Model i Doctrine

v3.1

Plan

- Doctrine
- Rozpoczęcie pracy z Doctrine
- Model - podstawy stosowania
- Praca z encjami
- Relacje w Doctrine
- Zaawansowane tematy

Doctrine

Co to jest Doctrine?

Doctrine to zbiór bibliotek w PHP skupiających się przede wszystkim na obsłudze różnych rodzajów baz danych oraz mapowaniu ich struktury na obiekty w PHP.

Oficjalna strona projektu:

- http://www.doctrine-project.org

Kod projektu:

- http://github.com/doctrine

Co to jest Doctrine?

- **Doctrine nie jest integralną częścią Symfony.**
- Samo Symfony nie ma żadnej warstwy obsługi baz danych, dopiero dystrybucja Symfony Standard Edition integruje się z **Doctrine**.
- Ten zbiór bibliotek możemy używać też w projektach, które nie korzystają z Symfony.

Doctrine

Doctrine składa się przede wszystkim z dwóch kluczowych elementów:

- **DBAL** (Database Abstraction Layer) – warstwa abstrakcji baz danych, rozszerzająca funkcjonalności PDO (PHP Data Objects),
- **ORM** (Object Relational Mapper) – warstwa mapowania obiektów w PHP na strukturę relacyjnych baz danych opartą na DBAL. ORM wprowadza dialekt DQL alternatywny wobec SQL.

Doctrine składa się także m.in. z:

- **Migrations** – narzędzia służące do wersjonowania struktury baz danych,
- **ODM** (Object Document Mapper) – warstwa mapowania obiektów na tzw. dokumentowe bazy danych (NoSQL), tj. MongoDB, CouchDB

Rozpoczęcie pracy z Doctrine

Konfigurowanie połączenia

Doctrine używa konfiguracji zdefiniowanej w swoim bloku w pliku **app/config/config.yml**

W tym pliku nie powinniśmy wpisywać wartości!

Warto natomiast pamiętać o nim jeśli byśmy chcieli użyć różnych baz danych.

Więcej informacji w dokumentacji:
https://symfony.com/doc/2.8/doctrine.html

```yaml
doctrine:
  dbal:
    driver:     "%database_driver%"
    host:       "%database_host%"
    port:       "%database_port%"
    dbname:     "%database_name%"
    user:       "%database_user%"
    password:   "%database_password%"
    charset:    UTF8
    # path:     "%database_path%"
```

Konfigurowanie połączenia

Wartości ujęte w znaki **"%"** są parametrami, które definiuje się w pliku: **app/config/parameters.yml**

To w tym pliku powinniśmy wpisywać poprawne wartości.

```
parameters:
  database_driver: pdo_mysql
  database_host: 127.0.0.1
  database_port: null
  database_name: symfony
  database_user: root
  database_password: null
  # database_path: %kernel.root_dir%/data/data.db
```

Stworzenie bazy danych

Jeśli konfiguracja połączenia jest poprawna, a użytkownik ma odpowiednie uprawnienia, wtedy **Doctrine** może utworzyć dla nas właściwą bazę danych:

Utworzenie bazy danych

```
php app/console doctrine:database:create [--env=dev|prod]
```

Usunięcie bazy danych

```
php app/console doctrine:database:drop --force [--env=dev|prod]
```

Zadania

Model Podstawy stosowania

Model

- We wzorcu projektowym model jest klasą, która reprezentuje nam tabelę z bazy danych.
- Jeden obiekt tej klasy reprezentuje jeden wiersz w tej tabeli.
- W **Doctrine** obiekty te są nazywane **encjami**.

- Ideologicznie na jedną klasę modelu powinna przypadać jedna klasa kontrolera, nie jest błędem używanie w kontrolerze wielu modeli. .
- Zadaniem modelu jest utrzymanie informacji i ich synchronizacja z bazą danych.
- Zadaniem kontrolera jest korzystanie z modelu (wczytywanie, wpisywanie nowych danych itp.).

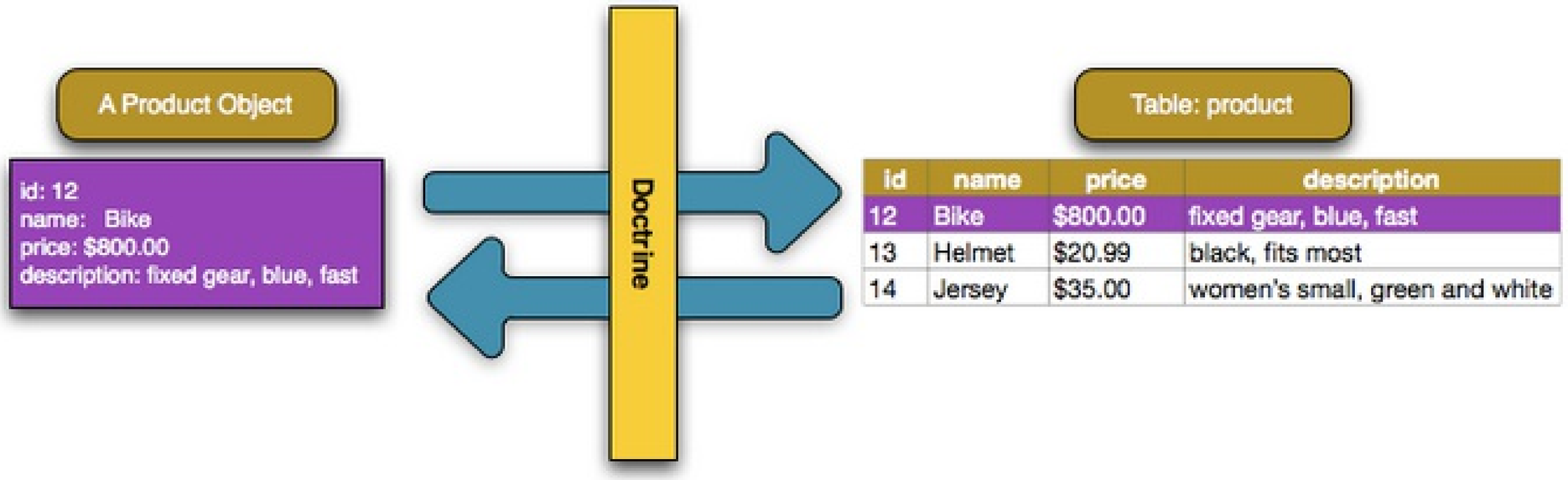
Model, encja

Głównym zadaniem encji jest przenoszenie informacji. Zgodnie z MVC mogą one także realizować proste zadania logiczne, np.:

- encja zamówienia może zwracać jego wartość jako sumę cen poszczególnych elementów zamówienia,
- encja kupon rabatowy może sprawdzać, czy przysługuje on danemu użytkownikowi,

- ważne żeby pamiętać o ogólnej koncepcji modelu, który ma dostarczać dane dla aplikacji w wygodnej do użycia formie.

Model, encja



id	name	price	description
12	Bike	$800.00	fixed gear, blue, fast
13	Helmet	$20.99	black, fits most
14	Jersey	$35.00	women's small, green and white

Namespace encji

Encje – tak samo jak w przypadku modeli – muszą należeć do swojego **namespace**.

Zawsze należy dodać **namespace** na początku pliku z encją gdyż bez tego nasza encja nie będzie działać.

```
namespace Bundle_name\Entity;
```

Ta linia musi znajdować się na początku pliku z encją. Oczywiście wpisujemy swojego bundla.

Encja

- W **Doctrine** korzystamy z systemu adnotacji (znanego nam już np. z routingu).
- Adnotacje **Doctrin**e najpierw musimy podlinkować.
- Dzięki adnotacjom będziemy mogli połączyć klasę z tabelą, a jej atrybuty z poszczególnymi kolumnami.

```
use Doctrine\ORM\Mapping as ORM;
```

Na początku pliku z klasą

Klasa encji

- Encje nie dziedziczą po żadnej specjalnej klasie.
- Wystarczy, że będą znajdować się w odpowiednim namespace i będą mieć adnotację **@ORM\Entity**.

```php
/**
 * @ORM\Entity
 */
class Post
{
    /* ... */
}
```

Łączenie encji z tabelą

W celu połączenia encji z tabelą wystarczy dopisać adnotację:

- **@ORM\Table(name="table_name")**

```
/**
 * @ORM\Entity
 * @ORM\Table(name="posts")
 */

class Post
{
    /* ... */
}
```

Mapowanie atrybutów

- Kiedy mamy przypisaną tabelę do klasy encji, możemy przypisywać poszczególne atrybuty do kolumn.
- Nie wszystkie atrybuty muszą być przypisane!
- Atrybut przypisujemy za pomocą adnotacji:
  `@ORM\Column()`

@ORM\Column()

Adnotacja **@ORM\Column()** przyjmuje następujące informacje:

type
Typ kolumny w bazie danych.

name
Nazwa kolumny w bazie danych (jeżeli nie podamy, będzie taka sama jak atrybutu).

length
Długość (odnosi się do napisów), podstawowo nastawione na **255**.

unique
Czy ma być unikalna (domyślnie nie jest).

precision
Precyzja liczb zmiennoprzecinkowych (łączna liczba cyfr).

scale
Precyzja liczb zmiennoprzecinkowych (liczba cyfr po przecinku).

Typy danych w Doctrine

Adnotacji **@ORM\Column()** musimy podać typ danych. Dostępne typy to (nie wszystkie):

- **string,**
- **integer,**
- **smallint,**
- **bigint,**
- **decimal,**
- **time,**
- **datetime,**
- **text,**
- **object** (o tym będzie jeszcze później)

>>> Więcej o typach danych

Klucze główne

- W każdej tabeli musi być klucz główny.
- Żeby wskazać **Doctrine**, który z naszych atrybutów jest kluczem głównym, stosujemy adnotację **`@ORM\Id`** razem z **`@ORM\GeneratedValue`**.

```
/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 *
 */
 private $id;
```

Klucze główne

- W każdej tabeli musi być klucz główny.
- Żeby wskazać **Doctrine**, który z naszych atrybutów jest kluczem głównym, stosujemy adnotację **@ORM\Id** razem z **@ORM\GeneratedValue**.

```
/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 *
 */
 private $id;
```

Primary key

Klucze główne

- W każdej tabeli musi być klucz główny.
- Żeby wskazać **Doctrine**, który z naszych atrybutów jest kluczem głównym, stosujemy adnotację **`@ORM\Id`** razem z **`@ORM\GeneratedValue`**.

```
/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 *
 */
 private $id;
```

Kolumna będzie przechowywać liczby całkowite

Klucze główne

- W każdej tabeli musi być klucz główny.
- Żeby wskazać **Doctrine**, który z naszych atrybutów jest kluczem głównym, stosujemy adnotację **@ORM\Id** razem z **@ORM\GeneratedValue**.

```
/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 *
 */
 private $id;
```

Pole ma być tworzone automatycznie (**AUTO INCREMENT**)

Przykład mapowania atrybutów

```php
/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 */
 private $id;

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

```php
/**
 * @ORM\Column(type="decimal", precision=4, scale=2)
 */
 protected $rating;

/**
 * @ORM\Column(type="text")
 */
 protected $postText;
```

Tworzenie tabeli na podstawie Encji

W tej chwili mamy klasę encji, ale nie mamy jeszcze stworzonej do niej tabeli. Ręczne generowanie takich tabel byłoby żmudne i mogłoby prowadzić do błędów (np. zła nazwa kolumny).

Doctrine może wygenerować tablicę za nas:

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

Encja

W tej chwili mamy w pełni działającą encje z punktu widzenia **Doctrine.**

Aby model był funkcjonalny z naszego punktu widzenia musimy jeszcze:

- napisać funkcje **set** i **get** do tych atrybutów, do których chcemy dać dostęp.

Oczywiście nie musimy robić tego ręcznie. Służy do tego komenda konsolowa:

```
php app/console doctrine:generate:entities
My_Bundle/Entity/My_Entity
```

Zostanie również wygenerowane repozytorium. Zastosujemy je w praktyce później.

Encja

W tej chwili mamy w pełni działającą encje z punktu widzenia **Doctrine.**

Aby model był funkcjonalny z naszego punktu widzenia musimy jeszcze:

- napisać funkcje **set** i **get** do tych atrybutów, do których chcemy dać dostęp.

Oczywiście nie musimy robić tego ręcznie. Służy do tego komenda konsolowa:

```
php app/console doctrine:generate:entities
My_Bundle/Entity/My_Entity
```

To jest jedna komenda konsolowa

Zostanie również wygenerowane repozytorium. Zastosujemy je w praktyce później.

Encja

Komenda z poprzedniego slajdu robi trzy rzeczy:

- generuje setery i getery (pamiętajcie o usunięciu niepotrzebnych),
- tworzy klasę repozytorium (opcjonalnie),
- tworzy poprawne konstruktory dla relacji jeden do wielu i wiele do wielu.

Generowanie encji

Encje można też wygenerować interaktywnie czyli bardzo podobnie jak w przypadku generowania np. kontrolerów:

```
php app/console doctrine:generate:entity
```

Powyższa komenda generuje również repozytorium.

Wygenerowaną encję wraz z mapowaniem trzeba zainicjalizować jako strukturę w bazie danych:

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

Zadania

Praca z encjami

Praca z encją

Na obiekcie encji pracujemy w akcjach kontrolera. Trzeba pamiętać o podlinkowaniu encji w pliku z kontrolerem.

```
use My_Bundle\Entity\My_EntityClass;
```

Praca z encją

Na obiekcie encji pracujemy w akcjach kontrolera. Trzeba pamiętać o podlinkowaniu encji w pliku z kontrolerem.

```php
use My_Bundle\Entity\My_EntityClass;
```

Wpisujemy swojego bundla i nazwę modelu

Zapisywanie nowej encji do bazy danych

Na samym początku musimy stworzyć nowy obiekt naszego modelu w kontrolerze.

Tworzymy go jak zwykły obiekt z użyciem słowa kluczowego **new**.

```
public function createAction()
{
    $firstPost = new Post();
    $firstPost->setTitle('My first post');
    $firstPost->setRaiting(6.4);
    $firstPost->setPostText('Lorem ipsum dolor...');
    /* ... */
}
```

Zapisywanie nowej encji do bazy danych

Na samym początku musimy stworzyć nowy obiekt naszego modelu w kontrolerze.

Tworzymy go jak zwykły obiekt z użyciem słowa kluczowego **new**.

```php
public function createAction()
{
    $firstPost = new Post();
    $firstPost->setTitle('My first post');
    $firstPost->setRaiting(6.4);
    $firstPost->setPostText('Lorem ipsum dolor...');
    /* ... */
}
```

Tworzymy nowy obiekt modelu

Zapisywanie nowej encji do bazy danych

Na samym początku musimy stworzyć nowy obiekt naszego modelu w kontrolerze.

Tworzymy go jak zwykły obiekt z użyciem słowa kluczowego **new**.

```
public function createAction()
{
    $firstPost = new Post();
    $firstPost->setTitle('My first post');
    $firstPost->setRaiting(6.4);
    $firstPost->setPostText('Lorem ipsum dolor...');
    /* ... */
}
```

Ustawiamy seterami odpowiednie wartości

Zapisywanie nowej encji do bazy danych

Następnie musimy wykonać trzy kroki:

- wczytać obiekt klasy **`EntityManager`** – jest to obiekt, który zarządza naszymi encjami (zapamiętuje je, wczytuje, itp.),
- poinformować go o naszym nowym obiekcie,
- użyć metody **`flush`** na naszym managerze.

Ciąg dalszy naszej akcji

```
public function createAction()
{
    /* ... */
    $em = $this->getDoctrine()->getManager();

    $em->persist($firstPost);

    $em->flush();

    return new Response('New post - id: ' . $firstPost->getId());
}
```

Zapisywanie nowej encji do bazy danych

Następnie musimy wykonać trzy kroki:

- wczytać obiekt klasy **`EntityManager`** – jest to obiekt, który zarządza naszymi encjami (zapamiętuje je, wczytuje, itp.),
- poinformować go o naszym nowym obiekcie,
- użyć metody **`flush`** na naszym managerze.

Ciąg dalszy naszej akcji

```
public function createAction()
{
    /* ... */
    $em = $this->getDoctrine()->getManager();

    $em->persist($firstPost);

    $em->flush();

    return new Response('New post - id: ' . $firstPost->getId());
}
```

Wczytujemy **`EntiryManagera`**

Zapisywanie nowej encji do bazy danych

Następnie musimy wykonać trzy kroki:

- wczytać obiekt klasy **EntityManager** – jest to obiekt, który zarządza naszymi encjami (zapamiętuje je, wczytuje, itp.),
- poinformować go o naszym nowym obiekcie,
- użyć metody **flush** na naszym managerze.

Ciąg dalszy naszej akcji

```
public function createAction()
{
    /* ... */
    $em = $this->getDoctrine()->getManager();

    $em->persist($firstPost);

    $em->flush();

    return new Response('New post - id: ' . $firstPost->getId());
}
```

Informujemy go o naszej encji

Zapisywanie nowej encji do bazy danych

Następnie musimy wykonać trzy kroki:

- wczytać obiekt klasy **EntityManager** – jest to obiekt, który zarządza naszymi encjami (zapamiętuje je, wczytuje, itp.),
- poinformować go o naszym nowym obiekcie,
- użyć metody **flush** na naszym managerze.

Ciąg dalszy naszej akcji

```
public function createAction()
{
    /* ... */
    $em = $this->getDoctrine()->getManager();

    $em->persist($firstPost);

    $em->flush();

    return new Response('New post - id: ' . $firstPost->getId());
}
```

Używamy metody **flush**

Zapisywanie nowej encji do bazy danych

Następnie musimy wykonać trzy kroki:

- wczytać obiekt klasy **`EntityManager`** – jest to obiekt, który zarządza naszymi encjami (zapamiętuje je, wczytuje, itp.),
- poinformować go o naszym nowym obiekcie,
- użyć metody **`flush`** na naszym managerze.

Ciąg dalszy naszej akcji

```
public function createAction()
{
    /* ... */
    $em = $this->getDoctrine()->getManager();

    $em->persist($firstPost);

    $em->flush();

    return new Response('New post - id: ' . $firstPost->getId());
}
```

Pamiętamy, że akcja musi zwracać obiekt **Response**

Metoda persist() i flush()

- Metoda **`persist()`** informuje **Doctrine** o naszej nowej encji. Na razie jednak nie jest ona zapamiętana w bazie danych.

- Dopiero metoda **`flush()`** powoduje, że **Doctrine** sprawdza stan wszystkich znanych sobie encji.
- Jeżeli ten stan różni się od tego, co jest w bazie danych, to dane są dodawane lub aktualizowane.

Repozytoria

- Do wczytywania encji z bazy danych potrzebujemy repozytorium.
- O repozytoriach można myśleć jak o obiektach, których zadaniem jest pomoc w pobieraniu encji (pojedynczych lub kolekcji) określonego typu z bazy danych.
- Podstawowe repozytorium jest zawsze dla nas dostępne.
- Oznacza to, że podstawowe metody związane z pobieraniem danych są dla nas dostępne.

Wczytywanie istniejącej encji

Kiedy mamy już repozytorium dla naszych encji, możemy wczytać obiekty naszej klasy z bazy danych. Pierwszym krokiem jest załadowanie odpowiedniego repozytorium:

```
public function showPostAction()
{
    $em = $this->getDoctrine()->getManager();
    $repository = $em->getRepository('My_bundle:My_Entity');

    /* ... */
```

Wczytywanie istniejącej encji

Kiedy mamy już repozytorium dla naszych encji, możemy wczytać obiekty naszej klasy z bazy danych. Pierwszym krokiem jest załadowanie odpowiedniego repozytorium:

```
public function showPostAction()
{
    $em = $this->getDoctrine()->getManager();
    $repository = $em->getRepository('My_bundle:My_Entity');

    /* ... */
```

Gdzieś w kontrolerze

Wczytywanie istniejącej encji

Następnie możemy użyć metod naszego podstawowego repozytorium.

```
$post = $repository->find($id);
$post = $repository->findOneById($id);
$post = $repository->findOneByTitle('foo');
$post = $repository->findOneByRaiting(4.0);
$post = $repository->findOneByPostText('Some text...');
```

Wczytywanie istniejącej encji

Następnie możemy użyć metod naszego podstawowego repozytorium.

```php
$post = $repository->find($id);
$post = $repository->findOneById($id);
$post = $repository->findOneByTitle('foo');
$post = $repository->findOneByRaiting(4.0);
$post = $repository->findOneByPostText('Some text...');
```

Wyszukuj po kluczu głównym

Wczytywanie istniejącej encji

Następnie możemy użyć metod naszego podstawowego repozytorium.

```
$post = $repository->find($id);
$post = $repository->findOneById($id);
$post = $repository->findOneByTitle('foo');
$post = $repository->findOneByRaiting(4.0);
$post = $repository->findOneByPostText('Some text...');
```

Wyszukuj po danym atrybucie (te metody są tworzone dynamicznie dla każdego atrybutu)

Wczytywanie wielu encji

Z repozytorium możemy wczytać wiele encji. Otrzymamy wtedy tablicę z tymi encjami.

```
$allPosts = $repository->findAll();
$posts = $repository->findByRaiting(3.65);
$posts = $repository->findByTitle('foo');
$posts = $repository->findByPostText('Some text...');
```

Wczytywanie wielu encji

Z repozytorium możemy wczytać wiele encji. Otrzymamy wtedy tablicę z tymi encjami.

```
$allPosts = $repository->findAll();
$posts = $repository->findByRaiting(3.65);
$posts = $repository->findByTitle('foo');
$posts = $repository->findByPostText('Some text...');
```

Znajdzie nam wszystkie encje. Wynik może być bardzo długi.

Wczytywanie wielu encji

Z repozytorium możemy wczytać wiele encji. Otrzymamy wtedy tablicę z tymi encjami.

```
$allPosts = $repository->findAll();
$posts = $repository->findByRaiting(3.65);
$posts = $repository->findByTitle('foo');
$posts = $repository->findByPostText('Some text...');
```

Znajdzie wszystkie encje spełniające założenia dla danej kolumny.

Wczytywanie encji po wielu kolumnach

Encje możemy też wczytywać po wielu kolumnach. Do metody **findBy** musimy przekazać wtedy tablicę, gdzie kluczem jest nazwa kolumny, a zawartością szukana wartość.

```
$post = $repository->findOneBy(
    ['title' => 'foo', 'raiting' => 2.00]
);

$posts = $repository->findBy(
    ['title' => 'foo', 'raiting' => 2.00]
);
```

Zapisywanie nowego stanu encji

- Zapisywanie już istniejącej encji do bazy danych jest bardzo proste.
- Wystarczy wprowadzić zmiany do encji, a następnie wywołać metodę **`flush()`**.

Encja wczytana z bazy danych jest od razu zapamiętywana w **`EntityManagerze`**, więc nie musimy go informować o jej istnieniu.

```
$em = $this->getDoctrine()->getManager();
$post = $em->getRepository('MyBundle:Post')->find($id);

if (!$post) {
    // Sprawdzamy, czy post o podanym id istnieje...
}

$post->setTitle('Zmieniony tytuł!');
$em->flush();
```

Usuwanie encji

Usunięcie encji z bazy danych polega na wczytaniu jej, następnie użyciu metody **`remove()`** na obiekcie **`EntityManager`**, a następnie zapamiętaniu stanu bazy danych poprzez metodę **`flush()`**.

```
$em = $this->getDoctrine()->getManager();
$post = $em->getRepository('MyBundle:Post')->find($id);

if (!$post) {
    // Sprawdzamy, czy post o podanym id istnieje...
}
$em->remove($post);
$em->flush();
```

Zadania

Relacje w Doctrine

Relacje

W **Doctrine** mamy możliwość stworzenia relacji między encjami. Jest ich o wiele więcej niż w czystym SQL.

Podczas kursu omówimy tylko najbardziej przydatne relacje.

Relacje, które omówimy:

- wiele do jednego, jednokierunkowa,
- jeden do wielu, dwukierunkowa,
- jeden do jednego, dwukierunkowa,
- wiele do wielu, dwukierunkowa.

Pełną listę możecie znaleźć tutaj:
http://doctrine-orm.readthedocs.org/projects/doctrine-orm/en/latest/reference/association-mapping.html

Wiele do jednego, jednokierunkowa

Jest to relacja, w której:

- encja A wie o jednej przypisanej do siebie encji B,
- encja B nie wie nic o encjach A (może być przypisana do wielu).

Użytkownik wie, jaki ma adres, ale adres nie ma informacji o przypisanych do siebie użytkownikach.

```
/** @ORM\Entity */
class User{
    /**
     * @ORM\ManyToOne(targetEntity="Address")
     * @ORM\JoinColumn(name="address_id", referencedColumnName="id")
     */
    private $address;
}

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

Wiele do jednego, jednokierunkowa

Jest to relacja, w której:

- encja A wie o jednej przypisanej do siebie encji B,
- encja B nie wie nic o encjach A (może być przypisana do wielu).

Użytkownik wie, jaki ma adres, ale adres nie ma informacji o przypisanych do siebie użytkownikach.

```
/** @ORM\Entity */
class User{
    /**
     * @ORM\ManyToOne(targetEntity="Address")
     * @ORM\JoinColumn(name="address_id", referencedColumnName="id")
     */
    private $address;
}

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

Encja User

Wiele do jednego, jednokierunkowa

Jest to relacja, w której:

- encja A wie o jednej przypisanej do siebie encji B,
- encja B nie wie nic o encjach A (może być przypisana do wielu).

Użytkownik wie, jaki ma adres, ale adres nie ma informacji o przypisanych do siebie użytkownikach.

```
/** @ORM\Entity */
class User{
    /**
     * @ORM\ManyToOne(targetEntity="Address")
     * @ORM\JoinColumn(name="address_id", referencedColumnName="id")
     */
    private $address;
}

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

Wskazanie na relację z encją Address

Wiele do jednego, jednokierunkowa

Jest to relacja, w której:

- encja A wie o jednej przypisanej do siebie encji B,
- encja B nie wie nic o encjach A (może być przypisana do wielu).

Użytkownik wie, jaki ma adres, ale adres nie ma informacji o przypisanych do siebie użytkownikach.

```
/** @ORM\Entity */
class User{
    /**
     * @ORM\ManyToOne(targetEntity="Address")
     * @ORM\JoinColumn(name="address_id", referencedColumnName="id")
     */
    private $address;
}

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

Wskazanie na relację kolumny **address_id** z encji **User** z kolumną **id** z encji **Address**

Jeden do wielu, dwukierunkowa

Jest to relacja, w której:

- encja A ma wiele encji B,
- encja B może mieć tylko jedną encje A.
- obie encje wiedzą o sobie.

Jest to na przykład produkt w sklepie internetowym i opinie o nim.

Produkt może mieć wiele opinii, ale opinia należy tylko do jednego produktu.

Jeden do wielu, dwukierunkowa

```php
class Product{
    /**
     * @ORM\OneToMany(targetEntity="Review", mappedBy="product")
     */
    private $reviews;

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

```php
class Review{
    /**
     * @ORM\ManyToOne(targetEntity="Product", inversedBy="reviews")
     * @ORM\JoinColumn(name="product_id", referencedColumnName="id")
     */
    private $product;
}
```

Relacje do wielu obiektów

- Jeśli trzymamy relacje do wielu innych obiektów, to musimy specjalnie zainicjalizować atrybut naszej encji.
- Atrybut, który będzie trzymał wiele encji, musi być obiektem klasy**ArrayCollection.**

Musimy pamiętać o podlinkowaniu obiektu ArrayCollection.

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

Jeden do jednego, dwukierunkowa

W tej relacji obie encje wiedzą o sobie wzajemnie. Na przykład użytkownik wie o swoim koszyku, koszyk wie, do kogo jest przypisany.

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

class Cart{
    /**
     * @ORM\OneToOne(targetEntity="Customer", inversedBy="cart")
     * @ORM\JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;
}
```

Wiele do wielu, dwukierunkowa

W tej relacji encja A wie o wielu encjach B, a encja B wie o wielu encjach A.

Na przykład użytkownik może być w wielu grupach, a grupa ma wielu użytkowników.

```
class User{
    /**
     * @ORM\ManyToMany(targetEntity="Group", inversedBy="users")
     * @ORM\JoinTable(name="users_groups")
     */
    private $groups;

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

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

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

Wczytywanie połączonych encji

Wczytanie połączonej encji jest niezwykle proste. Wystarczy odwołać się do atrybutu, który oznaczyliśmy relacją, i otrzymujemy podłączoną encję (lub tablice encji).

```
$em = $this->getDoctrine()->getManager();
$customer = $em->getRepository('myBundle:Customer')->find($id);

$cart = $customer->getCart();
```

Wczytywanie połączonych encji

Wczytanie połączonej encji jest niezwykle proste. Wystarczy odwołać się do atrybutu, który oznaczyliśmy relacją, i otrzymujemy podłączoną encję (lub tablice encji).

```
$em = $this->getDoctrine()->getManager();
$customer = $em->getRepository('myBundle:Customer')->find($id);

$cart = $customer->getCart();
```

Mamy encje koszyka

Nastawianie relacji

Aby ustawić relację pomiędzy encjami, wystarczy wstawić encję odpowiedniej klasy do atrybutu i zapamiętać stan bazy danych.

```
$customer->setCart($cart);
/* ... */
$em = $this->getDoctrine()->getManager();
$em->persist($cart);
$em->persist($customer);
$em->flush();
```

Zadania

Zaawansowane tematy

Doctrine Query Language (DQL)

- Jeśli chcemy znaleźć naszą encję na podstawie bardziej rozbudowanego zapytania, to możemy skorzystać z DQL.
- DQL jest językiem zapytań podobnym do SQL.

- W DQL nie myślimy w kategoriach bazy danych i jej tabel, a w kategoriach obiektów i ich klas.
- Dla osób znających SQL język DQL powinien być od razu zrozumiały.

Doctrine Query Language

Zamiast kolumn wybieramy cały obiekt,

a zamiast tabeli - przeszukujemy naszą klasę.

```
#SQL:
SELECT * FROM Posts WHERE raiting > 5;
#DQL:
SELECT post FROM myBundle:Post post WHERE post.raiting > 5
```

```
#SQL:
SELECT * FROM products WHERE price > 300.00 ORDER BY price DESC;
#DQL:
SELECT p FROM myBundle:Product p WHERE p.price > 300.00 ORDER BY p.price DESC
```

Przygotowywanie zapytań DQL

- Zapytania przygotowujemy, używając obiektu **`EntityManagera`** i jego metody **`createQuery()`**.
- Tworzymy wtedy obiekt zapytania.

```
$em = $this->getDoctrine()->getManager();

$query = $em->createQuery(
    'SELECT p
    FROM myBundle:Product p
    WHERE p.price > 300.00
    ORDER BY p.price DESC'
);
```

Przygotowywanie zapytań DQL

Jeżeli chcemy dynamicznie nastawiać wartość, według której będziemy wyszukiwać, możemy przekazać do zapytania zmienną.

```
$em = $this->getDoctrine()->getManager();

$query = $em->createQuery(
    'SELECT p
    FROM myBundle:Product p
    WHERE p.price > :priceToLook
    ORDER BY p.price DESC'
)->setParameter('priceToLook', $somePrice);
```

Wywoływanie zapytań DQL

Jeśli przygotowaliśmy zapytanie, to możemy go użyć.

Metoda ta zwraca nam tabelkę z wynikami (może być pusta!).

W przypadku gdy chcemy usunąć encję z użyciem DQL użyjemy metody, która zwróci odpowiedź **`true / false`**.

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

```
$products = $query->execute(['priceToLook', $somePrice]);
```

Wywoływanie zapytań DQL

Jeśli przygotowaliśmy zapytanie, to możemy go użyć.

Metoda ta zwraca nam tabelkę z wynikami (może być pusta!).

W przypadku gdy chcemy usunąć encję z użyciem DQL użyjemy metody, która zwróci odpowiedź **`true / false`**.

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

```
$products = $query->execute(['priceToLook', $somePrice]);
```

Metody **`getResult()`** używamy, gdy nie przekazujemy zmiennych do zapytania

Wywoływanie zapytań DQL

Jeśli przygotowaliśmy zapytanie, to możemy go użyć.

Metoda ta zwraca nam tabelkę z wynikami (może być pusta!).

W przypadku gdy chcemy usunąć encję z użyciem DQL użyjemy metody, która zwróci odpowiedź **`true / false`**.

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

```
$products = $query->execute(['priceToLook', $somePrice]);
```

Metody **`execute()`** używamy, gdy przekazujemy zmienne do zapytania

Określanie limitu zwracanych danych

Jeżeli chcemy nastawić limit na liczbę zwracanych danych, to możemy użyć metody **`setMaxResults(n)`** na naszym zapytaniu:

```
$products = $query->setMaxResults(20)->getResult();
```

Zwracanie tylko jednego wyniku

Jeżeli chcemy, żeby zapytanie zwróciło nam tylko jeden wynik zamiast tablicy wyników, powinniśmy wykonać dwa kroki:

- nastawić limit na 1,
- zamiast **`getResult()`** użyć **`getOneOrNullResult()`**

```
$product = $query->setMaxResults(1)->getOneOrNullResult();
```

Zwracanie tylko jednego wyniku

Jeżeli chcemy, żeby zapytanie zwróciło nam tylko jeden wynik zamiast tablicy wyników, powinniśmy wykonać dwa kroki:

- nastawić limit na 1,
- zamiast **`getResult()`** użyć **`getOneOrNullResult()`**

```
$product = $query->setMaxResults(1)->getOneOrNullResult();
```

Uwaga – może zwracać **`null`** (jak sama nazwa wskazuje).

Zwracanie wyników

Jeżeli chcemy otrzymać wyniki:

np. pomiędzy 20, a 30 (przydatne przy paginacji) możemy użyć klauzuli **BETWEEN** ... **AND** ...

```
$query = $em->createQuery(
    'SELECT u
    FROM myBundle:User u
    BETWEEN :start AND :stop'
);

$query->setParameter("start", 20);
$query->setParameter("stop", 30);
$users= $query->getResult();
```

Repozytoria

- Wcześniej już wspominaliśmy, że repozytoria to pomocnicze klasy, które służą do wyszukiwania encji w bazie danych.
- Do tej pory korzystaliśmy z podstawowego repozytorium.
- Teraz będziemy korzystać z repozytorium stworzonego przez nas.

Klasa repozytorium

Nasza klasa powinna spełniać następujące warunki:

- znajdować się w tym samym namespace co nasza encja,
- nazywać się tak samo jak encja z dodatkiem **`Repository`**.
- repozytoria muszą dziedziczyć z klasy **`EntityRepository`**.
- powinny znajdować się w tym samym katalogu co encja

- wyjątkiem są repozytoria wygenerowane automatycznie z użyciem poleceń konsolowych

Repozytoria

Przykładowa klasa repozytorium

```
namespace AppBundle\Entity;
use Doctrine\ORM\EntityRepository;

class PostRepository extends EntityRepository {
    /* ... */
}
```

Encje z ich repozytoriami wiążemy przez adnotacje:

```
/**
 *  @ORM\Entity(repositoryClass="AppBundle\Entity\PostRepository")
 */
class Product{
    /* ... */
}
```

Repozytoria

- Repozytorium najłatwiej stworzyć przez automatyczne wygenerowanie komendą konsolową **`doctrine:generate:entities`**
- Komenda ta również automatycznie generuje nam repozytorium, które znajduje się w katalogu Repository
- W klasie naszego repozytorium tworzymy metody, dzięki którym możemy wczytywać encje, używając bardziej skomplikowanych zapytań.
- Później będziemy mogli z tych metod korzystać w naszym kontrolerze tak samo, jak z podstawowych metod repozytorium.

Repozytoria

Klasa repozytorium

```
public function findOrderedByRaiting($start, $stop){
    $em = $this->getDoctrine()->getEntityManager();
    $posts = $em->createQuery(
            'SELECT p FROM MyBundle:Post p
            ORDER BY p.raiting DESC
            BETWEEN :start AND :stop')
    ->setParameter("start", $start)
    ->setParameter("stop", $stop)
    ->getResult();

    return $posts;
}
```

Kontroler

```
$em = $this->getDoctrine()->getManager();
$posts = $em->getRepository('MyBundle:Post')->findOrderedByRaiting(40, 50);
```

Repozytoria

Klasa repozytorium

```
public function findOrderedByRaiting($start, $stop){
    $em = $this->getDoctrine()->getEntityManager();
    $posts = $em->createQuery(
            'SELECT p FROM MyBundle:Post p
            ORDER BY p.raiting DESC
            BETWEEN :start AND :stop')
    ->setParameter("start", $start)
    ->setParameter("stop", $stop)
    ->getResult();

    return $posts;
}
```

Kontroler

```
$em = $this->getDoctrine()->getManager();
$posts = $em->getRepository('MyBundle:Post')->findOrderedByRaiting(40, 50);
```

Zastosowanie metody z repozytorium pozwala tworzyć nam swego rodzaju "aliasy" do bardziej skomplikowanych zapytań

Co warto poznać po kursie?

Oto istotne tematy dotyczące Doctrine i encji, które dobrze jest opanować.

- callbacki encji,
- wiele innych połączeń w Doctrine,
- używanie JOIN w DQL.

O wszystkim możecie doczytać w odpowiedniej dokumentacji:
http://doctrine-orm.readthedocs.org

Zadania