Coordenadoria de Tecnologia da Informação

Tecnologia em Análise e Desenvolvimento de Sistemas

Lumberjack’s ORM

*Diego Francklin Martins dos Santos*

Sorocaba

Junho – 2018

Coordenadoria de Tecnologia da Informação

Tecnologia em Análise e Desenvolvimento de Sistemas

Lumberjack’s ORM

Diego Francklin Martins dos Santos

Prof. Cristiane Palomar Mercado - Orientadora

Sorocaba

Junho – 2018

**Dedicatória**

Dedico este trabalho aos meus pais, amigos e esposa que sempre me ajudaram e incentivaram a seguir adiante e vencer todas as barreiras e desafios.

**Agradecimento**

Agradeço a todos que de uma forma ou de outra colaboraram para com este trabalho.

**Sumário**

[Resumo 6](#_Toc516748975)

[1. Objetivo 7](#_Toc516748976)

[2. Introdução 8](#_Toc516748977)

[3. *Framework* 9](#_Toc516748978)

[4. *Design Patterns* 11](#_Toc516748979)

[4.1. Quando não usar? 12](#_Toc516748980)

[4.2. Qual a diferença entre o *Design Pattern* e o *Framework* 12](#_Toc516748981)

[5. ORM 13](#_Toc516748982)

[5.1. *Active Record* 14](#_Toc516748983)

[5.2. *Data Mapper* 14](#_Toc516748984)

[6. Estudo de caso 15](#_Toc516748985)

[6.1. Tecnologias 15](#_Toc516748986)

[6.2. Mapeamento 15](#_Toc516748987)

[6.3. Conexões 20](#_Toc516748988)

[6.4. *Drivers* 22](#_Toc516748989)

[6.4.1. Criar Conexões 23](#_Toc516748990)

[6.4.2. Tipos de Dados 23](#_Toc516748991)

[6.4.3. Valores Auto Gerados 24](#_Toc516748992)

[6.4.4. Paginação 25](#_Toc516748993)

[6.5. Criação das Tabelas 25](#_Toc516748994)

[7. Codificação 28](#_Toc516748995)

[8. Considerações Finais 29](#_Toc516748996)

[Referências 30](#_Toc516748997)

[Glossário 32](#_Toc516748998)

[Anexo – Manual do Desenvolvedor 35](#_Toc516748999)

**Lista de Figuras**

[Figura 1: Estrutura do *Framework* 7](#_Toc516749000)

[Figura 2: Classe Pessoa 16](#_Toc516749001)

[Figura 3: Exemplo de uso da classe *ReflectionClass* 17](#_Toc516749002)

[Figura 4: Exemplo da classe *ReflectionProperty* 17](#_Toc516749003)

[Figura 5: Exemplo de Expressão Regular 18](#_Toc516749004)

[Figura 6: Exemplo de arquivo de conexões 21](#_Toc516749005)

[Figura 7: Exemplo para utilizar a conexão 22](#_Toc516749006)

[Figura 8: Exemplo para configurar a criação de tabelas 25](#_Toc516749007)

[Figura 9: Exemplo para configurar para apagar as tabelas 26](#_Toc516749008)

[Figura 10: Exemplo para executar ações antes de apagar as tabelas e depois de criá-las 27](#_Toc516749009)

**Lumberjack’s ORM**

# Resumo

Durante a criação e manutenção de projetos é investido um grande esforço e uma grande quantidade de tempo para analisar, criar e integrar bancos de dados à aplicação. E geralmente, torna-se muito trabalhoso migrar para outros bancos de dados e com grande potencial de se tornar um verdadeiro pesadelo.

O ORM (*Object-Relational Mapping*) é uma técnica que visa o mapeamento entre o mundo orientado à objetos (os nossos modelos) e o mundo relacional (o banco de dados).

Essa técnica está em crescente uso no mercado pois permite abstrair a utilização de comandos SQL, não cria dependência com uma marca de banco de dados específica e pode-se observar um ganho de produtividade nas tarefas diárias.

Diversos *frameworks* surgiram no mercado nos últimos anos escritos em diversas linguagens de programação para facilitar a utilização dessa técnica. Porém, durante os anos que trabalhei com a linguagem PHP e utilizei diversos *frameworks* *ORM* para a mesma, identifiquei que alguns possuem certas limitações, as quais este trabalho pretende remover.

# Objetivo

O objetivo deste trabalho é desenvolver um *framework* chamado Lumberjack’s ORM para facilitar a utilização de bancos de dados relacionais em aplicações desenvolvidas em PHP, permitindo que estas eliminem por completo a dependência com a linguagem SQL e com um banco de dados de marca específica, tornando simples a configuração e integração com os bancos de dados mais comuns do mercado, de forma que o *framework* forneça um suporte à diversas funcionalidades que visam simplificar tarefas, que antes seriam feitas manualmente e ocupariam muito tempo; tempo este que poderia ter um aproveitamento mais significativo se focado em tarefas mais importantes do projeto.

Figura 1: Estrutura do *Framework*



Fonte: Autor, 2018

O *framework* conta com as seguintes funcionalidades em sua arquitetura:

* Mapeamento de entidades (tabelas ou *views*) utilizando *Annotations*, possibilitando também que seja mapeado todos os tipos de relacionamentos entre as entidades;
* Armazenamento e utilização de múltiplas conexões com bancos de dados;
* Possibilitar a criação de tabelas no banco de dados caso não existam baseando-se no mapeamento dos modelos;

Para fins de demonstração das funcionalidades do *framework*, será criada uma aplicação simples.

# Introdução

Para desenvolver um software de qualidade, é preciso pensar em manutenção e evolução. Desenvolvedores lidam com diversas variáveis e resolução de diversos problemas ao longo de um dia de trabalho. Muitos desses problemas são recorrentes, e ao invés de desenvolver novas soluções, eles reutilizam soluções que funcionaram no passado e as utilizam repetidamente em seus projetos.

Por isso a importância dos *frameworks* e *design patterns (*padrões de projetos), que proporcionam reaproveitamento de soluções para projetos e não apenas a reutilização de código.

Atualmente no mercado existem alguns *frameworks* ORM em PHP, porém, vários tem problemas com documentação ou são insuficientes no que diz respeito a relacionamentos entre tabelas, principalmente em um relacionamento N para N.

Durante a criação de novos projetos, exige do desenvolvedor uma preocupação extra no que se relaciona a banco de dados, pois, sem a utilização de um *framework* ORM as aplicações ficam muito dependentes de um determinado banco em si, o que torna a manutenção e/ou evolução da aplicação muito mais complicada de ser realizada, por exemplo, para migrar para um outro banco de dados mais robusto no futuro. Sendo assim, como estudo de caso, será apresentada as funcionalidades do *framework* para a persistência e manipulação de suas informações.

Ao final do projeto, o ORM desenvolvido será disponibilizado gratuitamente à comunidade de desenvolvedores através de um repositório público no GitHub, uma vez que possua uma versão estável.

# *Framework*

Com os avanços tecnológicos na área de desenvolvimento de software, vem surgindo diversas abordagens para reduzir a complexidade do processo de produção de um software e aumentar a qualidade do produto e a produtividade da equipe de desenvolvimento, e uma dessas abordagens é a reutilização de código ou soluções, com o objetivo de evitar a resolução de cada problema partindo do básico ou mesmo do zero.

Segundo Gamma (1995), uma coisa que os projetistas de software mais experientes sabem é que não devem resolver cada problema a partir de princípios elementares ou do zero, mas reutilizar soluções que funcionaram no passado.

Essa ideia de reutilização não é nova nem exclusiva para o desenvolvimento de software, ela surgiu quando começaram a encontrar soluções consistentes e que poderiam ser aplicadas a novos problemas. O uso frequente dessas soluções faz com que fiquem conhecidas e se tornem aceitas, generalizadas e padronizadas.

De acordo com os princípios da Engenharia de Software, principalmente em termos de reutilização, a Orientação a Objetos surgiu como poderoso instrumento para o desenvolvimento de software. De acordo com Jones (2001), os objetos tornaram-se os blocos de construção onipresentes do software moderno e a orientação a objetos o paradigma dominante da era contemporânea.

O *framework* é uma técnica da Orientação a Objetos que é voltada exclusivamente para o reaproveitamento de soluções que se beneficia de três das características que linguagens de programação orientadas à objetos oferecem: herança, polimorfismo e abstração.

Um *framework* engloba uma funcionalidade específica e comum a várias aplicações. Ele define a organização de um software, os tipos de objetos e as interações entre esses objetos. Ele pode ser considerado o esqueleto no qual o sistema se sustenta.

Os *frameworks* possuem interfaces complexas e são de fácil customização. Em sua composição contam com interfaces e classes abstratas e seu uso se dá através da especialização ou composição dos seus serviços.

Por isso, pode-se afirmar que os *frameworks* possuem pontos fixos, que são os serviços e componentes implementados nele, os quais realizam chamadas indiretas aos pontos extensíveis, que são serviços e funcionalidades que devem ser implementados por meio de herança, onde os desenvolvedores irão inserir os códigos relacionados ao domínio do problema da aplicação.

Segundo Fayad (1999), a utilização de *frameworks* apresenta os seguintes benefícios:

I. Melhora a modularização – encapsulamento dos detalhes voláteis de implementação através de interfaces estáveis.

II. Aumenta a reutilização – definição de componentes genéricos que podem ser replicados para criar novos sistemas.

III. Extensibilidade – favorecida pelo uso de métodos “gancho” que permitem que as aplicações estendam interfaces estáveis.

IV. Inversão de controle – o código do desenvolvedor é chamado pelo código do *framework*. Dessa forma, o *framework* controla a estrutura e o fluxo de execução dos programas.

# *Design Patterns*

Segundo Gerra (2012), um *design pattern* (padrão de projeto) é composto por três partes: um contexto, um problema e uma solução. O contexto define o ambiente e as circunstâncias dentro do qual algo existe. O problema é uma questão indefinida, algo que precisa de investigação e solução, e que está diretamente relacionado ao contexto. E por fim a solução, diz respeito a resposta do problema que ajuda a soluciona-lo.

Um *design pattern* descreve uma solução para um problema em um contexto, porém não descreve qualquer solução, mas sim, uma solução que já tenha sido utilizada com sucesso diversas vezes em vários contextos diferentes. Por isso, podemos dizer que um padrão não descreve soluções novas, mas soluções já testadas e consolidadas. Mas para ser um padrão, uma solução não basta ser recorrente, mas precisar ser uma boa solução.

Segundo Christopher Alexander (1977), “cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca o fazer da mesma maneira”.

Os padrões popularizaram-se com o conhecido livro “*Design Patterns: Elements of Reusable Object-Oriented Software*”. O livro é um catálogo que possui 23 *design patterns*, cada uma descrevendo uma solução para problemas de software orientado a objeto, o contexto e suas eventuais consequências. As soluções são utilizadas até hoje por desenvolvedores em todo o mundo.

A primeira definição do que seria um design pattern foi apresentada no livro “*A Times Way of Building*” (Oxford University Press, 1979), que é: “Cada padrão é uma regra de três partes, que expressa uma relação entre um certo contexto, um problema e uma solução”.

## Quando não usar?

Quanto mais *design patterns* eu utilizar, melhor meu projeto vai ficar? Não! Um padrão é uma solução para um problema, portanto, utilizá-lo onde não há um problema apenas complica mais as onde não há necessidade.

Um *design pattern* também possui consequências negativas que podem se sobrepor às vantagens em alguns casos. Sua utilização desnecessária pode ser desastrosa e isso, no mínimo, aumenta a complexidade do sistema sem necessidade, o que torna muito mais difícil a manutenção do mesmo. Esse uso inadequado ou o uso exagerado é o que constitui o chamado *anti-pattern* (antipadrão). De acordo com Andrew Koenig (2018), se um padrão representa a “melhor prática”, então um *anti-pattern* representa uma “lição aprendida”.

## Qual a diferença entre o *Design Pattern* e o *Framework*

O conceito de design pattern é muito parecido ao do *framework*, e isso gera frequentemente confusão sobre o que caracteriza cada um e suas particularidades.

Para Fowler (1997), os padrões descrevem maneiras comuns de fazer as coisas e são coletados por pessoas que identificam temas repetitivos em projetos. Essas pessoas identificam cada tema e o descrevem de modo que outras pessoas possam ler o padrão e ver como aplicá-los.

Padrões documentam *frameworks* e ajudam a garantir o uso correto de sua funcionalidade. Para Johnson (1997), padrões são elementos microarquiteturas de *frameworks*. Um *framework* usualmente utiliza muitos padrões, ou seja, padrões são menores do que muito *frameworks*, podendo ser vistos como seus blocos construtores, ou seja, as partes constituintes de um *framework*.

# ORM

Para trabalhar com dados em uma aplicação, você provavelmente precisará de um ORM. Nesse capítulo será abordado o que é um ORM e quais os tipos mais comuns.

De acordo com Bauer (2004), ORM (da sigla em inglês que significa *Object-Relational Mapping*, traduzindo para português Mapeamento Objeto-Relacional) é a camada de aplicação que está situada entre a camada de acesso a dados da aplicação e o banco de dados. Ele é responsável por tomar conta de boa parte do trabalho pesado criando, atualizando, deletando e consultando dados do banco de dados.

Em uma linguagem de programação orientada a objetos, o principal ponto de referência para os desenvolvedores é o objeto. No entanto, os bancos de dados trabalham com a estrutura de tabelas e registros, em sua forma mais simples (textos e números).

Trabalhando com objetos, uma única estrutura para armazenar as informações e relacionamentos nas propriedades do mesmo objeto. No banco de dados, as informações são armazenadas em registros através de uma ou várias tabelas.

Pode-se dizer então que o ORM é a camada “mágica” que transforma os dados armazenadas na forma de objetos para a estrutura das tabelas utilizadas pelo banco de dados e vice-versa.

Normalmente não é necessário se preocupar com o tipo de ORM que está sendo utilizado para desenvolver a aplicação, porém, conforme o desenvolvedor se aprofunda em como aplicações são desenhadas, vale a pena explorar um pouco mais a fundo os tipos de ORM disponíveis para uso.

Os tipos mais comuns são o padrão *Active Record* e o padrão *Data Mapper*.

## *Active Record*

De acordo com Fowler (2018), dentro do padrão *Active Record*, cada objeto do modelo precisa saber como se comunicar com o banco de dados e como persistir suas informações, ou seja, o objeto do modelo contém a lógica para abrir uma conexão com o banco de dados, criar, alterar e deletar dados do banco de dados. Portanto, não existe uma camada específica para que essas tarefas sejam executadas.

Em sistemas mais simples, o desenvolvimento se torna mais rápido e leve, porém, existe uma dependência muito grande entre o modelo e o esquema do banco de dados. Isso faz com que a manutenção seja mais complicada, por algumas razões: qualquer mudança em um, implica diretamente na mudança do outro; quanto mais complexos os modelos são, mais complexo se torna o mapeamento; aumenta muito a duplicação de código pois cada modelo acaba possuindo logicas de como persistir os dados.

## *Data Mapper*

Segundo Fowler (2018), o padrão *Data Mapper* consiste em uma camada de objetos que mapeiam, ou seja, movem dados entre os objetos do modelo e o banco de dados, mantendo um independente do outro. O uso desse padrão torna o código mais limpo e simples de se compreender.

Uma implementação simples de *Data Mapper*, mapeia uma tabela do banco de dados para uma classe equivalente, campo a campo.

Os objetos do modelo geralmente estão muito interconectados, por isso é necessário utilizar um padrão que limita até qual nível desses relacionamentos é desejado alcançar, do contrário, seria possível trazer muito mais dados do banco de dados do que o desejado. Esse padrão é denominado *Lazy Load*.

Fowler (2018) define *Lazy Load* como um objeto que não contém todos os dados que você precisa, ou seja, o objeto não está completo, mas sabe como obter os dados.

Para o desenvolvimento desse trabalho, esse foi o padrão escolhido.

# Estudo de caso

O foco desse trabalho é desenvolver um *framework* que atenda às necessidades dos programadores, com relação ao mapeamento de relacionamento entre tabelas dos bancos de dados 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 (2018) 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*.

De acordo com Dohms (2018), 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.

As linguagens de programação possuem uma estrutura para documentar as classes, propriedades e métodos, essa estrutura é o bloco de comentário. No PHP, essa estrutura é comumente conhecida 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 modelo de dados:

Figura 2: Classe Pessoa

1. <?php
2. namespace App\Model;
3. /\*\*
4. \* @ORM/Entity
5. \* @ORM/Table(name=pessoa)
6. \*/
7. class Pessoa {
8. /\*\*
9. \* @ORM/Id
10. \* @ORM/Generated
11. \* @ORM/Column(name=pessoa\_id, type=int)
12. \*/
13. public $id;
14. /\*\*
15. \* @ORM/Column(name=nome, 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

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

Figura 3: Exemplo de uso da classe ReflectionClass

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

Fonte: Autor, 2018

No exemplo mostrado na Figura 3, na linha 2, é criada a instância 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:

Figura 4: Exemplo da classe ReflectionProperty

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

Fonte: Autor, 2018

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 expressões regulares para extrair as *annotations* e suas informações.

Expressão Regular (PHP, 2018) 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” 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” pára quando encontra a primeira correspondência e a função “preg\_match\_all” procura no texto inteiro e retorna todas as correspondências.

Na Figura 2, a classe “Pessoa” possui algumas *annotations*, entre elas a *annotation* “@ORM/Table”, que informa, entre outras coisas, o nome da tabela em questão que deve ser mapeada. Então para identificar essa *annotation* e suas propriedades, para então extrair suas informações, faz-se da seguinte maneira:

Figura 5: Exemplo de Expressão Regular

1. <?php
2. $reflection = new ReflectionClass(‘App\Model\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. ?>

Fonte: Autor, 2018

As linhas 2 e 3 já foram mostradas anteriormente, mas nas linhas 4, 5 e 6, estão declaradas as expressões regulares que serão usadas para identificar a *annotation* e extrair as informações dela. Para um melhor entendimento das expressões regulares, ela pode ser quebrada em algumas partes menores:

* **“/”**

A primeira barra indica o começo da expressão regular e a última indica o começo da expressão regular;

* **“@ORM\/”**

Esse trecho indica que o texto correspondente precisa, obrigatoriamente, começar com o texto “@ORM/”;

* **“[]”**

O trecho “[]” indica um conjunto, ou seja, a expressão deve considerar qualquer combinação dos caracteres entre colchetes como um texto válido;

* **“@A-Za-z0-9=,\_\/\(\)\{\}”**

O trecho acima indica que a expressão regular deve considerar o caractere “@”, todas letras maiúsculas e minúsculas, os números de 0 a 9, os caracteres “=”, “,”, “/”, “(“, “)”, “{“, “}”;

* **“\s”**

O trecho “\s” indica que a expressão regular deve considerar os caracteres de espaço e tabulação;

* “\w”

O “\w” representa uma palavra com letras, números e underline;

* **“i”**

A letra “i” sinaliza que a expressão regular deve ser *case insensitive*, ou seja, a expressão regular não deve diferenciar entre letras maiúsculas e minúsculas;

* **“+”**

O “+” indica que a expressão deve que o padrão que o precede ocorra uma ou mais vezes, por exemplo, “\w+”, o padrão indica que a expressão deve esperar uma ou mais palavras dentro do texto em que a expressão será aplicada;

* **“?”**

O “?” indica que a expressão deve que o padrão que o precede ocorra zero ou uma vez, por exemplo, “\w?”, o padrão indica que a expressão deve esperar zero ou uma (no caso a primeira) palavra dentro do texto em que a expressão será aplicada.

A combinação dessas partes e de várias outras possíveis é o que cria o padrão a ser encontrado dentro do texto. As funções “preg\_match” e “preg\_match\_all” utilizam a expressão, que é informada no primeiro parâmetro da função, para encontrar as ocorrências dentro do bloco de texto, informado no segundo parâmetro e armazena as ocorrências em um *array* que é armazenado na variável informada no terceiro parâmetro.

Com todas as informações identificadas, elas serão armazenadas em um *array* acessível pelo ORM, para que posteriormente, sejam usadas para realizar os mapeamentos de consultas simples, consultas com relacionamentos (para identificar as colunas usadas para relacionar as duas tabelas), inclusões, alterações e exclusões na tabela a ser mapeada pela classe do modelo.

A leitura e extração das informações de classes para o mapeamento é realizado sob demanda, isso quer dizer que, o mapeamento de uma entidade irá ocorrer somente no momento em que o ORM precise trabalhar com a classe do modelo.

Por exemplo, considerando a classe “Pessoa” na Figura 2, para realizar uma consulta à tabela “pessoa”, o ORM lerá o mapeamento dessa classe somente no momento em que precisar montar a consulta pela primeira vez. Uma vez que o mapeamento esteja completo, essas informações estão disponíveis até o final do processamento do *script* PHP.

## Conexões

Para que se possa enviar comandos ao banco de dados e receber as respostas desses comandos, é necessário que exista uma conexão com o banco de dados para ser possível realizar essa comunicação.

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 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.

Na Figura 6, pode-se ver um exemplo de arquivo de conexões para melhor compreensão.

Figura 6: Exemplo de arquivo de conexões

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

De acordo com o exemplo da Figura 6, duas conexões são definidas, na linha 3 está a declaração de uma conexão com o nome “exemplo-mysql” e na linha 11 a conexão com o nome “exemplo-sqlite”.

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 banco 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, faz-se através da classe principal do ORM conforme o exemplo a seguir:

Figura 7: Exemplo para utilizar a conexão

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

Fonte: Autor, 2018

A classe “ORM\Orm” concentra entre outras informações, as conexões com o banco de dados, e para isso ela utiliza o *design pattern* *Singleton*. O objetivo desse *design pattern* é definido no livro *“Design Patterns: Elements of Reusable Object-Oriented Software”* (1995) para garantir que existirá apenas uma instância de um determinado objeto e que essa instância estará disponível de forma púbica em todo o escopo da aplicação. Portanto, a classe “ORM/Orm” retém e centraliza todas as informações públicas pertinentes ao ORM, que estarão acessíveis para todo o ORM e no escopo da aplicação.

No entanto, cada banco de dados requer que uma conexão diferente seja estabelecida e como pode-se verificar na Figura 6, cada conexão também requer informações diferentes. Para solucionar essa questão, é necessário delegar essa tarefa a alguém responsável por conhecer as particularidades de cada banco de dados. A solução para isso é o *Driver*.

## *Drivers*

Os bancos de dados possuem muitas características em comum, porém, existem várias características que, embora tenham o mesmo objetivo, são realizadas de maneiras diferentes. Essas diferenças são centralizadas no *Driver* e isoladas do ORM, portanto, o banco de dados pode ser trocado a qualquer momento sem que o ORM precise passar por qualquer adaptação.

Além das variações de banco de dados, essas informações podem variar de versão para versão do mesmo banco de dados. E por isso é necessário que haja uma implementação para cada banco de dados e suas versões para que assim, o ORM possa lidar com todas as particularidades de cada um, separadamente.

O ORM identifica qual *Driver* deve utilizar, usando as chaves “db”, “version” que se encontram na declaração da conexão no arquivo “connection.config.php” conforme abordado na seção Conexões.

Caso uma versão não seja definida, é possível usar um *Driver* genérico. O *Driver* genérico contém as informações mais comuns do banco de dados em questão, por exemplo, o *Driver* do MySQL na versão 5.7 possui as informações específicas para essa versão, e o *Driver* genérico do MySQL contém as informações mais comuns desse banco de dados.

### Criar Conexões

Enquanto o papel do ORM, entre outras coisas, é centralizar as conexões, um dos papeis do *Driver* é criar a conexão com o banco de dados.

O *Driver* valida as informações que constam na declaração da conexão no arquivo “connection.config.php” (conforme abordado na seção Conexões) e utiliza as informações para criar uma conexão e retorna para o ORM.

### Tipos de Dados

O *Driver* concentra os tipos de dados que o ORM pode aceitar na propriedade “type” da *annotation* “@ORM/Column” e também os tipos correspondentes para o banco de dados, por exemplo, o tipo “string” que é informado na *annotation* “@ORM/Column” é um tipo de dados aceito pelo ORM e “varchar” é o tipo de dado equivalente para o banco de dados MySQL.

Para o ORM mapear as classes do modelo para executar os comandos nos dados do banco de dados e mapear os resultados oriundos do banco para as classes do modelo, é necessário converter os dados do PHP para o banco de dados e do banco de dados para o PHP. Para isso, o *Driver* contém a lógica para converter os tipos contidos nele. Para os tipos de dados “date”, “datetime” e “time” que requerem um formato específico para ser armazenado no banco de dados, o *Driver* contém também o formato para que o ORM possa formatar os dados no formado correto.

### Valores Auto Gerados

O ORM possui a *annotation* “@ORM/Generated”, que indica que a chave primária da tabela que a classe modelo está mapeando é auto gerada. Mas, como cada banco de dados gera o valor da chave primária de maneira diferente, o *Driver* deve indicar ao ORM como o banco de dados trabalha nesse aspecto. Existem 3 maneiras para gerar o valor da chave primária que o ORM pode considerar e o *Driver* tem por obrigação informar o tipo pelo qual o valor será gerado.

O primeiro tipo é indicando na coluna da tabela no banco de dados, por exemplo, no MySQL a propriedade da coluna que é a chave primária chama-se “AUTO\_INCREMENT” e no SQLite o nome da propriedade é “AUTOINCREMENT”. Essa diferença também é indicada pelo *Driver*.

O segundo tipo é conhecido como *sequence*. Uma *sequence* é um objeto que gera um número sequencial e está disponível em bancos de dados como o Oracle e o PostgreSQL. O ORM possui também o nome de uma *sequence* genérica, o qual deve ser utilizado para gerar o valor da chave primária de todas as tabelas mapeadas na aplicação.

E por fim, o terceiro tipo, que em último caso, pode ser usado para gerar o valor da chave primaria através de uma consulta na tabela mapeada pela classe do modelo, buscando maior valor da chave primária da tabela para incrementar manualmente. A query a ser usada é fornecida pelo *Driver*.

### Paginação

Em consultas que retornam uma grande quantidade de registros, uma abordagem comum dos desenvolvedores é dividir os registros em páginas ou simplesmente retornar somente os registros mais recentes. Então, quando o ORM necessita criar uma consulta paginada ou trazendo somente os primeiros registros resultantes da consulta, o *Driver* possui as informações necessárias para o ORM montar essa consulta.

## Criação das Tabelas

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.

O *namespace*, na definição mais ampla, é uma forma de encapsular itens, por exemplo, em qualquer sistema operacional, os diretórios servem para agrupar arquivos relacionados, e atuam como um *namespace* para os arquivos, então, no sistema operacional é possível dois arquivos com o mesmo nome coexistirem desde que estejam em pastas diferentes. Esse mesmo princípio se estende a *namespaces* no mundo da programação. *Namespace* no PHP fornece uma maneira de agrupar classes, interfaces, funções e constantes relacionadas.

Então, para configurar o ORM para criar as tabelas, segue o exemplo:

Figura 8: Exemplo para configurar a criação de tabelas

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

Fonte: Autor, 2018

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

Figura 9: Exemplo para configurar para apagar as tabelas

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

Fonte: Autor, 2018

O ORM possui todas as informações necessárias para criar as tabelas usando o mapeamento definido nas classes do modelo em conjunto com as informações sobre o banco de dados contidas no *Driver*, como os tipos de dados e como fazer as chaves primárias com valores auto gerados.

Porém, criar e apagar as tabelas no banco de dados não é uma tarefa muito trivial, ela precisa ser executada em ordem.

Para executar a ação de criar, o ORM precisa ler o mapeamento de todas as classes do modelo, e identificar os atributos e os relacionamentos de cada classe. Com isso. de forma recursiva, o ORM irá criar o comando de criação de tabela, organizando hierarquicamente, partindo da tabela que não possui nenhuma referência para outras tabelas até a tabela que possui mais referências. Uma vez que a lista de tabelas esteja completa, o ORM irá executar os comandos na ordem em que foram criados.

A ação de apagar as tabelas do banco de dados funciona de forma semelhante, porém, para executar essa tarefa, o ORM irá começar a criar os comandos de apagar uma tabela, também organizando de maneira hierárquica, mas nesse caso, partindo da tabela que possui maior número de referências apontando para ela até a tabela que não possui nenhuma tabela fazendo referência para ela. Então, mais uma vez, o ORM irá executar os comandos na ordem em que foram criados.

O ORM permite 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:

Figura 10: Exemplo para executar ações antes de apagar as tabelas e depois de criá-las

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. ‘beforeCreate’ => [ $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 mesmas.

# Codificação

Toda a parte de codificação será anexa na mídia física entregue junto a esta documentação.

O código fonte relacionado ao *framework* Lumberjack’s ORM pode ser encontrado no seguinte repositório do GitHub https://github.com/dfrancklin/orm/

# Considerações Finais

Em desenvolvimento de software, quando um novo projeto é iniciado, é necessário pensar no futuro, pois um dos processos mais caros nos dias de hoje na área, depois da correção de erros, é a manutenção de um software.

Existe ainda uma preocupação extra no que diz respeito a escolha de um banco de dados, pois em uma situação onde o banco de dados não atende as necessidades do sistema, é preciso substituí-lo por um melhor. Porém, sem a utilização de um framework ORM, as aplicações ficam completamente dependentes de um único banco de dados e qualquer tarefa de melhoria ou manutenção pode se tornar muito mais complicada.

No mercado, atualmente, existem opções de framework ORM em PHP, entretanto, a maioria deles tem problemas com documentação ou não atendem as necessidades do sistema, principalmente na questão de mapeamentos N para N, e ainda, não oferecem a possibilidade de criação de tabelas a partir das classes do modelo.

Sendo assim, o objetivo deste trabalho foi atingido com o desenvolvimento de um framework ORM utilizando a linguagem PHP, um framework ORM que possa, de maneira simples, ser utilizado para mapear modelos de dados complexos, suportando todos os tipos de relacionamentos de tabelas do banco de dados e, criar as tabelas no banco de dados seguindo as classes do modelo de dados da aplicação.

Com a facilidade de modificar somente a conexão do banco de dados que o framework ORM utiliza, é possível trocar para outro banco de dados, completamente novo, sem que haja necessidade de adaptar qualquer trecho de código produzido na aplicação. E com apenas algumas linhas, ter a habilidade de criar as tabelas do banco de dados e fazer com que o ORM crie todas as tabelas necessárias para a aplicação. Tendo ainda a possibilidade de executar ações antes e depois da criação das tabelas, por exemplo, para incluir dados em tabelas de configuração ou para backup.

# Referências

FAYAD, M; SCMIDT, D; JOHNSON, R. **Building Applications Frameworks.** John Willey, 1999.

FOWLER, Martin. **UML Essencial: Um breve guia para a linguagem-padrão de modelagem de objetos.** 3. ed. Porto Alegre, Bookman, 2005.

FOWLER, Martin. **Active Record.** Disponível em: <https://www.martinfowler.com/eaaCatalog/activeRecord.html>. Acesso em: 30 de abril de 2018.

FOWLER, Martin. **Data Mapper.** Disponível em: <https://martinfowler.com/eaaCatalog/dataMapper.html>. Acesso em: 30 de abril de 2018.

FOWLER, Martin. **Lazy Load**. Disponível em: <https://martinfowler.com/eaaCatalog/lazyLoad.html>. Acesso em: 30 de abril de 2018.

GAMMA, E; HELM, R; JOHNSON, R; VLISSIDES, J. **Design Patterns – Elements of Reusable Object-Oriented Software.** Addison-Wesley, 1995.

JONES, Meilir Page. **Fundamentos do desenho Orientado a Objetos com UML.** São Paulo, Makron Books, 2001.

KOENIG, Andrew. **AntiPattern.** Disponível em: <http://wiki.c2.com/?AntiPattern>. Acesso em: 30 de abril de 2018.

ALEXANDER, Christopher. et al. **A Pattern Language.** Oxford University Press, New York, 1977.

GUERRA, Eduardo. **Design Pattern com Java: Projeto orientado a objetos guiado por padrões.** Casa do Código, 2012.

PHP. **What is PHP?** Disponível em <http://php.net/manual/en/intro-whatis.php>. Acesso em: 30 de abril de 2018.

PHP. **Função preg\_match.** Disponível em <http://php.net/manual/en/function.preg-match.php>. Acesso em: 30 de abril de 2018.

PHP. **Função preg\_match\_all.** Disponível em <http://php.net/manual/en/function.preg-match-all.php>. Acesso em: 30 de abril de 2018.

PHP. **Pattern Syntax.** Disponível em: <http://php.net/manual/pt\_BR/reference.pcre.pattern.syntax.php>. Acesso em: 30 de abril de 2018.

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

Bauer, C; King, G. **Hibernate in Action.** Manning Publications, 2004.

Dohms, Rafael. **Annotations in PHP: They Exist.** Disponível em <https://pt.slideshare.net/rdohms/annotations-in-php-they-exist>. Acesso em: 30 de abril de 2018.

# Glossário

**Abstração:** é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em modelagem orientada a objetos, uma classe é uma abstração de entidades existentes no domínio do sistema de software.

**Active Record:** Um *Desgin Pattern*, nomeado pela primeira vez por Martin Fowler.

**Annotation:** É um recurso para adicionar metadados que podem ser posteriormente interpretadas que irá realizar ou influenciar em alguma tarefa pré-definida.

**Anti-Pattern:** Um *Desgin Pattern* que pode ser comumente utilizado, porém é ineficiente e/ou contra-produtivo em prática.

**Classe Abstrata:** Serve como um modelo para outras classes que dela herdam, e especializam e implementam métodos da classe abstrata.

**Data Mapper:** Um *Desgin Pattern*, nomeado pela primeira vez por Martin Fowler.

**Design Pattern:** Em Engenharia de Software, um *Design Pattern* (ou em português “padrão de projeto”) é uma solução geral para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software.

**Driver:** Contém as informações e funções a serem integradas a um sistema para controlar ou integrar com um determinado sistema ou periférico. No caso, permite que o *framework* se comunique corretamente com diferentes bancos de dados.

**Framework:** Em desenvolvimento de software, é uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica.

**Git:** Um sistema de controle de versão distribuído e um sistema de gerenciamento de código-fonte.

**GitHub:** Uma plataforma de hospedagem de código-fonte com controle de versão usando o *Git*.

**Herança:** Um princípio da orientação a objetos, permite que atributos e métodos de classes sejam compartilhados com outras classes. A herança é usada para reaproveitamento de códigos ou especializar operações.

**Interface:** É um contrato que define um determinado conjunto de métodos que devem ser implementados nas classes que assinam esse contrato.

**Metadados:** São dados sobre outros dados. Um metadado pode indicar informações ou características sobre outro dado.

**Open Source:** é um termo em inglês que significa código aberto. Isso diz respeito ao código-fonte de um software, que pode ser adaptado para diferentes fins.

**ORM:** Sigla em inglês que significa *Objeto-Relational Mapping*, ou em português “Mapeamento Objeto-Relacional”

**PHP:** Sigla em inglês *Hypertext Preprocessor*, é uma linguagem interpretada, usada originalmente apenas para o desenvolvimento de aplicações web, atuante no lado do servidor.

**Polimorfismo:** Do grego que quer dizer várias formas. É uma característica da orientação a objetos que permite que uma classe mais genérica represente uma classe mais específicas que herdam ou implementam a classe mais genérica, assim é possível que vários tipos sejam tratados da mesma maneira.

**Reflection:** em português significa Reflexão, é o processo em que um programa pode observar e modificar sua própria estrutura, no PHP este recurso possibilita realizar engenharia reversa em classes, interfaces, funções e extensões, além disso também podemos resgatar comentários/documentação de funções, classes e métodos.

**Script:** Ou **Linguagem de Script**, é uma linguagem de programação que é interpretada. Um script é uma séria de instruções em código escritas para serem seguidas de maneira sequencial para realizar uma tarefa computacional.

**SQL:** Sigla em inglês que significa *Structured Query Language*, ou em português “Linguagem de Consulta Estruturada”, é a linguagem de pesquisa declarativa padrão para banco de dados relacional.

**View****:** Pode ser definida como uma tabela virtual composta por linhas e colunas de dados vindos de várias tabelas relacionadas.

# Anexo – Manual do Desenvolvedor

1. **Introdução**

Lumberjack’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 Lumberjack’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.

Este manual foi escrito para auxiliar o programador a trabalhar com o Lumberjack’s ORM e os exemplos nele descritos foram criados durante o desenvolvimento do projeto para melhor descrever a situação e fornecer uma maneira prática para o uso do ORM.

1. **Instalação**

Para usar o Lumberjack’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”:

Figura 1: Incluir o ORM ao projeto

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 ser substituído da seguinte maneira:

Figura 2: Definindo qual arquivo de conexões o ORM utilizará

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.

Figura 3: Definindo arquivo de conexões

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 banco 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:

Figura 4: Definindo conexões que o ORM poderá utilizar

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*:

Figura 5: Definindo conexões que o ORM poderá utilizar

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:

Figura 6: Definindo criação de tabelas no ORM

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:

Figura 7: Definindo criação e deleção de tabelas no ORM

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 ao ORM quais ações ele deve executar, basta fazer o seguinte:

Figura 8: Definindo ação para executar antes criação e da deleção de tabelas no ORM

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 criada uma instância da classe “App\Helpers\InitDatabase” e nas linhas 9 e 10, são informados, 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:

Figura 9: Definindo ação para executar antes criação e da deleção de tabelas no ORM

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:

Figura 10: Definindo ação para executar antes criação e da deleção de tabelas no ORM

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

Fonte: Autor, 2018

O ORM passa 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 à 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 das *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”. Normalmente utilizado para mapear uma *view*.

**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 Relacionamentos**
* ***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:

Figura 11: Exemplo de classe do modelo mapeada

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, de acordo com a Figura 11, 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”:

Figura 12: Exemplo de relacionamento “um para um”

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, de acordo com a Figura 12, 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:

Figura 13: Exemplo de relacionamento “um para um” (outra ponta)

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, de acordo com a Figura 13, 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”:

Figura 14: Exemplo de relacionamento “um para muitos”

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, de acordo com a Figura 14, 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:

Figura 15: Exemplo de relacionamento “um para muitos” (outra ponta)

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, de acordo com a Figura 15, 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”:

Figura 16: Exemplo de relacionamento “muitos para muitos”

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, de acordo com a Figura 16, 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 **“Projeto”** 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:

Figura 17: Exemplo de relacionamento “muitos para muitos” (outra ponta)

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

O exemplo acima, de acordo com a Figura 17, exibe o mapeamento da classe **“Projeto”**, ela representa a tabela “projeto” e possui o atributo **“$empregados”**, que por sua vez, mapeia a outra ponta o relacionamento do tipo “muitos para muitos” com a classe **“Empregado”** e indica que a tabela de ligação é mapeada pelo atributo **“$projetos”**.

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:

Figura 18: Exemplo de como obter uma instância do EntityManager

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, de acordo com a Figura 18, a conexão “exemplo-mysql”.

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

Figura 19: Exemplo de como obter uma instância do EntityManager com conexão segundária

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 realizadas 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. ***BeginTransaction***

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

Figura 20: Exemplo de como iniciar uma trasação

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.

Figura 21: Exemplo de como aplicar alterações de uma trasação

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.

Figura 22: Exemplo de como desfazer alterações de uma trasação

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.

Figura 23: Exemplo de como carregar registro por chave primária

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*).

Figura 24: Exemplo de como listar os registro de uma dada tabela

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

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. ***QueryBuilder***

Para criar consultas, desde as simples até as complexas, o desenvolvedor pode optar por usar o *QueryBuilder*.

* + 1. **Obtendo o *QueryBuilder***

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

Figura 25: Exemplo de como obter uma instância do QueryBuilder

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. $em = $orm->createEntityManager();
5. $q1=$orm->createQuery();
6. $q2=$orm->createQuery(App\Models\*Empregado*::class,‘u’);
7. ?>

Fonte: Autor, 2018

O método *createQuery* retorna uma instância do *QueryBuilder* e pode ser utilizado de duas formas, na primeira, basta chamar o método sem passar parâmetro algum e ele irá retornar uma instância sem informação alguma, da segunda maneira, o desenvolvedor pode passar a classe que mapeia a tabela a ser consultada e um *alias* para a tabela. O *alias* é um apelido que pode ser atribuído para uma tabela em uma consulta, é normalmente utilizado com reduzir o nome de tabelas e para realizar consultas com *Self* *Join* (consultas que fazem ligação de uma tabela com ela mesma).

* + 1. **Criando uma consulta simples usando *list***

Para realizar uma consulta simples, o desenvolvedor pode fazer da seguinte maneira:

Figura 26: Exemplo de como listar os registro de uma dada tabela usando o QueryBuilder

1. <?php
2. use App\Models\*Empregado*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’);
5. $em = $orm->createEntityManager();
6. $query = $orm->createQuery();
7. $query->from(*Empregado*::class);
8. $emprs = $query->list();
9. ?>

Fonte: Autor, 2018

Ou:

Figura 27: Exemplo de como listar os registro de uma dada tabela usando o QueryBuilder

1. <?php
2. use App\Models\*Empregado*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’);
5. $em = $orm->createEntityManager();
6. $query = $orm->createQuery(*Empregado*::class);
7. $emprs = $query->list();
8. ?>

Fonte: Autor, 2018

Ou ainda:

Figura 28: Exemplo de como listar os registro de uma dada tabela usando o QueryBuilder

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

Fonte: Autor, 2018

A consulta ilustrada acima, de acordo com a Figura 28, irá retornar todos os registros da tabela mapeada pela classe “Empregado” em um *array* com instâncias da mesma classe, onde cada instância representa um registro da tabela.

Para tabelas que possuem um número muito grande de registros, recomenda-se paginar ou filtrar os registros.

* + 1. **Criando uma consulta simples usando *one***

Além do método *list* o *QueryBuilder* possui o método *one*, que nesse caso, retorna apenas um registro mapeado em uma instância da classe indicada no método *from*. Por exemplo:

Figura 29: Exemplo de como carregar um registro de uma dada tabela usando o QueryBuilder

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

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 29, irá retornar um registro da tabela mapeada pela classe “Empregado”, apenas o primeiro registro retornado pela consulta.

* + 1. ***Join***

Para realizar uma consulta com múltiplas tabelas, o desenvolvedor pode fazer da seguinte maneira:

Figura 30: Exemplo de como realizar uma consulta com múltiplas tabelas

1. <?php
2. use App\Models\*Empregado*;
3. use App\Models\*Projeto*;
4. $orm = ORM\*Orm*::getInstance();
5. $orm->setConnection(‘exemplo-mysql’);
6. $em = $orm->createEntityManager();
7. $query = $orm->createQuery();
8. $query->from(*Empregado*::class, ‘e’);
9. $query->join(*Projeto*::class, ‘p’);
10. $emprs = $query->list();
11. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 30, será executada com as tabelas mapeadas pelas classes “Empregado” e “Projeto”. O resultado será todos os registros da tabela mapeada pela classe “Empregado” (informado no método *from*) em um *array* com instâncias da mesma classe, onde cada instância representa um registro da tabela.

O método *join* pode ser chamado quantas vezes forem necessárias, porém, o desenvolvedor deve ter em mente que, quanto mais tabelas forem adicionadas à consulta, mais a consulta ficará pesada para ser executada.

Para fazer uma consulta usando o chamado *Outer Join*, o desenvolvedor pode passar como terceiro parâmetro o tipo de *join* (*left join* ou *right join*) que deve ser usado, por exemplo, para realizar um *left join*:

Figura 31: Exemplo de como realizar uma consulta com múltiplas tabelas usando OUTER JOIN

1. <?php
2. use App\Models\*Empregado*;
3. use App\Models\*Projeto*;
4. use ORM\Contants\*JoinTypes*;
5. $orm = ORM\*Orm*::getInstance();
6. $orm->setConnection(‘exemplo-mysql’);
7. $em = $orm->createEntityManager();
8. $query = $orm->createQuery();
9. $query->from(*Empregado*::class, ‘e’);
10. $query->join(*Projeto*::class, ‘p’, *JoinTypes*::LEFT);
11. $emprs = $query->list();
12. ?>

Fonte: Autor, 2018

Caso o tipo de *join* não for informado, o valor padrão assumido é *INNER.*

Observação: o método *join* não implica que a tabela informada como parâmetro nesse método será carregada juntamente com a tabela informada como parâmetro no método *from*. Essa ação é conhecida como *Eager Load*, porém o ORM atualmente suporta apenas *Lazy Load*, ou seja, os registros dos relacionamentos serão carregados somente quando necessário (sob demanda).

* + 1. ***Where***

Para realizar uma consulta com utilizando filtro, o desenvolvedor pode fazer da seguinte maneira:

Figura 32: Exemplo de como realizar consultas com filtros

1. <?php
2. use App\Models\*Empregado*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’);
5. $em = $orm->createEntityManager();
6. $query = $orm->createQuery(*Empregado*::class, ‘e’);
7. $query->where(‘e.nome’)->equals(‘João da Silva’);
8. $emprs = $query->list();
9. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 32, irá retornar um *array* contendo todos os registros da tabela mapeada pela classe “Empregado” cuja coluna “nome” possua o valor “João da Silva”.

Utilizar mais de um filtro, pode ser feito através do método *and* ou *or*:

Figura 33: Exemplo de como realizar consultas com filtros

1. <?php
2. use App\Models\*Empregado*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’);
5. $em = $orm->createEntityManager();
6. $query = $orm->createQuery(*Empregado*::class, ‘e’);
7. $query->where(‘e.nome’)->equals(‘João da Silva’);
8. $query->or(‘e.nome’)->equals(‘Predo da Silva’);
9. $query->and(‘e.dataNasc’)->equals(new *DateTime*(‘1990-05-23’));
10. $emprs = $query->list();
11. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 33, irá retornar um *array* contendo todos os registros da tabela mapeada pela classe “Empregado” cuja coluna “nome” possua o valor “João da Silva” ou o valor “Pedro da Silva” e a coluna “dataNasc” seja igual à data “23/05/2018”.

Para realizar uma consulta com filtro e múltiplas tabelas, o desenvolvedor pode fazer da seguinte maneira:

Figura 34: Exemplo de como realizar consultas com filtros e multiplas tabelas

1. <?php
2. use App\Models\*Empregado*;
3. use App\Models\*Projeto*;
4. $orm = ORM\*Orm*::getInstance();
5. $orm->setConnection(‘exemplo-mysql’);
6. $em = $orm->createEntityManager();
7. $query = $orm->createQuery();
8. $query->from(*Empregado*::class, ‘e’);
9. $query->join(*Projeto*::class, ‘p’);
10. $query->where(‘e.nome’)->equals(‘João da Silva’);
11. $query->and(‘p.nome’)->equals(‘PRJ\_OZOB’);
12. $emprs = $query->list();
13. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 34, irá retornar um *array* contendo todos os registros da tabela mapeada pela classe “Empregado” cuja coluna “nome” possua o valor “João da Silva” e a coluna “nome” da tabela mapeada pela classe “Projeto” possua o valor “PRJ\_OZOB”.

Os métodos *where*, *and* e *or* esperam receber por parâmetro um texto que indica o *alias* utilizado pela classe e qual atributo da classe deve ser utilizado para o filtro. O retorno desses métodos é um objeto que contém todas as operações de comparação que podem ser utilizados no filtro.

As operações permitidas são:

| **Operação** | **Parâmetros** | **Atalho** | **Descrição** |
| --- | --- | --- | --- |
| equals | $valor | eq | Compara se os valores são iguais |
| notEquals | $valor | neq | Compara se os valores são diferentes |
| isNull | - | isn | Compara se o valor da coluna é nulo |
| isNotNull | - | isnn | Compara se o valor da coluna não é nulo |
| between | $inicial, $final | bt | Compara se o valor da coluna está entre o valor $inicial e o valor $final |
| notBetween | $inicial, $final | nbt | Compara se o valor da coluna não está entre o valor $inicial e o valor $final |
| greaterThan | $valor | gt | Compara se o valor da coluna é maior do que o $valor |
| greaterOrEqualsThan | $valor | goet | Compara se o valor da coluna é maior ou igual ao $valor |
| lessThan | $valor | lt | Compara se o valor da coluna é menor do que o $valor |
| lessOrEqualsThan | $valor | loet | Compara se o valor da coluna é menor ou igual ao $valor |
| in | *Array* $valores | - | Compara se o valor da coluna está dentro dos $valores do *array* |
| notIn | *Array* $valores | - | Compara se o valor da coluna não está dentro dos $valores do *array* |
| like | $valor | lk | Compara o valor da coluna é compatível com $valor informado. O valor informado pode utilizar o “%” como coringa em qualquer posição, por exemplo, no começo do $valor “%da Silva” indica que o valor da coluna deve terminar com o trecho “da Silva” |
| notLike | $valor | nlk | Compara o valor da coluna não é compatível com $valor informado. O valor informado pode utilizar o “%” como coringa em qualquer posição, por exemplo, no começo do $valor “%da Silva” indica que o valor da coluna deve terminar com o trecho “da Silva” |
| contains | $valor | ctn | Compara se o $valor está contido no valor da coluna. |
| notContains | $valor | nctn | Compara se o $valor não está contido no valor da coluna. |
| beginsWith | $valor | bwt | Compara se o valor da coluna começa com $valor. |
| notBeginsWith | $valor | nbwt | Compara se o valor da coluna não começa com $valor. |
| endsWith | $valor | ewt | Compara se o valor da coluna termina com $valor. |
| notEndsWith | $valor | newt | Compara se o valor da coluna não termina com $valor. |

* + 1. **Paginação**

A paginação é um recurso muito útil para o desenvolvedor, ele permite que os registros sejam divididos em páginas menores para serem listados em tela para o usuário, caso contrário, todos os registros que existem na tabela seriam exibidos de uma só vez.

Isso não parece assim tão mal, porém, em um caso onde a tabela possua milhares ou mesmo centenas de milhares de registros, levaria muito tempo para carregar tudo, ou o sistema iria travar, ou ainda resultaria em erro.

A paginação serve para resolver esse tipo de problema, e pode ser utilizada das seguintes maneiras:

Figura 35: Exemplo de como realizar consultas com paginação

1. <?php
2. use App\Models\*Empregado*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’);
5. $em = $orm->createEntityManager();
6. $emprs = $em->list(*Empregado*::class, 2, 10);
7. ?>

Fonte: Autor, 2018

Figura 36: Exemplo de como realizar consultas com paginação usando QueryBuilder

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

Fonte: Autor, 2018

Ambas as consultas realizam uma consulta na tabela mapeada pela classe “Empregado”. O ORM irá gerar uma consulta onde ele divide os registros da tabela em páginas, contendo 10 registros por página, então irá retornar um *array* contendo os registros da segunda página.

Uma maneira alternativa para recuperar a primeira página seria o método *top*:

Figura 37: Exemplo de como realizar consultas com paginação usando top

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

Fonte: Autor, 2018

As queries acima, de acordo com a Figura 37, realizam uma consulta na tabela mapeada pela classe “Empregado” muito semelhante às anteriores. O ORM irá gerar uma consulta onde ele irá retornar um *array* contendo 10 registros, que seria equivalente à primeira página.

* + 1. **OrderBy**

O método *orderBy* define a ordem em que os registros devem ser retornados na consulta.

Figura 38: Exemplo de como realizar consultas ordenadas

1. <?php
2. use App\Models\*Empregado*;
3. use ORM\Constants\*OrderTypes*;
4. $orm = ORM\*Orm*::getInstance();
5. $orm->setConnection(‘exemplo-mysql’);
6. $em = $orm->createEntityManager();
7. $query = $em->createQuery(*Empregado*::class);
8. $query->orderBy(‘e.dataNasc’*, OrderTypes*::DESC);
9. $query->orderBy(‘e.nome’*, OrderTypes*::ASC);
10. $emprs = $query->list();
11. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 38, irá retornar todos os registros da tabela mapeada pela classe “Empregado” em um *array* com instâncias da mesma classe, onde cada instância representa um registro da tabela, ordenados pelas colunas “dataNasc” em ordem decrescente e “nome” em ordem crescente.

O método *orderBy* é acumulativo, pode ser chamado quantas vezes forem necessárias. Caso o segundo parâmetro não for informado, o ORM irá usar a ordem padrão que é *ASC (ascendente)*.

* + 1. ***GroupBy***

O método *groupBy* define um agrupamento para criar relatórios e totalizações separados em grupos. A instrução de agrupamento é frequentemente usada com funções de agregação para agrupar o conjunto de resultados em uma ou mais colunas.

Figura 39: Exemplo de como realizar consultas agrupadas

1. <?php
2. use App\Models\*Empregado*;
3. use ORM\Constants\*OrderTypes*;
4. $orm = ORM\*Orm*::getInstance();
5. $orm->setConnection(‘exemplo-mysql’);
6. $em = $orm->createEntityManager();
7. $query = $em->createQuery(*Empregado*::class);
8. $query->count(‘e.id’*,* ‘total’);
9. $query->groupBy(‘e.dataNasc’);
10. $emprs = $query->list();
11. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 39, irá retornar em um *array* contendo o resultado da consulta, onde cada entrada do resultado é um *array* que possui os campos “total” e “dataNasc”. O valor de “total” representa uma contagem de quantos “Empregados” nasceram no mesmo dia representado pelo valor de “dataNasc”.

O método *groupBy* pode receber por parâmetro quantos parâmetros forem necessários, um após o outro, na mesma chamada do método.

* + 1. **Agregação**

Uma função de agregação executa um cálculo em um conjunto de valores e retorna um único valor. As funções de agregação frequentemente são usadas com a cláusula *GROUP BY* em uma consulta.

Figura 40: Exemplo de como realizar consultas com funcões de agregação

1. <?php
2. use App\Models\*Empregado*;
3. use ORM\Constants\*OrderTypes*;
4. $orm = ORM\*Orm*::getInstance();
5. $orm->setConnection(‘exemplo-mysql’);
6. $em = $orm->createEntityManager();
7. $query = $em->createQuery(*Empregado*::class);
8. $query->count(‘e.id’*,* ‘total’);
9. $query->max(‘e.salario’*,* ‘maiorSalario’);
10. $query->groupBy(‘e.dataNasc’);
11. $emprs = $query->list();
12. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 40, irá retornar em um *array* contendo o resultado da consulta, onde cada entrada do resultado é um *array* que possui os campos “total”, “maiorSalario” e “dataNasc”. O valor de “total” representa uma contagem de quantos “Empregados” nasceram no mesmo dia representado pelo valor de “dataNasc”, e o valor de “maiorSalario” representa o maior salário que um “Empregado”, que nasceu nesse dia, recebe.

O segundo parâmetro do método de agregação, por exemplo, o count, é um *alias* que o desenvolvedor pode atribuir para o campo para facilitar na utilização do mesmo.

As funções de agregação permitidas são:

| **Operação** | **Parâmetros** | **Descrição** |
| --- | --- | --- |
| avg | $coluna, $alias | Calcula a média dos valores da $coluna |
| sum | $coluna, $alias | Calcula a soma dos valores da $coluna |
| min | $coluna, $alias | Retorna o menor valor da $coluna |
| max | $coluna, $alias | Retorna o maior valor da $coluna |
| count | $coluna, $alias | Calcula a quantidade dos valores da $coluna |

* + 1. ***Having***

O método *having* especifica um critério de filtro utilizando uma função de agregação, pois a clausula *where* não consegue realizar esse tipo de filtro. Na maioria dos bancos de dados, é obrigatório usado com a cláusula *GROUP BY* antes de usar o *having*.

Figura 41: Exemplo de como realizar consultas com having

1. <?php
2. use App\Models\*Empregado*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’);
5. $em = $orm->createEntityManager();
6. $query = $em->createQuery(*Empregado*::class);
7. $query->groupBy(‘e.dataNasc’);
8. $query->having()->avg(‘e.salario’)->gt(5000)
9. ->and()->count(‘e.id’)->gt(5);
10. $emprs = $query->list();
11. ?>

Fonte: Autor, 2018

A consulta acima, de acordo com a Figura 41, irá retornar uma lista contendo a “dataNasc” da tabela mapeada pela classe “Empregado”, filtrando pelos registros cujo salário seja maior do que a média, e que a quantidade de registros agrupados pela data seja maior do que 5.

As funções de agregação são as mesmas das descritas na seção 5.5.10 Having e os métodos de comparação são os mesmos dos descritos na seção 5.5.5 Where com exceção dos métodos: in, notIn, like, notLike, contains, notContains, beginsWith, notBeginsWith, endsWith, notEndsWith.

* 1. ***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.

Figura 42: Exemplo de como utilizar o método save para incluir registro

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 *DateTime*(‘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, de acordo com a Figura 42, irá resultar em um novo registro na tabela “empregado” . O método *save* irá retornar o novo registro.

Figura 43: Exemplo de como utilizar o método save para alterar registro

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

Fonte: Autor, 2018

O exemplo acima, de acordo com a Figura 43, 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.

Figura 44: Exemplo de como utilizar o método remove para deletar 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. $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, de acordo com a Figura 44, irá remover um registro já existente na tabela “empregado” com a chave primária de valor “1”. O método *remove* irá devolver a quantidade de linhas afetadas com essa operação.

1. **Logger**

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

Para recuperar a instância do *Logger* a ser configurado, basta chamar o método *getLogger* do ORM.

O método *setTemplate* define um padrão para as mensagens que serão gravadas no arquivo de *log*. O padrão é "[{datetime}][{level}] {class} - {message}\n".

O método *setLocation* define a localização onde os arquivos do *log* serão salvos. O valor padrão é a pasta do ORM.

O método *setFilename* define o prefixo que será utilizado para criar o nome do arquivo de *log*. O valor padrão é “orm”.

O método *setLevel* define qual o nível de *log* o ORM deve registrar, os valores esperados são Logger::LEVEL\_ALL, Logger::LEVEL\_DEBUG, Logger::LEVEL\_INFO, Logger::LEVEL\_WARN, Logger::LEVEL\_ERROR, Logger::LEVEL\_FATAL ou Logger::LEVEL\_OFF. O valor padrão é Logger::LEVEL\_ERROR.

O método *setOccurrency* 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 Logger::OCCURRENCY\_DAILY ou Logger::OCCURRENCY\_MONTHLY. O valor padrão é Logger::OCCURRENCY\_DAILY.

O método *setDisable* 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 *Logger* do ORM pode ser utilizado em outras partes da aplicação. Para isso basta usar os métodos *debug*, *info*, *warn* *error* e *fatal* 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.

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*:

Figura 45: Exemplo de como utilizar o método save para alterar registro

1. <?php
2. use ORM\Contants\*GeneratedTypes;*
3. use ORM\Core\*Driver;*
4. if (!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 = “%s \nLIMIT %d, %d”;
15. $this->TOP\_TEMPLATE = “%s \nLIMIT %d”;
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