1. **Introdução**

Lamberjack’s ORM é um *framework* ORM para a linguagem PHP. ORM é uma sigla em inglês que significa *Object-Relational Mapper*. Um ORM é uma ferramenta bastante útil no dia-a-dia do desenvolvedor de software.

O Lamberjack’s ORM trabalha com mapeamento de tabelas em classes do modelo de dados utilizando *Annotations*. O ORM permite ao desenvolvedor trabalhar com mais de uma conexão de banco de dados e criar as tabelas do banco de dados baseado nas classes do modelo.

1. **Instalação**

Para usar o Lamberjack’s ORM, pode-se obtê-lo no repositório **orm** no GitHub no link: https://github.com/dfrancklin/orm.

Basta então copiar a pasta “/orm” que foi baixada e para incluir o ORM no projeto, use o comando “require\_once”:

1. <?php
2. require\_once ‘./orm/load.php’;
3. ?>

Fonte: Autor, 2018

1. **Definindo Conexões**

As conexões que serão utilizadas pelo ORM devem ser declaradas em um arquivo com extensão “.php”. Por padrão, o arquivo é esperado que esteja na pasta raiz do ORM com o nome “connection.config.php”, ou seja, supondo que o ORM esteja localizado “/home/user/app/orm/”, então o caminho para o arquivo seria “/home/user/app/orm/connection.config.php”.

O arquivo de conexões pode substituído da seguinte maneira:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnectionsFile(\_\_DIR\_\_.‘/db/connections.php’);
4. ?>

Fonte: Autor, 2018

O arquivo deve conter um *array* com uma ou mais conexões, onde a chave da conexão é o nome identificador da conexão e o valor é um *array* contendo as informações da conexão. As informações variam de acordo com o banco de dados a ser utilizado.

1. <?php
2. return [
3. ‘exemplo-mysql’ => [
4. ‘db’ => ‘mysql’,
5. ‘version’ => ‘5.7.11’,
6. ‘host’ => ‘localhost’,
7. ‘schema’ => ‘app’,
8. ‘user’ => ‘root’
9. ‘pass’ => ‘root’
10. ],
11. ‘exemplo-sqlite’ => [
12. ‘db’ => ‘sqlite’,
13. ‘version’ => ‘3’,
14. ‘file’ => ‘../data/app-storage.sq3’,
15. ],
16. ];
17. ?>

Fonte: Autor, 2018

Os valores para a conexão “exemplo-mysql” é um *array* contendo as chaves “db”, “version”, “host”, “schema”, “user” e “pass”. A chave “db” contém o banco de dados a ser utilizado. A chave “version” indica a versão do banco de dados utilizado, no qual o *Driver* deve corresponder à essa versão. A chave “host” é o endereço onde o banco de dados está localizado. A chave “schema” é o bando de dados (conjunto de tabelas) que será utilizado. As chaves “user” e “pass” são respectivamente o usuário e a senha de acesso ao banco de dados.

Os valores para a conexão “exemplo-sqlite” é um *array* contendo as chaves “db”, “version”, “file”. As chaves “db” e “version” funcionam da mesma maneira que a conexão anterior. A chave “file” indica o arquivo local o qual o banco de dados SQLite utilizará para armazenar os dados.

Para informar ao ORM qual (ou quais) conexão será utilizada na aplicação, deve ser feito através da classe principal do ORM conforme o exemplo a seguir:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $orm->addConnection(‘exemplo-sqlite’);
5. ?>

Fonte: Autor, 2018

O método *setConnection* adiciona a conexão à lista de conexões que o ORM pode utilizar e faz com que a conexão informada seja a conexão padrão para o ORM, ou seja, qualquer operação que será realizada pelo ORM, se não for informada uma conexão explicitamente, o ORM irá assumir que a conexão que precisa ser usada é a conexão padrão.

Já o método *addConnection* apenas adiciona a conexão à lista de conexões que o ORM pode utilizar. A conexão padrão pode ser substituída a qualquer momento, para isso basta utilizar o método *setDefaultConnection*:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->addConnection(‘exemplo-mysql’);
4. $orm->setDefaultConnection(‘exemplo-mysql’);
5. ?>

Fonte: Autor, 2018

* 1. **Criar Tabelas Automaticamente**

O ORM tem a habilidade de criar as tabelas a partir das classes modelo. Para que o ORM saiba como criar, é necessário informar o caminho para a pasta que contém os modelos e o *namespace* no momento em que estiver configurando a conexão no ORM. Por exemplo:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’, [
4. ‘namespace’ => ‘App\Models’,
5. ‘modelsFolder’ => ‘/home/user/app/models’,
6. ‘create’ => true
7. ]);
8. ?>

Fonte: Autor, 2018

Pode ser necessário também, apagar as tabelas antes de criá-las, para isso, basta informar também na configuração da conexão:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’, [
4. ‘namespace’ => ‘App\Models’,
5. ‘modelsFolder’ => ‘/home/user/app/models’,
6. ‘create’ => true,
7. ‘drop’ => true
8. ]);
9. ?>

Fonte: Autor, 2018

O ORM permite ainda, que uma ação seja executada antes de apagar as tabelas e uma ação após criar as tabelas. Essas ações podem ser úteis para criar uma rotina de *backup/restore* ou de migração de banco de dados. Para informar o ORM quais ações ele deve executar, basta fazer o seguinte:

1. <?php
2. $dbHelper = new App\Helpers\*InitDatabase()*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’, [
5. ‘namespace’ => ‘App’,
6. ‘modelsFolder’ => ‘/home/user/app/models’,
7. ‘drop’ => true,
8. ‘create’ => true,
9. ‘beforeDrop’ => [ $dbHelper, ‘beforeDrop’ ],
10. ‘afterCreate’ => [ $dbHelper, ‘afterCreate’ ]
11. ]);
12. ?>

Fonte: Autor, 2018

Na linha 2, é criado uma instância da classe “App\Helpers\InitDatabase” e nas linhas 9 e 10, é informado para a conexão respectivamente, quais métodos devem ser executados antes de apagar as tabelas e depois de criá-las. Utilizando esses métodos é possível que o desenvolvedor desenvolva uma lógica de como realizar o backup das informações essenciais do banco de dados antes de apagar as tabelas e posteriormente restaurar essas informações após a criação das tabelas.

Os valores esperados pelas chaves “beforeDrop” e “afterCreate” podem ser também uma função anônima:

1. <?php
2. …
3. ‘beforeDrop’ => function($entityManager) { … },
4. ‘afterCreate’ => function($entityManager) { … }
5. …
6. ?>

Fonte: Autor, 2018

Ou uma string contendo o nome de uma função:

1. <?php
2. …
3. ‘beforeDrop’ => ‘beforeDrop’,
4. ‘afterCreate’ => ‘afterCreate’
5. …
6. function beforeDrop($entityManager) { … }
7. function afterCreate($entityManager) { … }
8. ?>

Fonte: Autor, 2018

O ORM uma instância de um *EntityManager* por parâmetro para os métodos ou funções que irão ser executados antes e depois do processo de criação das tabelas. Ele pode ser usado para realizar ações no banco de dados. O *EntityManager* será abordado mais a frente.

1. **Definindo Modelos**

Um modelo é uma classe que representa uma tabela no banco de dados e pode ser mapeada da classe para a tabela e da tabela para a classe em operações de consulta, inserção, alteração e deleção.

Para que um modelo possa representar devidamente uma tabela no banco de dados dentro do ORM, ela deve ser “anotada” utilizando o padrão de *annotation* definido pelo ORM.

* 1. ***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.

Abaixo, uma lista completa de as *annotations* e suas propriedades:

* + 1. ***Annotations* de Classes**
* ***Annotation*:** @ORM/Entity

**Descrição:** Define que a classe deve ser considerada como uma tabela no banco de dados.

**Preenchimento:** Obrigatório. O não preenchimento resulta em erro.

* ***Annotation*:** @ORM/Table

**Descrição:** Define informações sobre a tabela mapeada.

**Preenchimento:** Opcional.

**Propriedades:**

* + **Nome:** name.

**Descrição:** Define o nome da tabela mapeada.

**Preenchimento:** Opcional. Caso não preenchido, o ORM assume que o nome da tabela é o mesmo que o nome da classe.

* + **Nome:** schema

**Descrição:** Define qual é o conjunto de tabelas ou banco de dados ao qual a tabela mapeada em questão existe.

**Preenchimento:** Opcional. Caso não preenchido, o ORM assume que o schema a ser usado é o padrão definido na conexão ou nenhum, dependendo do banco de dados usado.

* + **Nome:** mutable

**Descrição:** Caso o valor seja “true” define que a tabela não pode ser modificada pelas operações de inserção, alteração e deleção. O valor padrão é “false”.

**Preenchimento:** Opcional. Assume o valor padrão caso não seja preenchido.

* + 1. ***Annotations* de Propriedades**
* ***Annotation*:** @ORM/Id

**Descrição:** Define que a propriedade representa a chave primária da tabela mapeada.

**Preenchimento:** Obrigatório. O não preenchimento resulta em erro.

* ***Annotation*:** @ORM/Generated

**Descrição:** Define que o valor da chave primaria é auto gerado, seja através de *sequence* ou qualquer tipo de *autoincrement* (isso é definido no driver para cada banco de dados).

**Preenchimento:** Opcional. O não preenchimento indica que o preenchimento e o incremento deverão ser feitos manualmente.

* ***Annotation*:** @ORM/Column

**Descrição:** Define informações sobre a coluna ser mapeada.

**Preenchimento:** Opcional. Assume os valores padrões das propriedades listadas a seguir.

**Propriedades:**

* + **Nome:** name.

**Descrição:** Define o nome da coluna a ser mapeada

**Preenchimento:** Opcional. Caso não seja preenchido, o ORM assume que o nome da coluna é o mesmo nome do atributo.

* + **Nome:** type.

**Descrição:** Define o tipo da coluna a ser mapeada

**Tipos:** string, int, float, lob (large object), date, time, datetime, bool

**Preenchimento:** Opcional. Caso não seja preenchido, o ORM assume que o tipo da coluna é string.

* + **Nome:** length.

**Descrição:** Define o tamanho da coluna a ser mapeada quando a coluna é do tipo “string”

**Preenchimento:** Opcional. Caso não seja preenchido, o ORM assume que o tamanho da coluna é 255.

* + **Nome:** scale.

**Descrição:** Define o tamanho da coluna a ser mapeada quando a coluna é do tipo “float”

**Preenchimento:** Opcional. Caso não seja preenchido, o ORM assume que o tamanho da coluna é 14.

* + **Nome:** precision.

**Descrição:** Define a precisão da coluna (quantidade de dígitos após a virgula) a ser mapeada quando a coluna é do tipo “float”

**Preenchimento:** Opcional. Caso não seja preenchido, o ORM assume que a precisão da coluna é 2.

* + **Nome:** unique.

**Descrição:** Se o valor do campo for “true”, define que o campo deve ser conter um valor único.

**Preenchimento:** Opcional. Valor padrão é “false”.

* + **Nome:** nullable.

**Descrição:** Se o valor do campo for “false”, define que o campo não pode receber valores nulos.

**Preenchimento:** Opcional. Valor padrão é “true”.

* + 1. ***Annotations* de Propriedades**
* ***Annotation*:** @ORM/HasOne

**Descrição:** Define um relacionamento do tipo “um para um”. É necessário que a classe de referência tem um atributo equivalente à outra ponta do relacionamento com a *annotation* “@ORM/BelongsTo”.

**Preenchimento:** Opcional.

**Propriedades:**

* + **Nome:** class.

**Descrição:** Define qual classe deve ser referenciada no mapeamento.

**Preenchimento:** Obrigatório. O não preenchimento resulta em erro.

* + **Nome:** cascade.

**Descrição:** Define que as operações de inserção, alteração e deleção pode acontecer em cascata, ou seja, a operação realizada na classe que mapeia essa *annotation*, deve ser estendida para a classe referenciada.

**Valores:** INSERT, UPDATE, DELETE, ALL.

**Preenchimento:** Opcional. Caso não seja preenchida, a operação não é estendida.

* ***Annotation*:** @ORM/HasMany

**Descrição:** Define um relacionamento do tipo “um para muitos”. É necessário que a classe de referência tem um atributo equivalente à outra ponta do relacionamento com a *annotation* “@ORM/BelongsTo”.

**Preenchimento:** Opcional.

**Propriedades:**

* + **Nome:** class.

**Descrição:** Define qual classe deve ser referenciada no mapeamento.

**Preenchimento:** Obrigatório. O não preenchimento resulta em erro.

* + **Nome:** cascade.

**Descrição:** Define que as operações de inserção, alteração e deleção pode acontecer em cascata, ou seja, a operação realizada na classe que mapeia essa *annotation*, deve ser estendida para a classe referenciada.

**Valores:** INSERT, UPDATE, DELETE, ALL.

**Preenchimento:** Opcional. Caso não seja preenchida, a operação não é estendida.

* ***Annotation*:** @ORM/BelongsTo

**Descrição:** Define a outra ponta dos relacionamentos do tipo “um para um” e do tipo “um para muitos”, ou seja, define a chave estrangeira para o relacionamento. É necessário que a classe de referência tem um atributo equivalente à outra ponta do relacionamento com a *annotation* “@ORM/HasOne” ou “@ORM/HasMany”.

**Preenchimento:** Caso uma relação do tipo “um para um” ou do tipo “um para muitos” seja definida, é obrigatório que a classe referenciada possua essa *annotation*.

**Propriedades:**

* + **Nome:** class.

**Descrição:** Define qual classe deve ser referenciada no mapeamento.

**Preenchimento:** Obrigatório. O Não preenchimento resulta em erro.

* + **Nome:** cascade.

**Descrição:** Define que as operações de inserção, alteração e deleção pode acontecer em cascata, ou seja, a operação realizada na classe que mapeia essa *annotation*, deve ser estendida para a classe referenciada.

**Valores:** INSERT, UPDATE, DELETE, ALL.

**Preenchimento:** Opcional. Caso não seja preenchida, a operação não é estendida.

* + **Nome:** optional.

**Descrição:** Define se o relacionamento é opcional, ou seja, indica que o valor pode ou não ser nulo.

**Preenchimento:** Opcional. Valor padrão é “false”

* ***Annotation*:** @ORM/JoinColumn

**Descrição:** Define as informações da coluna que deve ser a chave estrangeira. Somente a propriedade que possui a *annotation* “@ORM/BelongsTo” deve possuir essa *annotation* para complementar as informações.

**Preenchimento:** Opcional. Assume os valores padrões das propriedades listadas a seguir.

**Propriedades:**

* + **Nome:** name.

**Descrição:** Define o nome da coluna a ser mapeada como chave estrangeira.

**Preenchimento:** Opcional. O valor padrão é o nome da propriedade mais o sufixo “\_id”, por exemplo, “pessoa\_id”.

* ***Annotation*:** @ORM/ManyToMany

**Descrição:** Define um relacionamento de do tipo “muitos para muitos”. É necessário que a classe de referência tem um atributo equivalente à outra ponta do relacionamento com a *annotation* “@ORM/ManyToMany”.

**Preenchimento:** Caso uma relação do tipo “muitos para muitos” seja definida, é obrigatório que a classe referenciada possua essa *annotation*.

**Propriedades:**

* + **Nome:** class.

**Descrição:** Define qual classe deve ser referenciada no mapeamento.

**Preenchimento:** Obrigatório. O não preenchimento resulta em erro.

* + **Nome:** cascade.

**Descrição:** Define que as operações de inserção, alteração e deleção pode acontecer em cascata, ou seja, a operação realizada na classe que mapeia essa *annotation*, deve ser estendida para a classe referenciada.

**Valores:** INSERT, UPDATE, DELETE, ALL.

**Preenchimento:** Opcional. Caso não seja preenchida, a operação não é estendida.

* + **Nome:** mappedBy.

**Descrição:** Define que o lado principal do mapeamento é a classe referenciada e define também qual é o atributo ao qual é o equivalente. O lado principal pode definir também as informações da tabela de ligação.

**Preenchimento:** Opcional.

* ***Annotation*:** @ORM/JoinTable

**Descrição:** Define informações para a tabela de ligação. Somente o lado principal do relacionamento deve possuir essa *annotation* para complementar as informações.

**Preenchimento:** Opcional. Assume os valores padrões das propriedades listadas a seguir.

**Propriedades:**

* + **Nome:** tableName

**Descrição:** Define o nome da tabela de ligação.

**Preenchimento:** Opcional. O valor padrão e composto pelo nome das duas tabelas que compõe o relacionamento, por exemplo, “empregado\_role”.

* + **Nome:** schema

**Descrição:** Define qual é o conjunto de tabelas ou banco de dados ao qual a tabela de ligação em questão existe.

**Preenchimento:** Opcional. Caso não preenchido, o ORM assume que o schema a ser usado é o padrão definido na conexão ou nenhum, dependendo do banco de dados usado.

* + **Nome:** join

**Descrição:** Define o nome da coluna que é a chave estrangeira que aponta para o lado principal do relacionamento.

**Preenchimento:** Opcional. Assume os valores padrões das propriedades listadas a seguir.

**Propriedades:**

* + - **Nome:** name

**Descrição:** Define o nome da coluna a ser mapeada como chave estrangeira.

**Preenchimento:** Opcional. O valor padrão é o nome da propriedade mais o sufixo “\_id”, por exemplo, “empregado\_id”.

* + **Nome:** inverse

**Descrição:** Define o nome da coluna que é a chave estrangeira que aponta para o lado secundário do relacionamento.

**Preenchimento:** Opcional. Assume os valores padrões das propriedades listadas a seguir.

**Propriedades:**

* + - **Nome:** name

**Descrição:** Define o nome da coluna a ser mapeada como chave estrangeira.

**Preenchimento:** Opcional. O valor padrão é o nome da propriedade mais o sufixo “\_id”, por exemplo, “role\_id”.

* 1. **Exemplo de Mapeamento Simples**

Um exemplo básico de como criar uma classe do modelo, pode ser encontrado no exemplo a seguir. A classe “Empregado” mapeia a tabela “empregados” no banco de dados:

1. <?php
2. namespace App\Models;
3. /\*\*
4. \* @ORM/Entity
5. \* @ORM/Table(name=empregados)
6. \*/
7. class Empregado {
8. /\*\*
9. \* @ORM/Id
10. \* @ORM/Generated
11. \* @ORM/Column(name=empregado\_id, type=int)
12. \*/
13. public $id;
14. /\*\*
15. \* @ORM/Column(type=string, length=50)
16. \*/
17. public $nome;
18. /\*\*
19. \* @ORM/Column(name=data\_nasc, type=date)
20. \*/
21. public $dataNasc;
22. }

Fonte: Autor, 2018

O exemplo acima exibe o mapeamento da classe **“Empregado”**, ela representa a tabela “empregados” e possui os atributos **“$id”** (chave primária do tipo inteiro e com o nome da coluna “empregado\_id”), **“$nome”** (tipo texto com o tamanho 50) e **“$dataNasc”** (tipo data com o nome da coluna “data\_nasc”).

* 1. **Exemplos de Mapeamento de Relacionamentos**

Relacionamento entre tabelas é um recurso essencial nos bancos de dados, para representar isso devidamente, no mundo orientado a objetos, existe as seguintes opções.

* + 1. **Um para Um**

Um exemplo básico de como mapear um relacionamento do tipo “um para um”, pode ser encontrado no exemplo a seguir. A classe “Empregado” mapeia o relacionamento através do atributo “$informacoes”:

1. <?php
2. namespace App\Models;
3. /\*\*
4. \* @ORM/Entity
5. \* @ORM/Table(name=empregados)
6. \*/
7. class Empregado {
8. ...
9. /\*\*
10. \* @ORM/HasOne(class=App\Models\EmpregadoInfo, cascade={ALL})
11. \*/
12. public $informacao;
13. ...
14. }

Fonte: Autor, 2018

O exemplo acima exibe o mapeamento da classe **“Empregado”**, ela representa a tabela “empregados” e possui o atributo **“$informacao”**, que por sua vez, mapeia o relacionamento do tipo “um para um” com a classe **“EmpregadoInfo”**.

A classe “EmpregadoInfo” referenciada pela classe “Empregado” seria:

1. <?php
2. namespace App\Models;
3. /\*\*
4. \* @ORM/Entity
5. \* @ORM/Table(name=empregado\_info)
6. \*/
7. class EmpregadoInfo {
8. ...
9. /\*\*
10. \* @ORM/BelongsTo(class=App\Models\Empregado)
11. \* @ORM/JoinColumn(name=empregado\_id)
12. \*/
13. public $empregado;
14. ...
15. }

Fonte: Autor, 2018

O exemplo acima exibe o mapeamento da classe **“EmpregadoInfo”**, ela representa a tabela “empregado\_info” e possui o atributo **“$empregado”**, que por sua vez, mapeia a outra ponta o relacionamento do tipo “um para um” com a classe **“Empregado”** e mapeia a coluna “empregado\_id” da tabela “pedido”.

* + 1. **Um para Muitos**

Um exemplo básico de como mapear um relacionamento do tipo “um para muitos”, pode ser encontrado no exemplo a seguir. A classe “Cliente” mapeia o relacionamento através do atributo “$pedidos”:

1. <?php
2. namespace App\Models;
3. /\*\*
4. \* @ORM/Entity
5. \*/
6. class Cliente {
7. ...
8. /\*\*
9. \* @ORM/HasMany(class=App\Models\Pedido)
10. \*/
11. public $pedidos;
12. ...
13. }

Fonte: Autor, 2018

O exemplo acima exibe o mapeamento da classe **“Cliente”**, ela representa a tabela “cliente” e possui o atributo **“$pedidos”**, que por sua vez, mapeia o relacionamento do tipo “um para muitos” com a classe **“Pedido”**.

A classe “Pedido” referenciada pela classe “Cliente” seria:

1. <?php
2. namespace App\Models;
3. /\*\*
4. \* @ORM/Entity
5. \*/
6. class Pedido {
7. ...
8. /\*\*
9. \* @ORM/BelongsTo(class=App\Models\Cliente)
10. \*/
11. public $cliente;
12. ...
13. }

Fonte: Autor, 2018

O exemplo acima exibe o mapeamento da classe **“Pedido”**, ela representa a tabela “pedido” e possui o atributo **“$cliente”**, que por sua vez, mapeia a outra ponta o relacionamento do tipo “um para muitos” com a classe **“Cliente”** e mapeia a coluna “cliente\_id” da tabela “pedido”.

* + 1. **Muitos para Muitos**

Um exemplo básico de como mapear um relacionamento do tipo “muitos para muitos”, pode ser encontrado no exemplo a seguir. A classe “Empregado” mapeia o relacionamento através do atributo “$projetos”:

1. <?php
2. namespace App\Models;
3. /\*\*
4. \* @ORM/Entity
5. \*/
6. class Empregado {
7. ...
8. /\*\*
9. \* @ORM/ManyToMany(class=App\Models\Projeto)
10. \* @ORM/JoinTable(tableName=empregado\_projeto, join={ name=empregado\_id }, join={ name=projeto\_id })
11. \*/
12. public $projetos;
13. ...
14. }

Fonte: Autor, 2018

O exemplo acima exibe o mapeamento da classe **“Empregado”**, ela representa a tabela “empregado” e possui o atributo **“$projetos”**, que por sua vez, mapeia o relacionamento do tipo “muitos para muitos” com a classe **“Projto”** e é o lado principal do relacionamento, isso quer dizer que é o lado do relacionamento que define as informações da tabela de ligação.

A classe “Projeto” referenciada pela classe “Empregado” seria:

1. <?php
2. namespace App\Models;
3. /\*\*
4. \* @ORM/Entity
5. \*/
6. class Projeto {
7. ...
8. /\*\*
9. \* @ORM/ManyToMany(class=App\Models\Empregado, mappedBy=projetos)
10. \*/
11. public $empregados;
12. ...
13. }

Fonte: Autor, 2018

1. ***Entity Manager***

O *EntityManager* é o gerenciador de entidades do ORM, através dele é que o desenvolvedor tem acesso às funções de consulta, persistência e deleção.

Essa seção irá abordar as funcionalidades que envolvem o *EntityManager*.

* 1. **Obtendo Uma Instância do *EntityManager***

Para obter uma instância, basta que o ORM crie uma, da seguinte maneira:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $orm->addConnection(‘exemplo-sqlite’);
5. $em = $orm->createEntityManager();
6. ?>

Fonte: Autor, 2018

O método *createEntityManager* retorna uma instância do *EntityManager* utilizando a conexão padrão do ORM, ou seja, todas as operações realizadas no *EntityManager* serão realizadas através da conexão padrão, no caso do exemplo acima, a conexão “exemplo-mysql”.

Para utilizar uma conexão secundária, é necessário passar o nome da conexão para o método *createEntityManager*:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $orm->addConnection(‘exemplo-sqlite’);
5. $em = $orm->createEntityManager(‘exemplo-sqlite’);
6. ?>

Fonte: Autor, 2018

Agora o método *createEntityManager* irá retorna uma instância do *EntityManager* utilizando a conexão secundária “exemplo-sqlite”.

* 1. **Transações**

A transação no banco de dados, é uma unidade que realiza um trabalho, ou seja, qualquer trabalho realizado no banco de dados, mesmo que em etapas, é realizado dentro de uma transação e todas as operações realizadas dentro dessa transação tem a garantia de ser executada integralmente no banco de dados, isso significa que caso um problema ocorra durante a execução de uma transação, as operações já realizada dentro da mesma transação serão desfeitas.

Bernstein (2009) define que uma transação de banco de dados deve ser atômica, consistente, isolada e durável conhecido pela sigla ACID.

* **Atômica:** uma série indivisível e irredutível de operações de banco de dados;
* **Consistente:** toda e qualquer transação deve alterar os dados no banco apenas de formas permitidas, ou seja, quaisquer dados gravados devem ser válidos de acordo com todas as regras definidas na tabela;
* **Isolada:** determina como a integridade da transação é visível para outros usuários e sistemas;
* **Durável:** garante que as transações que foram confirmadas sobreviverão permanentemente no banco de dados.
  + 1. ***Begin Transaction***

Para iniciar uma transação no ORM, o desenvolvedor deve usar o método *beginTransaction*.

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
5. $em->beginTransaction();
6. ?>

Fonte: Autor, 2018

O método *beginTransaction* irá criar iniciar uma transação para a conexão padrão do ORM. Para fechar a transação, o desenvolvedor deverá executar o método *commit* ou o método *rollback*.

* + 1. ***Commit***

O método *commit* irá confirmar a transação atual para que as operações realizadas no banco de dados sejam efetivadas. Liberando assim o *EntityManager* para criar uma nova transação.

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
5. $em->beginTransaction();
6. // operações no banco de dados, por exemplo, um insert
7. $em->commit();
8. ?>

Fonte: Autor, 2018

* + 1. ***Rollback***

O método *rollback* irá desfazer as operações realizadas na transação atual. Liberando assim o *EntityManager* para criar uma nova transação. Esse método é normalmente utilizado em um *try*/*catch* para tratamento de erros.

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
5. try {
6. $em->beginTransaction();
7. // operações no banco de dados
8. $em->commit();
9. } catch(Exception $ex) {
10. $em->rollback();
11. }
12. ?>

Fonte: Autor, 2018

* 1. ***Find***

O método *find* é utilizado para carregar um registro do banco de dados através da chave primária dessa tabela.

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
5. $id = 1;
6. $empr = $em->find(App\Models\Empregado::class, $id);
7. ?>

Fonte: Autor, 2018

O método *find* espera como parâmetro a classe que mapeia a tabela a ser utilizada na consulta e a chave primária correspondente ao registro necessário.

* 1. ***List***

O método *list* é utilizado para carregar uma lista de registros do banco de dados, porém, sem utilizar filtros (clausula *where*).

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
5. $top = 15;
6. $page = 2;
7. $page = 10;
8. $emprs = $em->find(App\Models\Empregado::class);
9. $emprs = $em->find(App\Models\Empregado::class, $top);
10. $emprs = $em->find(App\Models\Empregado::class, $page, $quantity);
11. ?>

Fonte: Autor, 2018

O método *list* pode ser usado de três formas. A primeira, passando como parâmetro somente a classe que mapeia a tabela a ser listada, e traz todos os registros que estão atualmente na tabela. A segunda, passando como parâmetro a classe que mapeia a tabela a ser lista e a quantidade de registros a serem carregados pela consulta, no caso do exemplo, os primeiros 15 registros. A terceira e última forma, utilizada para dividir os registros em páginas, passando como parâmetro a classe que mapeia a tabela a ser lista, a página a ser trazida e a quantidade de registros que a página deve conter.

* 1. ***Query Builder***
     1. **Paginação**
     2. **Agregação**
     3. ***Group by***
     4. ***Order by***
     5. ***Having***
  2. ***Save***

O método *save* é utilizado para cadastrar ou alterar um registro no banco de dados. Caso o registro já exista e a chave primária esteja preenchida, o método *save* irá atualizar o registro existente no banco de dados, caso contrário, um novo registro será adicionado ao banco.

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
6. try {
7. $empregado = new App\Models\Empregado();
8. $empregado->nome = ‘João da Silva’;
9. $empregado->dataNasc = new Empregado(‘1990-05-23’);
10. $em->beginTransaction();
11. $empregadoNovo = $em->save($empregado);
12. $em->commit();
13. } catch(Exception $ex) {
14. $em->rollback();
15. }
16. ?>

Fonte: Autor, 2018

O exemplo acima irá resultar em um novo registro na tabela “empregado” . O método *save* irá retornar o novo registro.

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
6. try {
7. $empr = $em->find(App\Models\Empregado::class, 1);
8. $empr->nome = ‘Pedro da Silva’;
9. $em->beginTransaction();
10. $emprAtualizado = $em->save($empr);
11. $em->commit();
12. } catch(Exception $ex) {
13. $em->rollback();
14. }
15. ?>

Fonte: Autor, 2018

O exemplo acima irá atualizar um registro já existente na tabela “empregado” com a chave primária de valor “1”. O método *save* irá retornar o registro atualizado.

* 1. ***Remove***

O método *remove* é utilizado para deletar um registro no banco de dados, caso o mesmo exista.

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
6. try {
7. $empr = $em->find(App\Models\Empregado::class, 1);
8. $em->beginTransaction();
9. $linhas = $em->remove($empr);
10. $em->commit();
11. } catch(Exception $ex) {
12. $em->rollback();
13. }
14. ?>

Fonte: Autor, 2018

O exemplo acima irá remover um registro já existente na tabela “empregado” com a chave primária de valor “1”. O método *remove* irá retornar a quantidade de linhas afetadas com essa operação.

1. **Logger**

É possível definir um log para que o ORM possa registar as operações que ele realiza. Através de alguns métodos o desenvolvedor pode configurar o *Logger* para as suas necessidades.

O método *setLogDisable* define se o log deve estar ativo ou inativo. Passando “false” por parâmetro para essa função, o desenvolvedor desabilita o *Logger* e passando “true” ele é habilitado novamente.

O método *setLogLocation* define a localização onde os arquivos do log serão salvos.

O método *setLogFilename* define o prefixo que será utilizado para criar o nome do arquivo de log.

O método *setLogLevel* define qual o nível de log o ORM deve registrar, os valores esperados são “DEBUG”, “INFO”, “WARNING” ou “ERROR”. O valor padrão é “ERROR”.

O método *setLogOccurrency* define com qual frequência o *Logger* deve criar um novo arquivo. Essa configuração impede que os arquivos de log tenham tamanhos grandes demais. Os valores esperados são “DAILY” ou “MONTHLY”. O valor padrão é “DAILY”.

O método *getLogger* retorna a instância que está sendo utilizada pelo ORM. O Logger do ORM pode ser utilizado em outras partes da aplicação. Para isso basta usar os métodos *debug*, *info*, *warning* e *error* para registrar uma entrada no log conforme o nível desejado.

1. **Driver**

O desenvolvedor pode criar o próprio *Driver* para um banco de dados ou uma versão ainda não suportada pelo ORM. O *Driver* permite que o ORM consiga se comunicar com o banco de dados correspondente, por exemplo, o banco de dados MySQL, possui um *Driver* correspondente para permitir a comunicação do ORM com este banco de dados.

Para criar um novo *Driver*, o desenvolvedor precisa atender a algumas regras:

* **Nome do arquivo:** o nome do arquivo precisa conter o nome do banco de dados de acordo com o nome esperado pelo PDO (*PHP Data Object*, é uma interface do PHP para acessar banco de dados), por exemplo, “sqlite.php”. Caso o *Driver* seja para uma vesão específica, a versão precisa estar também no nome do arquivo, por exemplo, para a versão 5.11 do banco de dados MySQL, o nome do arquivo seria “mysql-5.11”;
* **Nome da classe:** o nome da classe precisa ser equivalente ao nome do arquivo conforme definido acima, por exemplo, para a versão 5.11 do banco de dados MySQL, o nome da classe seria “MySQLDriver\_5\_11”. E caso não possua uma versão definida, “MySQLDriver”. Essa é mais uma sugestão do que uma regra, mas ajuda evitar problemas de classe com nomes iguais;
* **Retonar o nome da classe:** o nome da classe precisa ser retornado ao final do arquivo, por exemplo, “return MySQLDriver::class;”;
* **Localização do arquivo:** o arquivo precisa estar localizado na pasta “driver” dentro da pasta raiz do ORM, por exemplo, “/home/user/app/orm/driver”;
* **Precisa estender de ORM\Core\*Driver*:** a classe para o novo *Driver* precisa extender a classe “ORM\Core\*Driver*”, do contrário, o ORM não irá considerar como válido.
* **O novo Driver precisa ser *Singleton*:** a classe precisa seguir o *desing pattern* *Singleton* para que o ORM use uma instância única do ORM. Caso a classe não seja *Singleton*, o ORM irá emitir um erro.
* **As configurações do novo *Driver*:** o *Driver* possui uma série de configurações para que o ORM consiga se comunicar com o banco de dados. Elas devem ser feitas no método construtor da classe. As configurações serão listadas mais à frente.

O *Driver* possui as seguintes configurações:

* **$GENERATE\_ID\_TYPE:** indica como o banco de dados faz a incrementação do valor da chave primária. Os valores esperados são: “ATTR”, “QUERY” ou “SEQUENCE”;
* **$GENERATE\_ID\_ATTR:** caso o tipo de incrementação seja igual a “ATTR”, aqui é definida qual é o atributo que o ORM irá usar, por exemplo, “AUTO\_INCREMENT” para o banco de dados MySQL;
* **$GENERATE\_ID\_QUERY:** caso o tipo de incrementação seja igual a “QUERY” ou “SEQUENCE”, aqui é definida *query* que o ORM irá usar para realizar a incrementação do valor da chave primária, por exemplo, “select orm\_sequence.nextval from dual” para o banco de dados Oracle;
* **$SEQUENCE\_NAME:** define o nome da sequence que o ORM irá criar caso o ORM precise criar as tabelas no banco de dados. O valor padrão é “orm\_sequence”;
* **$IGNORE\_ID\_DATA\_TYPE:** caso o valor desse atributo seja “true”, o ORM irá ignorar o valor definido para a coluna, como pode acontecer, por exemplo, com o banco de dados PostgreSQL;
* **$FK\_ENABLE:** indica de o ORM deve criar o campo como chave estrangeira. O valor padrão é “true”;
* **$PAGE\_TEMPLATE:** define como o banco de dados faz uma consulta paginada, por exemplo, o banco de dados MySQL possui a clausula “LIMIT” para realiza essa tarefa;
* **$TOP\_TEMPLATE:** define como o banco de dados faz uma consulta trazendo somente um determinado número de registros, por exemplo, o banco de dados MySQL possui a clausula “LIMIT” para realiza essa tarefa;
* **$DATA\_TYPES:** define os tipos de dados aceitos pelo ORM e mapeia para os dados suportados pelo banco de dados. Os tipos recomendados são: “'string”, “int”, “float”, “lob” (*Large Object*, por exemplo, BLOB ou CLOB), “date”, “time”, “datetime”, “bool”;
* **$FORMATS:** define os formatos que o banco de dados aceita para os tipos “date”, “time” e “datetime”.

A seguir, um exemplo de implementação para uma classe do *Driver*:

1. <?php
2. use ORM\Contants\*GeneratedTypes;*
3. use ORM\Core\*Driver;*
4. id (!class\_exists(‘MySQLDriver\_5\_11’)) {
5. class MySQLDriver\_5\_11
6. {
7. private *static* $instance;
8. const NAME = ‘MySQL’;
9. const VERSION = ‘5.11’;
10. private *function* \_\_construct()
11. {
12. $this->GENERATE\_ID\_TYPE = *GeneratedTypes*::ATTR;
13. $this->GENERATE\_ID\_ATTR = ‘AUTO\_INCREMENT’;
14. $this->PAGE\_TEMPLATE
15. $this->TOP\_TEMPLATE = ‘AUTO\_INCREMENT’;
16. $this->DATA\_TYPES = [
17. ‘string’ => ‘VARCHAR(%d)’,
18. ‘int’ => ‘INTEGER’,
19. ‘float’ => ‘DOUBLE’,
20. ‘lob’ => ‘TEXT’,
21. ‘date’ => ‘DATE’,
22. ‘time’ => ‘TIME’,
23. ‘datetime’ => ‘DATETIME’,
24. ‘bool’ => ‘TINYINT(1)’,
25. ];
26. }
27. public *static function* getInstance() : *Driver*
28. {
29. if (!*self*::$instance) {
30. *self*::$instance = new *self*();
31. }
32. return *self*::$instance;
33. }
34. public *function* getConnection(*Array* $config) : \*PDO*
35. {
36. $this->validateFields([‘db’, ‘host’, ‘schema’, ‘user’, ‘pass’], $config);
37. $dsn = ”$config[db]:host=$config[host];dbname=$config[schema]”;
38. return $this->createConnection($dsn, $config[‘user’] ?? null, $config[‘pass’] ?? null);
39. }
40. }
41. }
42. return *MySQLDriver\_5\_11*::class;

Fonte: Autor, 2018

**Referências**

**Philip A. Bernstein, Eric Newcomer (2009).** Principles of Transaction Processing, 2nd Edition, Morgan Kaufmann (Elsevier).