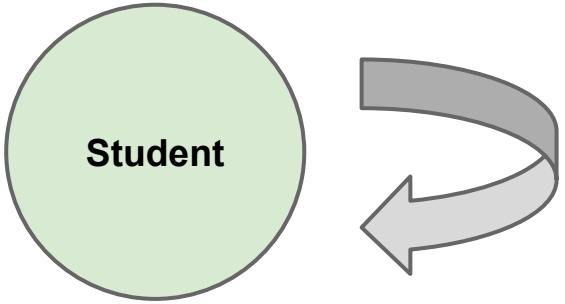
```
/** @Entity */
class Customer {
   /**
    * @OneToOne(targetEntity="Cart",
        mappedBy="customer")
    */
    private $cart;
 }
/** @Entity */
class Cart {
   /**
    * @OneToOne(targetEntity="Customer",
        inversedBy="cart")
    */
    private $customer;
}
```



Связи/объединения сущностей

Отношения “один к одному” со ссылкой на себя же

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

    /**
     * @OneToOne(targetEntity="Student")
     */
    private $mentor;

    // ...
}
```

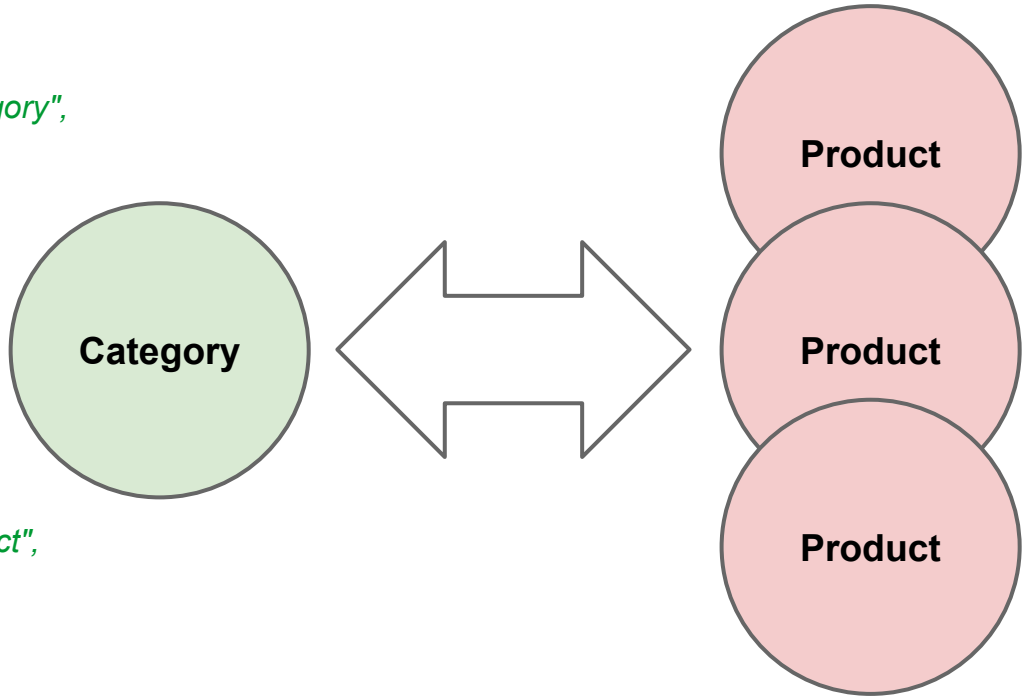


Связи/объединения сущностей

Отношения “один ко многим”, двусторонние

```
class Product
{
    /**
     * @ORM\ManyToOne (targetEntity="Category",
inversedBy="products")
     */
    protected $category;
}
```

```
class Category
{

    /**
     * @ORM\OneToMany(targetEntity="Product",
mappedBy="category")
     */
    protected $products;
}
```



to be continued...

Связи/объединения сущностей

Пример использования:

```
// src/Bunle/MainBundle/Entity/Category.php
// ...
use Doctrine\Common\Collections\ArrayCollection;

class Category
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="Product", mappedBy="category")
     */
    protected $products;


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

Объект **Category** связан со множеством объектов **Product**, свойство **products** будет массивом для хранения объектов **Product**

Связи/объединения сущностей

Пример использования:

```
// src/Bundle/MainBundle/Entity/Product.php
// ...

class Product
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
     * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
     */
    protected $category;
}
```

Сообщаем Doctrine что хотим сгенерировать отсутствующие методы getter и setter:

php app/console doctrine:generate:entities MainBundle

Перед тем как продолжить, убедитесь что сообщили Doctrine обновить БД:

php app/console doctrine:schema:update --force

Сохранение связанных сущностей

```php
$category = new Category();
$category->setName('Main Products');

$product = new Product();
$product->setName('Foo');
$product->setPrice(19.99);

// Связываем этот продукт с категорией
$product->setCategory($category);

$em = $this->getDoctrine()->getEntityManager();
$em->persist($category);
$em->persist($product);
$em->flush();
```

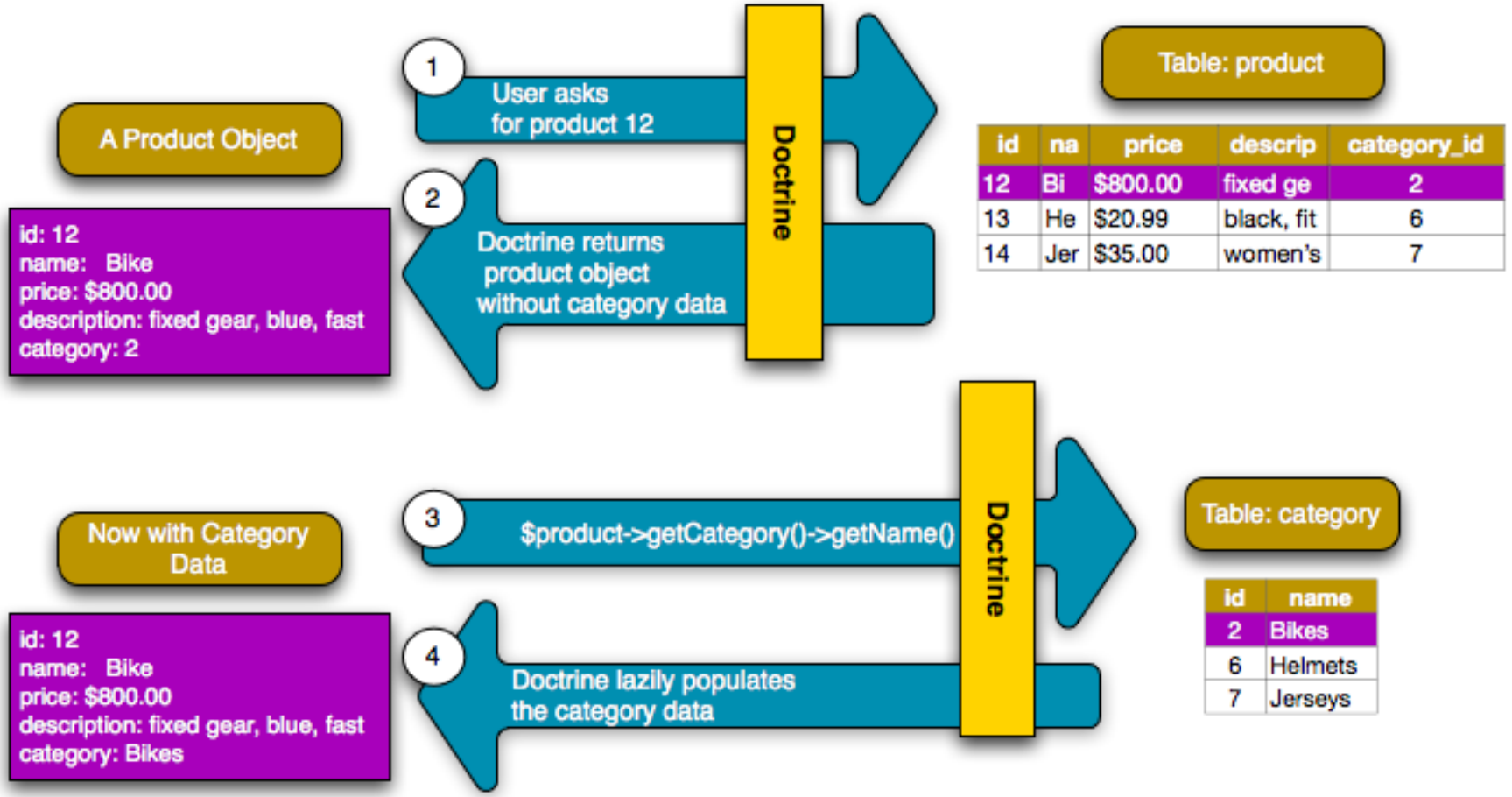
Получение связанных объектов

```
$product = $this->getDoctrine()->getRepository('MainBundle:Product')->find(12);
 $categoryName = $product->getCategory()->getName();
```

Рассмотрим как класс Продукт позволяет нам получить класс Категория:

```
class Product
{

    /** ... */
    protected $category;


    /**
     * Get Category
     *
     * @return \Bundle\MainBundle\Entity\Category
     */
    public function getCategory()
    {
        return $this->category;
    }
}
```

Получение связанных объектов



Получение связанных объектов

Также можно запросить в другом направлении:

```
$category = $em->getRepository('MainBundle:Category')->find($id);
$products = $category->getProducts();


$products;     // ArrayCollection of Product
foreach ($products as $product) {
    echo $product->getPrice();
}
```

Получение связанных объектов

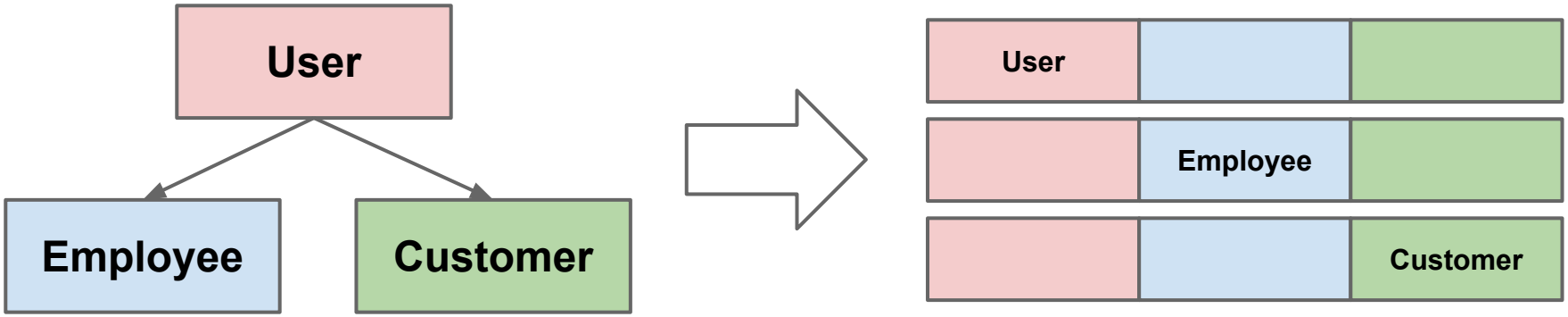
Если заранее известно, что будет необходим доступ к обоим объектам, то можно избежать двойного запроса, используя join в исходном запросе:

```
$query = $this->getEntityManager()
    ->createQuery(
        'SELECT p, c FROM MainBundle:Product p
        JOIN p.category c
        WHERE p.id = :id'
    )->setParameter('id', $id);

 return $query->getSingleResult();

 // ..... ИЛИ .....

 $product = $this->getDoctrine()
    ->getRepository(MainBundle:Product')
    ->findOneByIdJoinedToCategory($id);

 $category = $product->getCategory();
```

Наследование с единой таблицей

(Single Table Inheritance)

- это стратегии наследования отображений, в которой все классы в иерархии отображаются на одну единственную таблицу в базе данных.
- Чтобы определить какая запись каком классу соответствует, используется специальный столбец, называемый *столбцом дискриминатора*.



Наследование с единой таблицей

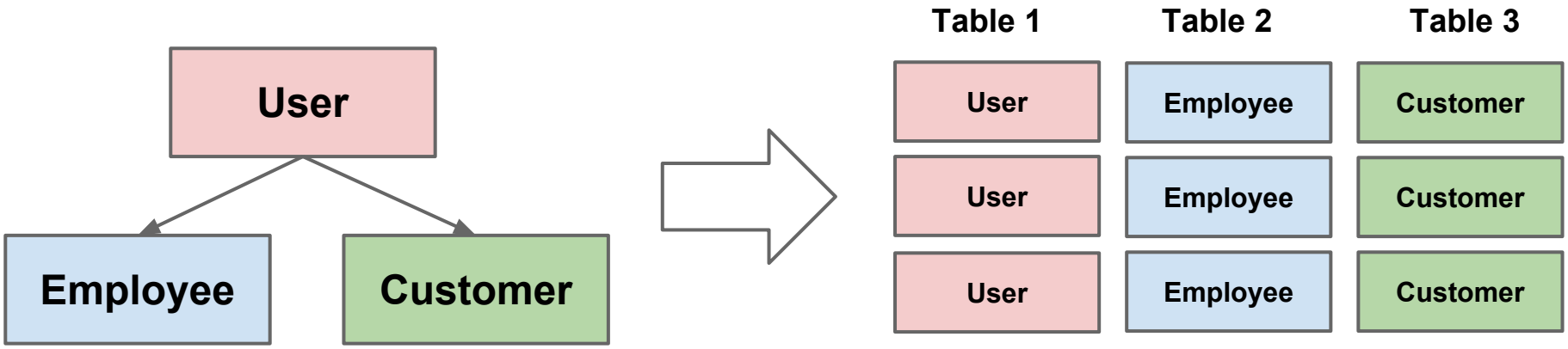
(Single Table Inheritance)

```
/**
* @Entity
* @InheritanceType("SINGLE_TABLE")
* @DiscriminatorColumn(name="discr", type="string")
* @DiscriminatorMap({"user" = "User", "employee" = "Employee", "customer" = "Customer"})
*/
class User { /* … */ }

/**
* @Entity
*/
class Employee extends User { /* … */ }

/**
* @Entity
*/
class Customer extends User { /* … */ }
```

Наследование с таблицами классов

(Class Table Inheritance)

- представляет собой стратегию, в которой каждый класс в иерархии отображается в итоге на несколько таблиц: одну — его собственную, остальные — для всех родительских классов.
- Таблица производного класса будет связана с таблицей класс-родителя спомощью внешнего ключа. В *Doctrine 2* эта стратегия реализована с применением столбца дискриминатора для “верхней” таблицы в иерархии.



Наследование с таблицами классов

(Class Table Inheritance)

```
/**
* @Entity
* @InheritanceType("JOINED")
* @DiscriminatorColumn(name="discr", type="string")
* @DiscriminatorMap({"user" = "User", "employee" = "Employee", "customer" = "Customer"})
*/
class User { /* … */ }

/**
* @Entity
*/
class Employee extends User { /* … */ }

/**
* @Entity
*/
class Customer extends User { /* … */ }
```

Lifecycle Callbacks

Стадии жизненного цикла сущности:

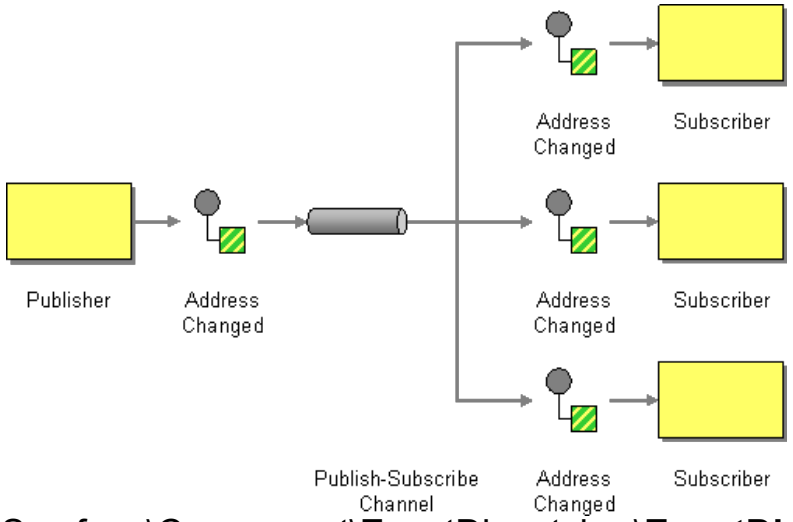
- preRemove
- postRemove
- prePersist
- postPersist
- preUpdate
- postUpdate
- postLoad
- loadClassMetadata
- preFlush
- onFlush
- postFlush
- onClear

Lifecycle Callbacks

Пример использования:

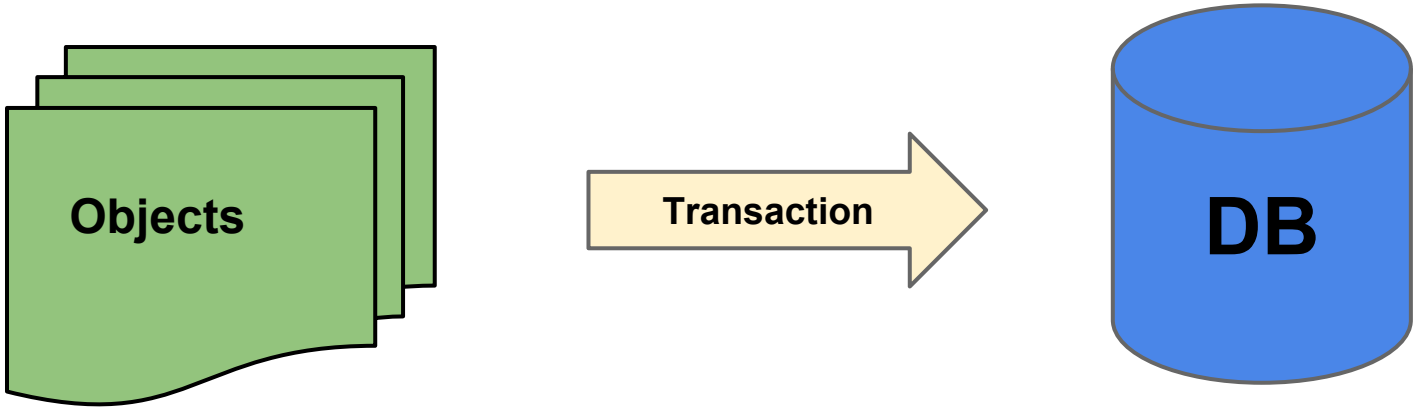
```
// src/Bundle/MainBundle/Entity/Product.php
class Product
{
    /**
     * @ORM\PrePersist
     */
    public function setCreatedAtValue()          // не принимает аргументов!
    {
        $this->createdAt = new \DateTime();
    }

    /* … */
}
```

Listener/Subscriber



- Диспетчер событий (Symfony\Component\EventDispatcher**EventDispatcher**) - выступает в роли Publisher'а, отвечает за транслирование этих событий на всю систему, можно сказать, посылает broadcast message с событием
- Событие (Symfony\Component\EventDispatcher**Event**) - это и есть событие, которое мы будем транслировать, если рассматривать на примере картинки с паттерном, то это Address Changed
- Слушатель или Подписчик (Listener/Subscriber) — это сервис, который мы подписываем на конкретное событие.
- Различие между этими двумя сервисами, это то что Listener подписывается на статичное событие посредством описания сервиса в конфиге, Subscriber же может быть динамически подписан на событие в ходе выполнения приложения.

Транзакции



Транзакции

Пример явной транзакции ORM EntityManager

```
$em->getConnection()->beginTransaction();
try {
    $user = new User;
    $user->setName('George');
    $em->persist($user);
    $em->flush();
    $em->getConnection()->commit();
} catch (Exception $e) {
    $em->getConnection()->rollback();
    $em->close();
}
```

Задание

Добавить новый столбец на основе базы данных (т.е. добавить в базу, изменить запросы, добавить в вывод).