# **ESTUDO DE CASO**

O foco desse trabalho é desenvolver um framework que atenda as necessidades dos programadores e concilie com a facilidade de utilização.

## Tecnologias

Para o desenvolvimento do ORM a linguagem PHP foi escolhida. PHP é uma sigla em inglês que significa *Hypertext PreProcessor* (ou em português Pré-Processador de Hipertexto).

O PHP é uma linguagem de *script*, largamente utilizada, principalmente para desenvolvimento web. Seu uso é extremamente simples para iniciantes e oferece diversos recursos para desenvolvedores profissionais.

Outro motivo para o grande uso do PHP é o fato de ser *open source* e poder ser utilizado na maioria dos sistemas operacionais, incluindo Linux, variantes Unix, Mac OS e Microsoft Windows.

## Mapeamento

Para realizar o mapeamento dos modelos de dados, foi utilizado o conceito de *Annotations*.

As *annotations* são “etiquetas” que adicionam metadados relevantes sobre classes, métodos e propriedades. Ou seja, através do uso de *annotations*, pode-se adicionar às classes informações para mapear tabelas do banco de dados, e adicionar às propriedades da classe para mapear as colunas de uma tabela do banco de dados, para que posteriormente, em tempo de execução, os metadados indicados pelas *annotations* sejam analisados e a partir disso, o ORM irá trabalhar de acordo com essas informações.

Porém, no PHP, como em outras linguagens de programação como o Java, não existe uma estrutura nativa para adicionar esses metadados dentro do contexto da classe. Então, qual seria a solução para isso? Utilizar expressões regulares e a família de classes *Reflection* do PHP para resolver essa questão.

O PHP, assim como outras linguagens de programação possuem uma estrutura para documentar a classe, propriedades e métodos, essa estrutura é o bloco de comentário, no PHP é comumente conhecido como *PHPDoc*.

Para obter os comentários contidos nas classes e extrair as informações dos comentários, pode-se utilizar a família de classes *Reflection*, no caso, especificamente as classes *ReflectionClass* e *ReflectionProperty*.

Para os exemplos a seguir, considere a seguinte classe do modelos de dados:

1. namespace App;
2. /\*\*
3. \* @ORM/Entity
4. \* @ORM/Table(name=pessoa)
5. \*/
6. class Pessoa {
7. /\*\*
8. \* @ORM/Id
9. \* @ORM/Generated
10. \* @ORM/Column(name=pessoa\_id, type=int)
11. \*/
12. public $id;
13. /\*\*
14. \* @ORM/Column(name=nome, type=string, length=50)
15. \*/
16. public $nome;
17. /\*\*
18. \* @ORM/Column(name=data\_nasc, type=date)
19. \*/
20. public $dataNasc;
21. }

Listagem 1 – Classe Pessoa

Extrair o *PHPDoc* de uma classe do modelo é realizado de maneira simples através do uso da classe *ReflectionClass*. Como no exemplo abaixo:

1. <?php
2. $reflection = new ReflectionClass(‘App\Pessoa’);
3. $doc = $reflection->getDocComment();
4. ?>

Listagem 2 – Exemplo de uso da classe ReflectionClass

No exemplo mostrado na Listagem 2, na linha 2, é criado a instancia da classe *ReflectionClass*, passando como parâmetro para o construtor, o nome da classe do modelo. Já na linha 3, o método *getDocComment* retorna o bloco de comentário da classe informada no construtor.

Com a instância da classe *ReflectionClass*, é possível acessar as propriedades da classe do modelo. Por exemplo:

1. <?php
2. $reflection = new ReflectionClass(‘App\Pessoa’);
3. $doc = $reflection->getDocComment();
4. $properties = $reflection->getProperties();
5. foreach($properties as $property) {
6. $propertyDoc = $property->getDocComment();
7. }
8. ?>

Listagem 3 – Exemplo da classe ReflectionProperty

Na linha 4, o método *getProperties* retorna um *array* contento instancias da classe *ReflectionProperity*, uma para cada propriedade contendo da classe. Então, para obter os blocos de comentário das propriedades da classe, percorre-se a o *array* na linha 6 e na linha 7 o método *getDocComment()* retorna o bloco de comentário da propriedade atual.

Agora, com os blocos de comentário em mãos, o próximo passo é utilizar a expressões regulares para extrair as *annotations* e suas informações. Expressão Regular é um recurso muito comum em diversas linguagens. Para executar uma expressão regular no PHP, usa-se duas das diversas funções nativas disponíveis chamadas “preg\_match” (PHP, 2018) e “preg\_match\_all” (PHP, 2018). Ambas executam uma expressão regular contra um texto procurando por referências, a diferença entre as duas é que, a função “preg\_match” para quando encontra a primeira correspondência e a função “preg\_match\_all” procura no texto inteiro e retorna todas as correspondências.

Na Listagem 1, a classe “Pessoa” possui algumas *annotations*, entre elas a *annotation* “@ORM/Table”, que informa, entre outras coisas, o noma da tabela em questão que deve ser mapeada. Então para identificar essa *annotation* e suas propriedades, para então extrair suas informações, é possível da seguinte maneira:

1. <?php
2. $reflection = new ReflectionClass(‘App\Pessoa’);
3. $doc = $reflection->getDocComment();
4. $findAnnotatoins = ‘/@ORM\/[@A-Za-z0-9=,\_\/\s\(\)\{\}]+/i’;
5. $findTable = ‘/Table\([@A-Za-z0-9=,\_\/\s\(\)]+\)/i’;
6. $findName = ‘/name[\s]?=[\s]?(\w+)/i’;
8. preg\_match\_all($findAnnotatoins, $doc, $annotations);
9. $annotations = join($annotations[0], ‘’);
10. preg\_match($findTable, $annotations, $table);
11. preg\_match($findName, $table[0], $name);
12. ?>

Listagem 3 – Exemplo de Expressão Regular

As linhas 2 e 3 já foram mostradas anteriormente, mas na linha 4, está a declaração da primeira expressão regular que será usada na variável “$findAnnotations”. Para um melhor entendimento, a expressão regular pode ser quebrada em XX partes:

* /
* @ORM\/
* [@A-Za-z0-9=,\_\/\s\(\)\{\}]+
* /
* i

### **Mapeamento Simples**

Como mostrado na Listagem 1, pode-se ver um exemplo de um mapeamento simples da classe “Pessoa”:

*==============================*

Na linha 5, a *annotation* “@ORM\Entity” informa para o ORM que a classe pessoa deve ser tratada como uma “Entity” (em português, Entidade), ou seja, a classe “Pessoa” representa uma tabela com o nome de “pessoa” no banco de dados.

Na linha 9, a *annotation* “@ORM/Id” informa que a propriedade “$id” (declarada na linha 13) é a chave primária da tabela “pessoa”. Essa propriedade será utilizada pelo ORM quando for necessário fazer consultas relacionando múltiplas tabelas.

Na linha 10, a *annotation* “@ORM/Generated” informa que a propriedade “$id” é um valor de geração automática e sequencial. O ORM deverá resolver o valor desse campo de acordo com o *Driver* utilizado para a conexão com banco de dados utilizado. Os *Drivers* serão abordados mais profundamente futuramente.

Nas linhas 11, 16, 21, a *annotation* “@ORM/Column” informa respectivamente que as propriedades “$id”, “$nome”, “$dataNasc”, estão mapeando colunas específicas da tabela “pessoa”.

A *annotation* “@ORM/Column” possui propriedades, a propriedade “name” define o nome da coluna mapeada, como no exemplo acima, as propriedades da classe “Pessoa” representam as colunas da tabela “pessoa” com os nomes “pessoa\_id”, “nome”, “data\_nasc”.

A propriedade “type” define os tipos dos dados a serem mapeados, como no exemplo acima, as colunas mapeadas acima são respectivamente os tipos “int”, “string” e “date”.

A propriedade “length” define o tamanho do campo do tipo “string”, no caso da propriedade “$nome” mostrada no exemplo acima, a propriedade “length” define que o tamanho do campo é igual à 50. Essa informação auxilia o ORM a criar a tabela, caso esse comportamento seja definido. Esse assunto será abordado mais à frente.

Além das propriedades citadas acima, também existem as propriedades “scale”, “precision”, “unique”, “nullable”. Essas propriedades auxiliam o ORM na tarefa de criar a tabela. As propriedades “scale” e “precision” definem o tamanho e a precisão de uma coluna do tipo número real. A propriedade “unique” define que a coluna deve ser indexada como única na tabela, ou seja, não deve possuir valores que repitam na tabela. E por fim, a propriedade “nullable” define que a coluna não deve aceitar valores nulos.

Além dos tipos citados no exemplo acima, outros tipos podem ser utilizados, esses tipos são definidos no Driver utilizado para a conexão. Esse assunto será abordado mais a fundo na seção de *Drivers*.

### **Mapeamento um Relacionamento**

Na linha 10, a *annotation* “@ORM/Generated” informa que a propriedade “$id” é um valor de geração automática e sequencial. O ORM deverá resolver o valor desse campo de acordo com o *Driver* utilizado para a conexão com banco de dados utilizado. Os *Drivers* serão abordados mais profundamente futuramente.

#### **Um para Um**

Texto.

#### **Um para Muitos**

Texto.

#### **Muitos para Muitos**

Texto.

## Conexões

### **Driver**

### **Criação das tabelas**

**Glossário**

Script:

Open Source:

Reflection:

**Referências**

http://php.net/manual/en/function.preg-match-all.php

Usaremos em nossos exemplos a função **preg\_match\_all** que serve melhor para esse propósito.  
De acordo com o escrito na sua documentação, existente no site php.net sob o endereço <a href=””http://php.net/manual/en/function.preg-match-all.php, vemos o seguinte:  
**int preg\_match\_all ( string $pattern , string $subject [, array &$matches [, int $flags = PREG\_PATTERN\_ORDER [, int $offset = 0 ]]] )**, onde:  
$pattern: é a nossa expressão regular, ou seja, a string que simboliza o padrão a ser reconhecido  
$subject: é o texto a ser pesquisado em busca do padrão $pattern. No nosso caso será o comentário da classe ou do atributo;  
$matches: é uma variável do tipo array que receberá todas as ocorrências de $pattern em $subject.  
Se você usa linux pode baixar o kiki para realizar os testes de expressões regulares. Se não, pode usar sites do <http://regexpal.com/> ou <http://www.solmetra.com/scripts/regex/> (esse último mais alinhado à nossa função preg\_math\_all, pois você pode escolher usar a mesma função). O kiki é, para o gnome, a sua melhor opção, nem precisa estar conectado.  
Sabendo disso, vamos ao próximo passo, que é criar as expressões regulares para os nossos padrões.

#### Reconhecendo as suas anotações

Temos @table e @column, ambas tags que precisaríamos no nosso para mapear essa classe, e cada uma com uma sintaxe diferente. Vejamos cada uma separadamente até encontrar cada padrão.  
Para a nossa anotação de tabela, temos @table=tb\_pessoa, que tem o seu padrão representado por:  
@ + tag + = + valor.  
Para esse padrão a nossa expressão regular será a seguinte:  
**@[w]+[ ]{0,1}=[ ]{0,1}[w]+**, onde:  
**[w]+** é a string que dá nome à tag;  
**[ ]{0,1}** é um espaço que pode ou não existir antes do =;  
**=** separa os dois termos (tag e valor)  
**[ ]{0,1}** é um espaço que pode ou não existir depois do =;  
**[w]+** é a string que atribui valor à anotação;

Já para a tag de coluna nós temos um padrão um pouco diferente:  
@tag + ( + valor + ).  
E a expressão regular para isso é pode ser:  
**@[w]+[ ]{0,1}([w=, .-<>:]+**, onde:  
**[w]+** é a string que dá nome à tag;  
**[ ]{0,1}** é um espaço que pode ou não existir antes do =;  
**(** escapa a abertura parênteses  
**[ ]{0,1}** é um espaço que pode ou não existir depois do =;  
**[w]+** é a string que atribui valor à anotação;  
**(** escapa o fechamento parênteses.  
Para separar os valores dentro dos parênteses você pode usar expressões regulares, explode ou outra função que seja necessária. No momento vamos focar somente em encontrar as tags e descobrir seus valores de forma mais bruta.  
Para que usemos a expressão regular tanto para @table quanto para @column, vamos uni-las com o caracter “|” que concatena as duas formando um só $pattern.  
A implementação em PHP disso ficaria como no bloco abaixo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | &lt;?  $tags = array();  $pattern = '/@[w]+[ ]{0,1}=[ ]{0,1}[w]+|@[w]+[ ]{0,1}([w=, .-&lt;&gt;:]+/';  $subject = '/\*\*  \* Chave primária  \* @type= integer  \* @notnull = true  \* @size=255  \* @meta (atributo1=valor1, atributo2=valor2)  \* @meta1 (atributo3=valor3)  \*/';  preg\_match\_all( $pattern, $subject, $tags );  ?&gt; |