

# Aprenda a construir um CRUD simples, fácil e rápido com Doctrine - Diego Brocanelli

15-21 minutos

---

Olá, tudo bem?!

No ecossistema PHP temos disponíveis diversos projetos super bacanas, e para trabalhar na camada de banco de dados temos a nossa disposição o [Doctrine](#), o mais popular projeto voltado para Database Storage e Object Mapping.

O que você pode estar pensando é “Meu Framework X já tem implementações para isso”, sim de fato todos os grandes frameworks de mercado tem disponível componentes para abstração de banco de dados, porem uma das grandes vantagens do Doctrine é justamente ele poder ser utilizado em praticamente qualquer framework ou estrutura de projeto.

A partir do momento que você estrutura sua aplicação para utilizar o Doctrine nada o impedirá que no futuro você migre toda a lógica para outro projeto, essa facilidade vai lhe poupar muitas dores de cabeça quando chegar o momento de refatorar sua aplicação.

Sem mais delongas, vamos analisar a estrutura que será utilizada neste post, abaixo segue estrutura de diretórios e arquivo.

- exemplo-doctrine
  - config
    - Será armazenado toda a lógica de configuração da aplicação.
  - db
    - Responsável por armazenar o banco de dados (SQLite) e o dump que será gerado.
  - src
    - Responsável por conter toda a lógica da nossa aplicação.
  - composer.json
    - Configurações do composer para o projeto.

Conforme apresentado acima, crie a estrutura de diretórios necessários para a execução deste post.

## Instalando as dependências necessárias

Abra o arquivo *composer.json* e insira o seguinte código:

```
{  
  
  "require": {  
  
    "doctrine/orm": "^2.5",
```

```

        "doctrine/dbal": "^2.5",

        "symfony/yaml": "^3.2"

    },

    "autoload": {

        "psr-4": {

            "DiegoBrocanelli\\": "src/"

        }

    }

}

```

Como podemos observar, estamos utilizando os componentes do Doctrine e uma dependência do Symfony. Após a inserção das configurações, acesse a raiz do projeto e execute o seguinte comando pelo terminal:

Aguarde o processo de instalação que pode levar alguns minutos dependendo da sua conexão de internet.

## Configurando o projeto

Agora devemos criar nossas configurações necessárias para utilização do componente, acesse o diretório *'exemplo-doctrine/config'* e crie o arquivo *'bootstrap.php'*. Neste arquivo iremos centralizar as configurações necessárias para execução do Doctrine, segue código abaixo:

```

1  <?php

2  // Importação do autoload do composer

3  require_once __DIR__.'../vendor/autoload.php';

4  // Importação dos pacotes necessários

5  use Doctrine\ORM\Tools\Setup;

6  use Doctrine\ORM\EntityManager;

7  // Criar uma configuração ORM do Doctrine simples "default" para utilizar Annotations

8  $isDevMode = true;

9  $configuration = Setup::createAnnotationMetadataConfiguration(

```

```
10
11
12
13     $isDevMode
14 );
15 // Configurações do banco de dados
16 // Estamos utilizando o SQLite, para facilitar a reprodução do post
17 $connection = [
18     'driver' => 'pdo_sqlite', // Vamos utilizar o drive pdo do sqlite
19     'path' => __DIR__.'../db/db.sqlite' // caminho onde será armazenado o DB.
20 ];
21 // Obtemos o Entity Manager
22 $entityManager = EntityManager::create($connection, $configuration);
23
24
25
```

**Observação:** O Doctrine suporta configurações por *Annotation*, *XML* ou *YAML*, porem na minha opinião as *annotations* é a forma mais clara e fácil de definir sua entidade, gerando o menor número de arquivos possível.

Após a criação das configurações, devemos criar um arquivo que nos auxiliará nas execuções dos comandos pelo terminal. Crie o arquivo *'cli-config.php'* no diretório *'exemplo-doctrine/config'*, segue código abaixo:

```
<?php

// Importação do autoloader do composer

require __DIR__.'bootstrap.php';

// Retorna o componente que nos auxilia na utilização do Schema tool

// Necessário para gerar Tabelas para trabalhar com metadados
```

```
return \Doctrine\ORM\Tools\Console\ConsoleRunner::createHelperSet($entityManager);
```

## Criação da entidade 'Products'

Com as devidas configurações realizadas, o próximo passo é criar a entidade que representara nossa tabela do banco de dados, para isso crie a classe '*Products.php*' dentro do diretório '*exemplo-doctrine/src*', segue código abaixo:

```
1  <?php
2  namespace DiegoBrocanelli;
3  /**
4   * @Entity @Table(name="products")
5   */
6  class Product
7  {
8      /**
9       * @Id @Column(type="integer") @GeneratedValue
10      */
11      protected $id;
12      /**
13       * @Column(type="string")
14       */
15      protected $name;
16      /**
17       * @Column(type="datetime")
18       */
19      protected $created;
20      public function getId()
21      {
```

```
22     return $this->id;
23 }
24 public function getName()
25 {
26     return $this->name;
27 }
28 public function setName($name)
29 {
30     $this->name = $name;
31 }
32 public function setCreated(\DateTime $created)
33 {
34     $this->created = $created;
35 }
36 public function getCreated()
37 {
38     return $this->created;
39 }
40 }
41
42
43
44
45
46
```

47

48

49

Como podemos observar, o código acima é bem simples contendo apenas atributos e métodos *getters* e *setters*, atentando ao fato de não termos criado *PHPDoc*, para que assim foquemos apenas nos conceitos das *annontations* minimizando conflito de entendimento. É por meio dessas *annontations* que iremos ‘informar’ ao *doctrine* nossa entidade e suas especificações, segue abaixo maiores detalhes sobre as *annontations* utilizadas:

- **Class Products**

- `@Entity`
  - Responsável por informar ao Doctrine que esta nossa classe é nossa entidade.
- `@Table(name="products")`
  - Informamos que nossa entidade representa a tabela ‘*products*’ contida no banco de dados.

- **Column id**

- `@Id`
  - Informa que essa coluna é *primary key*.
- `@Column(type="integer")`
  - Define o tipo “inteiro” para a coluna.
- `@GeneratedValue`
  - Informa o desejo de *auto increment* para os valores dessa coluna.

- **Column name**

- `@Column(type="string")`
  - Define o tipo ‘string’ para a coluna.

- **Column created**

- `@Column(type="datetime")`
  - Define o tipo *Datetime* para a coluna.
  - Observação: Assim que implementarmos a pesquisa ficará mais claro

a estrutura do objeto gerado pelo *Doctrine*



Após nossa entidade devidamente criada, podemos dar início a implementação dos nosso recursos.

## Criando nosso banco de dados

Após a criação da entidade devemos realizar a criação do banco de dados, acesse a raiz do projeto e execute o seguinte comando:

```
vendor\bin\doctrine orm:schema-tool:create
```

- **vendor\bin\doctrine**

- Esta parcela do comando referencia a utilização do *Doctrine* que instalamos com o [\*Composer\*](#).
- **orm:schema-tool:create**
  - Comando para criação do banco de dados, lembrando que o arquivo será armazenado em *'exemplo-doctrine/db/db.sqlite'*.

Após execução do comando o retorno deve ser semelhante ao apresentado abaixo:

ATTENTION: This operation should not be executed in a production environment.

Creating database schema...

Database schema created successfully!

Como o próprio retorno nos informa, nunca devemos utilizar esta ação em produção, apenas em ambiente de desenvolvimento!

## **Inserindo registros no banco de dados**

Para a implementação do recurso de *insert*, crie a classe *'CreateProduct.php'* dentro do diretório *'exemplo-doctrine/src'*, segue código abaixo:

```
1  <?php
2  // Importamos o autoload do composer
3  require_once __DIR__.'../config/bootstrap.php';
4  // Como vamos trabalhar com envio de dados passados pelo terminal
5  // apenas garanto que haja valor para que seja processado a ação.
6  if(isset($argv)){
7      // coletamos o nome passado pelo usuário no terminal
8      $newProductName = $argv[1];
9      // Instanciamos nossa entidade Products
10     $product = new DiegoBrocanelli\Product();
11     // Passamos o novo nome para a entidade
12     $product->setName($newProductName);
13     // Passamos a data de criação para a entidade
14     $product->setCreated(new \DateTime(date('Y-m-d H:i:s')));
15     // Persistimos seus dados
16     $entityManager->persist($product);
17     // Descarregamos a ação
18     $entityManager->flush();
19     // Para melhor visualização do resultado, retornamos uma mensagem
20     // com o id do registro salvo no DB.
21     echo 'Created Product with ID '.$product->getId()."\n";
22 }
23
24
25
```



26

27

28

29

Para realizarmos a inserção de dados, execute o seguinte comando pelo terminal na raiz do projeto:

```
php src\CreateProduct.php produto_1
```

Resultado:

Created Product with ID 1

Como podemos observar, com poucas linhas de código criamos um recurso simples para inserção

de registro no banco de dados



## **Pesquisando todos os registros cadastrados**

Para a implementação do recurso de *select*, crie a classe '*ListProduct.php*' dentro do diretório 'exemplo-doctrine/src', segue código abaixo:

```

1
2
3 <?php
4 // Importamos o autoloader do composer
5 require_once __DIR__.'../config/bootstrap.php';
6 // Como vamos trabalhar com envio de dados passados pelo terminal
7 // apenas garanto que haja valor para que seja processado a ação.
8 if(isset($entityManager)){
9     //Importamos o repository que nos auxiliará com a pesquisa.
10    $productRepository = $entityManager->getRepository('DiegoBrocanelli\Product');
11    // Podemos acessar o método findAll() responsável por retornar todos os
12    // registros cadastrados em nossa tabela products
13    $products = $productRepository->findAll();
14    // Realizamos uma iteração de dados
15    foreach ($products as $product) {
16        // Exibimos o resultado de cada registro encontrado
17        var_dump($product);
18    }
19 }
20

```

Exemplo de retorno de pesquisa:

```

class DiegoBrocanelli\Product#64 (3) {
    protected $id =>
    int(1)
    protected $name =>

```

```

string(8) "produto_1"

protected $created =>

class DateTime#61 (3) {

    public $date =>

    string(26) "2017-05-16 23:53:49.000000"

    public $timezone_type =>

    int(3)

    public $timezone =>

    string(17) "America/Sao_Paulo"

}

}

```

Como podemos observar, com apenas a chamada do método *findAll()* obtemos todos os registros inseridos em nossa tabela no banco de dados, o atributo *'created'* é um objeto do tipo *DateTime* conforme configurado na entidade.

## Pesquisando um produto em específico

Para a implementação do recurso que irá pesquisar um produto em específico crie a classe *'ShowProduct.php'* dentro do diretório *'exemplo-doctrine/src'*, segue código abaixo:

```

1  <?php

2  // Importamos o autoloader do composer

3  require_once __DIR__.'../config/bootstrap.php';

4  // Como vamos trabalhar com envio de dados passados pelo terminal

5  // apenas garanto que haja valor para que seja processado a ação.

6  if(isset($argv)){

7      // Recebemos o id informado

8      $id = (int)$argv[1];

9      // Nosso entity manager nos fornece acesso ao método find()

```

```
10
11
12
13 $product = $entityManager->find('DiegoBrocanelli\Product', $id);
14 // Caso não seja encontrado nenhum produto com o id desejado
15 // será retornado mensagem informando o usuário
16 if($product === null){
17     echo "No product found. \n";
18     exit(1);
19 }
20 // Caso o produto seja encontrado exibimos seu nome e data de criação
21 echo sprintf('-%\s\n', $product->getName() . ' - ' . $product->getCreated());
22 }
23
24
25
```

Como podemos observar, com o método *find()* podemos informar nossa entidade e o id do registro ao qual desejamos encontrar na tabela.

## **Atualizar produtos**

Para a implementação do recurso de *update* crie a classe *'UpdateProduct.php'* dentro do diretório *'exemplo-doctrine/src'*, segue código abaixo:

```
1  <?php
2  // Importamos o autoload do composer
3  require_once __DIR__.'../config/bootstrap.php';
4  // Como vamos trabalhar com envio de dados passados pelo terminal
5  // apenas garanto que haja valor para que seja processado a ação.
6  if(isset($argv)){
7      // Recebemos o id do registro a ser atualizado
8      $id    = $argv[1];
9      // Recebemos o novo nome do produto
10     $newName = $argv[2];
11     // Pesquisamos para validar existência do produto no banco de dados
12     $product = $entityManager->find('DiegoBrocanelli\Product', $id);
13     // Caso produto não seja localizado, será retornado mensagem informando o usuário
14     if ($product === null) {
15         echo "Product $id does not exist.\n";
16         exit(1);
17     }
18     // Informamos o nome atual
19     echo 'Old name: ' . $product->getName() ."\n";
20     // Inserimos o novo nome desejado para o produto
21     $product->setName($newName);
22     // Executamos a ação de update
23     $entityManager->flush();
24     // Retornamos para o usuário o produto com seu novo nome
25     echo 'New name: ' . $product->getName() ."\n";
```

26

27

28

29

30

31 }

32

33

34

35

Como podemos observar, o processo de update ocorre sem mistérios de forma clara e fácil apenas manipulando o objeto e passando para que o *Doctrine* faça todo o trabalho.

## **Remover produtos**

Para a implementação do recurso de *delete* crie a classe '*DeleteProduct.php*' dentro do diretório 'exemplo-doctrine/src', segue código abaixo:

```
1  <?php
2  // Importamos o autoload do composer
3  require_once __DIR__.'../config/bootstrap.php';
4  // Como vamos trabalhar com envio de dados passados pelo terminal
5  // apenas garanto que haja valor para que seja processado a ação.
6  if(isset($entityManager)){
7      // Recebemos o id do registro a ser atualizado
8      $id = $argv[1];
9      // Pesquisamos para validar existência do produto no banco de dados
10     $product = $entityManager->find('DiegoBrocanelli\Product', $id);
11     // Caso produto não seja localizado, será retornado mensagem informando o usuário
12     if ($product === null) {
13         echo "Product $id does not exist.\n";
14         exit(1);
15     }
16     // Executamos a ação de remoção
17     $entityManager->remove($product);
18     // Efetiva a ação de remoção
19     $entityManager->flush();
20     // Retorna mensagem informativa de ação realizada com sucesso para o usuário
21     echo 'Product remove successfully!';
22 }
23
24
25
```

26

27

28

29

30

Como podemos observar, o processo de remoção é similar a todos os demais processo, o Doctrine tem métodos descritivos que facilitam sua utilização 😊

### Remover o banco de dados (Drop database)

No dia a dia temos a necessidade de apagar nosso banco de dados, para isso o *Doctrine* disponibiliza um recurso para nos auxiliar. Na raiz do nosso projeto execute o seguinte comando:

Execute este comando com parcimônia, após sua execução seu banco de dados será APAGADO!

```
// Comando para APAGAR o banco de dados
```

```
vendor\bin\doctrine orm:schema:drop --force
```

Após execução do comando será exibido uma mensagem de retorno semelhante a descrita abaixo:

```
Dropping database schema...
```

```
Database schema dropped successfully!
```

### Gerar dump da nossa base de dados

Um dos inúmeros recursos bacanas que o *Doctrine* dispõem é a criação de um arquivo *dump* da base de dados, arquivo que conterá toda estrutura da tabela *products*, para gerarmos nosso arquivo execute o seguinte comando na raiz do projeto:

```
// Comando para geração de arquivo de dump da base de dados
```

```
vendor\bin\doctrine orm:schema-tool:create --dump-sql > db/dump.sql
```

Após execução com sucesso do comando, será criado nosso arquivo no diretório '*exemplo-doctrine/db/dump.sql*', contendo a seguinte estrutura:

```
CREATE TABLE products (
```



```
id INTEGER NOT NULL,  
  
name VARCHAR(255) NOT NULL,  
  
created DATETIME NOT NULL, PRIMARY KEY(id)  
  
);
```

**Observação:** O retorno se dá em uma única linha, porém para que fique melhor sua visualização no post foi quebrado em linhas.

No decorrer deste post podemos observar o quão fácil é a utilização do *Doctrine*, infelizmente apenas abordamos uma pequena fração de todo o potencial e recursos que ele nos fornece. Caso tenha interesse em seguir com seus estudos indico que adquira o livro escrito pelo [Elton Minneto](#) “[Doctrine na prática](#)” antes que alguém pense o contrário não estou ganhando nenhum centavo para indicar o livro, estou fazendo pois é o melhor livro de *Doctrine* em PT-BR escrito por um profissional que respeito e admiro.

Espero que tenham apreciado o post, ficou um pouco maior do que eu desejava, porém para abordar todo o ciclo de utilização tentei ser o mais descritivo possível para auxiliar a máximo a compreensão das etapas, como dito anteriormente este post é introdutório onde existem uma infinidade de recursos não explorados que valem muito a pena serem estudados.

Dúvidas, sugestões, crítica ou elogios recomendo que deixe nos comentários para que assim possamos interagir e gerar mais conhecimento.

Até a próxima, abraços



# Instalando o Doctrine ORM - Como criar um CRUD com PHP Artigo

Conheça os cursos gratuitos do WebDevBr! -

- Pesquisa:

- 

- [compartilhar](#)

- [google+](#)

- 

---

Este artigo foi publicado a 6 anos, 1 mês, 2 semanas, 1 dia atrás.

---

ORM ou Object Relational Mapping é a design pattern maravilhosa para se trabalhar com bancos de dados relacionais como é o MySQL, por exemplo.

A real vantagem do Doctrine ORM é que ele cria uma cópia do seu banco de dados em forma de objetos, cada tabela tem a sua própria classe PHP e cada campo é uma variável. Esse conceito é chamado de entidade, ou seja, cada objeto (classe) é uma entidade do Doctrine ORM que pode ser usada para gerar o seu banco de dados completo, O Doctrine ORM dispensa o uso de classes para disponibilizar o banco de dados, ele é o model e o migration da sua aplicação.

Olhando por outro ângulo, cada entidade é também um registro no banco, cada vez que criamos um novo objeto e preenchemos seus atributos (variáveis) com dados estamos na verdade trabalhando com um registro no banco de dados, na sequência apenas dizemos ao Doctrine para realizar a ação final (rodar o comando SQL), esta tarefa é o flush.

## Gostou deste artigo?

Receba atualizações semanais com novos artigos do WebDevBr e outras dicas!

Essa integração com ORM é perfeita para quem trabalha com Orientação a Objetos, você trata o banco como objeto PHP, não precisa se preocupar com nenhuma linha de SQL e se você perder o caminho, um simples `var_dump` te ajuda, afinal é tudo PHP, né?

## Baixando o Doctrine ORM

De primeira já digo, vamos usar o Composer, então se você não sabe usar, visite este link primeiro.

A linha pra baixar o Doctrine ORM é:

```
"doctrine/orm": "2.4.*"
```

Vou também alterar o local dos comando de console (config) pra ficar fácil mais pra frente e setar o diretório do projeto (autoload), nosso composer.json completo fica assim:

```
{
    "require": {
        "doctrine/orm": "2.4.*"
    },
    "autoload": {
        "psr-0": {
            "WebDevBr": "src"
        }
    },
    "config": {
        "bin-dir": "./bin"
    }
}
```

Depois que você rodar o Composer e instalar tudo, precisamos configurar nossa conexão com o banco e o "console" (muito, muito, muito \* 3 útil).

## Conectando ao banco de dados

Pra conectar ao banco de dados precisamos passar dois parâmetros, um "isDevMode" que informa se estamos em modo de desenvolvimento ou não e um array com os dados de conexão. Também precisamos configurar o nosso entity manager, ou gerenciador de entidades, pra isso precisamos apenas dizer aonde nossas entidades estão.

Veja como fica nosso arquivo de configuração do Doctrine:

```
<?php
// bootstrap.php
require_once "vendor/autoload.php";

use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

// Diretório aonde vou guardar as entidades
$paths = array("./src/App/Entities");
$isDevMode = false;

// Dados da conexão
$dbParams = array(
    'driver' => 'pdo_mysql',
    'user' => 'root',
    'password' => '',
    'dbname' => 'foo',
);

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

Veja que temos a variável \$entityManager no final, ela vai ser responsável por fazer todos os tramites entre o PHP OOP e o MySQL, em outras palavras ela vai ser responsável por inserir, cadastrar, requisitar e deletar dados do banco. Vou nomear este arquivo de bootstrap.php

## Configurando o console

Agora que já temos nosso Doctrine configurado vou passar estes dados para o "console", pra isso criamos um novo arquivo chamado cli-config.php

```
<?php
use Doctrine\ORM\Tools\Console\ConsoleRunner;

// Aqui é carregado o arquivo que criamos antes
require_once 'bootstrap.php';

return ConsoleRunner::createHelperSet($entityManager);
```

## Facilitando ainda mais

Fácil né, são poucas linhas de código, mas mesmo assim ainda posso simplificar mais, então eu criei um package com estas configurações prontas e que já instala o Doctrine ORM também, assim fica mais fácil.

Pra instalar chame esta linha no require do Composer ao invés da do Doctrine:

```
"webdevbr/doctrine": "1.0.0"
```

E o composer.json ficaria:

```
{
    "require": {
        "webdevbr/doctrine": "1.0.0"
    },
    "autoload": {
        "psr-0": {
            "WebDevBr": "src"
        }
    },
    "config": {
        "bin-dir": "./bin"
    }
}
```

Pra usar é simples, o bootstrap.php ficaria assim:

```
<?php

require 'vendor/autoload.php';
$isDevMode = true;
$conn = array(
    'driver' => 'pdo_mysql',
    'user' => 'root',
    'password' => '',
    'dbname' => 'foo'
);
$doctrine = new WebDevBr\Doctrine\Doctrine($conn, $isDevMode);
$doctrine->setEntitiesDir('./src/App/Entities');
$entityManager = $doctrine->getEntityManager();
```

E agora você só configura o cli-config.php (se quiser copiar de vendor/WebDevBr/Doctrine/cli-config.php).

```
<?php

require_once 'bootstrap.php';
```

```
use Doctrine\ORM\Tools\Console\ConsoleRunner;  
return ConsoleRunner::createHelperSet($entityManager);
```

Very ease, né. No próximo artigo vamos criar uma entidade e criar a tabela no banco sem usar nenhuma outra ferramenta (de banco de dados), só o Doctrine ORM, você pode usar qualquer um dos dois métodos de instalação, o importante é você ter o Entity Manager disponível e o cli-config.php configurado corretamente.

Att. Erik

<https://webdevbr.com.br/instalando-o-doctrine-orm-como-criar-um-crud-com-php>