1. Introdução ao Framework EngineDB v0.3

O Engine é um framework de persistência para PHP distribuído sob a licença GPL[1] que pretende oferecer persistência ORM com o máximo de simplicidade de forma menos obstrusiva possível, utilizando Annotations (semelhantes ao padrão JPA). Ele utiliza-se de técnicas de reflexão (funcionalidade incorporada ao PHP5) para acessar os objetos, suas propriedades e métodos, possibilitando assim aos seus usuários efetuar sua persistência sem precisar estender nenhuma classe do framework.

O EngineDB visa abstrair ao máximo a camada de dados do modelo dos objetos de uma aplicação. Não importa o mecanismo de persistência de dados utilizado: MySQL ou PostgreSQL. Através de algumas poucas anotações feitas no corpo da classe que deseja-se persistente, o framework é capaz de mapeá-la e adequá-la às várias estruturas de persistência, de acordo com a preferência do usuário. Atualmente, o Engine funciona somente com o SGBD MySQL e o PostgreSQL, mas possui como prioridade muito em breve dar suporte a XML e também aos bancos MSSQL e DB2.

2. Instalação e Configuração

O Engine é distribuído através de um pacote contido em um arquivo compactado (engine.rar). Como padrão, presumiremos que o diretório /lib será criado na raiz de seu servidor e este diretório irá receber o conteúdo do arquivo engine.rar. Para adicionar o Engine à sua aplicação, você deve então incluir o arquivo engine.db.php, que importa todas as classes e interfaces do framework que você precisa para utilizá-lo. Para fazê-lo, inclua as seguintes linhas de código:

```php
<?php
 //Incluir os arquivos do framework
 require_once("lib/engine/engine.db.php");
 ?>
```

[1] Disponível em http://www.gnu.org/copyleft/gpl.txt

Na utilização standalone, a única configuração a ser feita é informar qual o banco utilizado e fornecer os parâmetros da conexão (servidor do banco, nome da base, login e senha). Para configurar o banco, inclua as seguintes linhas de código:

```
<?php
 DAOFactory::addDSN('mysql', DbDriverFactory::getDriver(DbDriver::MYSQL)
       ->configure('<host>','<base_de_dados>','<usuário>','<senha>')
 );
 ?>
```

É possível criar mais de uma conexão, basta adicionar DSNs via chamada ao método `DAOFactory::addDSN`. Por padrão a primeira conexão criada é marcada como a conexão padrão obtida via chamada ao método `DAOFactory::getDAO()`. Para definir outra conexão como a padrão o método `DAOFactory::setDefaultDSN('<nome_da_conexao>');` deverá ser utilizado.

Para persistir seus objetos, o usuário deve utilizar um DAO (Data Access Object), que é um objeto de acesso a dados. É através dele que o usuário salva um novo objeto, atualiza-o ou ainda o apaga. Para obter o DAO padrão, utilize a seguinte chamada:

```
<?php
 DAOFactory::getDAO();
 ?>
```

Alternativamente você poderá obter um DAO por intermédio do DSN fornecido no momento da sua criação através da chamada:

```
<?php
 DAOFactory::getDAOByDSN('<nome_da_conexao>');
 ?>
```

A partir de então, já podemos salvar novos objetos, atualizá-los, apagá-los, etc. Para ilustrar, criamos um objeto `Cliente` e definimos seu id (atributo identificador).

```
<?php
 //Instanciar um objeto Cliente
 $cliente = new Cliente();
 //Definir valor da chave primária
 $cliente->setId(1);
 ?>
```

Ao utilizar o método `load()` do DAO, carregamos o cliente instanciado anteriormente, cujo atributo identificador é “1”.

```
<?php
 //carregar o objeto Cliente (id = 1)
 DAOFactory::getDAO()->load($cliente);
 ?>
```

Os outros dois métodos largamente utilizados para a persistência de objetos são os métodos `save()` e o `delete()`.

O método `save()`, caso o id do objeto que se deseja persistir esteja definido (e seja um id válido, obviamente), atualiza este objeto (realiza uma operação SQL UPDATE caso a persistência aconteça em um SGBD). Caso o atributo identificador do objeto que deseja-se salvar não esteja setado, uma nova entrada é criada e persistida (operação SQL INSERT caso a persistência ocorra em um SGBD).

O método `delete()` apaga um objeto, que deve ter um id válido definido.

3. Criando suas Entidades e Relacionamentos

Um diagrama de classes é um diagrama no qual estruturamos as classes de um sistema e seus relacionamentos e comportamento estático. A maneira que o framework utiliza para entender e mapear as classes aos vários mecanismos de persistência é através das **anotações**. As anotações são diretivas especiais colocadas dentro de comentários de bloco[2] e que fornece informações acerca da persistência ao framework. Uma anotação sempre começa com uma arroba, seguido do nome da anotação (`@nome`). Uma anotação pode possuir parâmetros. Neste caso, deve-se abrir um parêntese logo após o fim do nome da anotação e enumerar os parâmetros no formato `nome="valor"`, separando-os por vírgula. Após declarar todos os parâmetros deve-se, obviamente, fechar o parêntese. O padrão camelCase é utilizado para a nomenclatura de nomes de anotação e também de seus parâmetros.

Anotações de Entidade ou anotações de Classe

Para que uma entidade seja reconhecida como persistente pela Engine, ela deve ser anotada com `@Entity` e opcionalmente com `@Table`[3].

```php
<?php
 /**
 * @Entity
 * @Table(name="cliente")
 */
 class Cliente {
 ...
 }
 ?>
```

[2] Um comentário de bloco, no PHP, é todo o texto entre uma barra seguida de dois asteriscos (/**) e um asterisco seguido de uma barra (*/).
[3] Caso a tabela e a classe que se deseja vincular possuam exatamente o mesmo nome (*case sensitive*), a anotação @Table não é necessária: o framework realiza a vinculação automaticamente.

A anotação `@Entity` diz ao framework que a classe `Cliente` é persistente. Caso a anotação `@Table` seja omitida, o framework assumirá o nome da classe como nome de sua tabela (Case Sensitive). A anotação `@Table` vincula a classe a uma tabela na base de dados. `@Table` possui um parâmetro `name`, cujo valor é o nome da tabela no banco. Neste caso, o nome da tabela vinculada à classe `Cliente` é `cliente`, de acordo com `@Table(name="cliente")`. Isto significa, na prática, que a classe `Cliente` utiliza a tabela indicada por `@Table` para persistir suas propriedades.

Anotações de propriedades e métodos

Anotações de chave primária

Para informar ao Engine que uma determinada propriedade da classe é sua chave primária, deve-se anotá-la com `@Id`. O engine suporta 3 tipos de chave primária.

```
@Id(strategy=GenerationType.NONE)
O engine não testará gerar chave, ela deverá estar setada na entidade.
@Id(strategy=GenerationType.AUTO) <padrão, quando não informado>
O engine utilizará a chave gerada automaticamente pelo banco
@Id(strategy=GenerationType.MAX)
O engine buscará pela próxima chave disponível na tabela
```

```php
<?php
 /**
  * @Id
  * @var int
  */
 private $id;
 ?>
```

Chaves primárias compostas

Na versão 0.3 o suporte a entidades com índices compostos foi adicionado bastando somente anotar como índice as propriedades que o sejam.

Chaves primárias estrangeiras

Nos casos onde a entidade possua uma outra entidade como chave primária (Somente nos relacionamentos One to One e Many To One) assim como um índice de tipo primitivo, basta adicionar a anotação @Id. Por padrão ao EngineDB usará o nome da coluna do índice da entidade índice, que poderá ser mudada com uma anotação @Column.

Chaves primárias estrangeiras compostas

Se a entidade índice possuir índice composto será necessário adicionar a anotação @CompositeColumn na propriedade para que o EngineDB saiba como persistir corretamente a entidade dependente.

```
@CompositeColumn(columns={@Column(name="id1"),@Column(name="id2"),...})
```

A ordem de declaração das colunas deve obedecer a ordem das propriedades na classe índice ou o EngineDB poderá apresentar um comportamento não esperado.

As entidades deverão possuir, para todos os seus campos persistentes, **métodos *getter* e *setter*[4] públicos correspondentes**. A utilização da anotação `@var <tipo>` para definir o tipo da variável é necessária quando os campos no banco são outros que não string. Por exemplo, uma variável que tem um campo do tipo INTEGER no banco de dados deverá possuir a anotação `@var int` em seu comentário de propriedade.

Anotações de Coluna

Para informar ao Engine que um determinado campo da classe deve ser vinculado a uma coluna no banco de dados, deve-se anotá-lo com `@Column`[5]. A anotação `@Column` possui um parâmetro `name`, que informa qual o nome da coluna no banco de dados ao qual o campo corresponde. Caso a anotação seja omitida o framework assumirá como nome da coluna o nome da propriedade.

Observação

O local onde colocar as anotações é flexível, de acordo com o tipo da anotação. As anotações de chave primária e coluna podem ser colocadas tanto no comentário do campo quanto no comentário do método *getter* correspondente ao campo.

```
<?php
 /**
  * @Id
  * @var int
  */
 private $id;
 ?>
```

A anotação `@Id` está no bloco de comentários do campo.

---

[4] Os métodos *getter* e *setter* são os métodos de acesso às propriedades de uma classe, que por definição devem ser privadas. As regras para sua nomenclatura são definidas em **http://www.write-me.com**

[5] Caso a coluna da tabela e o campo da classe que se deseja vincular possuam exatamente o mesmo nome (*case sensitive*), a anotação @Column não é necessária: o framework realiza a vinculação automaticamente.

```php
<?php
 /**
  * @Id
  * @Column(name="id")
  * @return int
  */
 public function getId() {
       return $this->id;
 }
 ?>
```

A anotação `@Id` está no bloco de comentários do método *getter* do campo. A anotação `@Id` funciona corretamente e tem o mesmo efeito nos dois casos.

Abaixo, segue uma lista com as principais anotações, que parâmetros elas aceitam, se são ou não obrigatórias, onde e para quê são utilizadas.

@Table (opcional) - define a tabela a qual a classe irá se associar no banco de dados. Caso ela não exista, o framework assumirá que o nome da classe e o nome da tabela no banco de dados são idênticos (case sensitive).

Possui os seguintes parâmetros:

- name (obrigatório) - nome da tabela no banco de dados e é interpretado como uma string.

Exemplo:

```php
<?php
 /**
  * @Entity
  * @Table(name="<nome_da_tabela>")
  */
 ?>
```

@Column (opcional) - define a qual coluna de uma tabela uma propriedade se associa no banco de dados. Caso esta anotação não exista, o framework assumirá que o nome do campo/propriedade e o nome da coluna no banco de dados são idênticos.

Possui os parâmetros:

- name (obrigatório) - nome da coluna na tabela do banco de dados.

Exemplo:

```php
<?php
 /**
  * @Column(name="<nome_da_coluna>")
  */
 ?>
```

@Transient (especial) - define uma propriedade da classe que não é persistente, ou seja, que não deve ser interpretada pelo Engine. Neste caso, convenciona-se chamar a propriedade como transiente. Vale lembrar que uma propriedade não-persistente deve ser obrigatoriamente anotada com `@Transient`. Esta anotação não possui parâmetros.

```php
<?php
 /**
  * @Entity
  * @Table(name="curso")
  */
 class Curso {

        /**
         * @Column(name="horas_duracao")
         * @var int
         */
        private $horasDuracao;

        /**
         * @Column(name="preco_hora")
         * @var float
         */
        private $precoPorHora;

        /**
         * @Transient
         * @var float
         */
        private $precoTotal;
        ...
 }
 ?>
```

No código acima, temos um exemplo de propriedade transiente. Intui-se que a propriedade `$precoTotal` é o cálculo do produto entre as propriedades `$horasDuracao` e `$precoPorHora`, e por isto não é necessário persisti-la. A propriedade `$precoTotal` é, então, anotada com `@Transient`.

@OneToOne (especial) - define um relacionamento 1x1 (um para um) entre duas entidades. É obrigatório caso deseje-se que a entidade anotada estabeleça um relacionamento 1x1 com outra entidade.

Possui os parâmetros:

- mappedBy (opcional) - caso a classe anotada com `@OneToOne` esteja vinculada à tabela que não possui a chave estrangeira do relacionamento 1x1 (um para um), o parâmetro `mappedBy` deve ser setado com o nome do campo da outra classe do relacionamento que representa o relacionamento.

- fetch (opcional) – determina a forma com que os dados trazidos do banco serão carregados no objeto. Pode assumir os valores enumerados `FetchType.FETCH` (padrão, carrega o objeto

correspondente ao relacionamento por completo) ou `FetchType.LAZY` (carrega apenas um proxy para o objeto, que será carregado posteriormente sob demanda).

- cascade (opcional) – determina a forma com que as inclusões, alterações e exclusões se propagam a cada um dos objetos aninhados. Pode assumir os valores enumerados `CascadeType.NONE` (padrão, não propaga nenhuma alteração do objeto continente para o objeto contido), `CascadeType.SAVE` (propaga apenas inserções e atualizações do objeto continente para o objeto contido), `CascadeType.CREATE` (propaga apenas eventos de criação do objeto continente para o objeto contido), `CascadeType.UPDATE` (propaga apenas alterações do objeto continente para o objeto contido), `CascadeType.DELETE` (propaga apenas deleções do objeto continente para o objeto contido) e `CascadeType.ALL` (propaga todas os eventos de persistência do objeto continente para o objeto contido).

Exemplo:

```php
<?php
 /**
  * @Entity
  * @Table(name="cliente")
  */
 class Cliente {

       /**
        * @var Telefone
        */
       private $novoTelefone;


       /**
        * @Column(name="id_telefone")
        * @OneToOne(fetch=FetchType.FETCH,cascade=CascadeType.ALL)
        * @return Telefone
        */
       public function getNovoTelefone() {
             return $this->novoTelefone;
       }
       ...
 }
 ?>
```

De acordo com o exemplo acima, a classe `Cliente` é persistente, como declarado em `@Entity`, e é persistida na tabela `cliente`, como declarado em `@Table(name="cliente")`. Esta classe possui um campo `$novoTelefone`, que guarda um objeto do tipo `Telefone`, como declarado em `@var Telefone`. O método *getter* deste campo é, segundo sua regra de nomenclatura *obrigatória*, `getNovoTelefone()`. Este método busca o objeto do tipo `Telefone`, segundo declarado em `@var Telefone`, que possua o *id* igual ao valor da coluna `id_telefone`, como declarado em `@Column(name="id_telefone")`, armazena esse valor no campo `$novoTelefone` e retorna-o ao solicitante.

O objeto Telefone `$novoTelefone` é buscado via `getNovoTelefone()` e armazenado em `$novoTelefone` no momento de criação da classe `Cliente`, como declarado em `fetch=FetchType.FETCH`, e todas operações de persistência em `Cliente` afetam o objeto aninhado `$novoTelefone` (quando um novo `Cliente` é criado um `Telefone` é criado, um `Telefone` é atualizado quando um `Cliente` é atualizado, um `Telefone` é salvo quando um `Cliente` é salvo, um `Telefone` é apagado quando o `Cliente` é apagado), como declarado em `cascade=CascadeType.ALL`.

Caso o campo do relacionamento `id_telefone` não estivesse na tabela `cliente` (por sua vez vinculado ao campo `$novoTelefone` da classe `Cliente`), teríamos outro caso. Estando este campo na tabela `telefone`, por exemplo o campo `id_cliente`, os parâmetros `mappedBy` deve ser definido na anotação `@OneToOne` de `Cliente`, como mostrado abaixo[6].

```
<?php
 /**
  * @Entity
  * @Table(name="cliente")
  */
 class Cliente {

       /**
        * @var Telefone
        */
       private $novoTelefone;


       /**
        * @OneToOne(fetch=FetchType.FETCH,
        *           cascade=CascadeType.ALL,
        *           mappedBy="cliente")
        * @return Telefone
        */
       public function getNovoTelefone() {
             return $this->novoTelefone;
       }
       ...
 }
 ?>
```

Neste caso, assumimos que o relacionamento entre `Cliente` e `Telefone` está mapeado na tabela `telefone`, vinculada à classe `Telefone`. Por isto não faz sentido usar a anotação `@Column` em `getNovoTelefone()`, já que a tabela `cliente` não possui o campo do relacionamento com a tabela `telefone`. Esta anotação deverá estar setada, no entanto, no método *getter* do campo da classe `Telefone` que representa o relacionamento com um `Cliente` (que podemos chamar, por exemplo, como `cliente`).

[6] Vale lembrar que o parâmetro `mappedBy` deve ser setado com o nome da propriedade da classe associada que representa o relacionamento, e não o nome do campo da tabela correspondente.

@OneToMany - define um relacionamento 1xN (um para muitos) entre duas classes.

Possui os parâmetros:

- mappedBy (obrigatório) - uma classe anotada com `@OneToMany` não possui o campo que representa o relacionamento (que sempre fica do lado N). Então, o parâmetro `mappedBy` deve ser setado com o nome do campo da outra classe do relacionamento que o representa.

- fetch (opcional) - determina a forma com que os dados trazidos do banco serão carregados no objeto. Pode assumir os valores enumerados `FetchType.FETCH` (padrão, carrega o objeto correspondente ao relacionamento por completo) ou `FetchType.LAZY` (carrega apenas um proxy para o objeto, que será carregado somente sob demanda).

- cascade (opcional) - determina a forma com que as inclusões, alterações e exclusões se propagam a cada um dos objetos aninhados. Pode assumir os valores enumerados `CascadeType.NONE` (padrão, não propaga nenhuma alteração do objeto continente para o objeto aninhado), `CascadeType.SAVE` (propaga apenas inserções e atualizações do objeto continente para o objeto contido), `CascadeType.CREATE` (propaga apenas criações do objeto continente para o objeto contido), `CascadeType.UPDATE` (propaga apenas alterações do objeto continente para o objeto contido), `CascadeType.DELETE` (propaga apenas deleções do objeto continente para o objeto contido) e `CascadeType.ALL` (propaga todas as ações do objeto continente para o objeto contido).

- targetEntity (obrigatório) - nome da classe que representa o lado N do relacionamento e possui a propriedade mapeada por `mappedBy`, que é a propriedade que representa o relacionamento.

Exemplo:

```php
<?php
 /**
  * @Entity
  * @Table(name="cliente")
  */
 class Cliente {

       /**
        * @var Collection
        */
       private $telefones;


       /**
        * @OneToMany(fetch=FetchType.FETCH,
        *           cascade=CascadeType.ALL,
        *           mappedBy="cliente",
        *           targetEntity="Telefone")
        * @return Collection
        */
```

```
        public function getTelefones() {
              return $this->telefones;
        }
        ...
}
?>
```

Neste exemplo, a classe `Cliente` possui uma coleção de telefones (necessariamente do tipo `Collection`). O relacionamento entre as classes `Cliente` e `Telefone` é um para muitos (um cliente possui vários telefones, mas um telefone pertence a apenas um cliente).

A anotação `@OneToMany` informa que:

- objetos instância da classe `Telefone` irão compor a coleção que o método `getTelefones()` recuperará, de acordo com `targetEntity="Telefone"`;

- a propriedade que mapeia o relacionamento (propriedade vinculada ao campo do relacionamento na tabela correspondente) é o campo de nome `cliente` da classe `Telefone`, de acordo com `mappedBy="cliente"` e `targetEntity="Telefone"`;

- a coleção de telefones será recuperada no momento de criação do objeto `Cliente`, de acordo com `fetch=FetchType.FETCH`;

- todas as operações sobre um objeto `Cliente` serão propagadas (cascateadas) à coleção de objetos `Telefone`, de acordo com `cascade=CascadeType.ALL`.

@ManyToOne - define um relacionamento Nx1 (muitos para um) entre duas classes. Descreve o inverso da anotação `@OneToMany`. Como é a classe do lado N do relacionamento que possui o campo que descreve o relacionamento, nunca precisaremos setar os parâmetros `mappedBy` ou `targetEntity` para uma anotação `@OneToMany`.

Possui os parâmetros:

- fetch (opcional) - determina a forma com que os dados trazidos do banco serão carregados no objeto. Pode assumir os valores enumerados `FetchType.FETCH` (padrão, carrega o objeto correspondente ao relacionamento por completo) ou `FetchType.LAZY` (carrega apenas um proxy para o objeto, que será carregado somente sob demanda).

- cascade (opcional) - determina a forma com que as inclusões, alterações e exclusões se propagam a cada um dos objetos aninhados. Pode assumir os valores enumerados `CascadeType.NONE` (padrão, não propaga nenhuma alteração do objeto continente para o objeto aninhado), `CascadeType.SAVE` (propaga apenas inserções e atualizações do objeto continente

para o objeto contido), `CascadeType.CREATE` (propaga apenas criações do objeto continente para o objeto contido), `CascadeType.UPDATE` (propaga apenas alterações do objeto continente para o objeto contido), `CascadeType.DELETE` (propaga apenas deleções do objeto continente para o objeto contido) e `CascadeType.ALL` (propaga todas as ações do objeto continente para o objeto contido).

Exemplo:

```php
<?php
/**
 * @Entity
 * @Table(name="telefone")
 */
class Telefone {

	/**
	 * @var Pessoa
	 */
	private $proprietario;

	/**
	 * @Column(name="id_pessoa")
	 * @ManyToOne(fetch=FetchType.LAZY, cascade=CascadeType.SAVE)
	 * @return Pessoa
	 */
	public function getProprietario() {
		return $this->proprietario;
	}
	...
}
?>
```

Neste exemplo, um objeto `Telefone` possui um `$proprietario`, instância da classe `Pessoa`.

O campo `$proprietario` da entidade está vinculado ao campo `id_pessoa` da tabela `telefone`, como atestado por `@Table(name="telefone")` e `@Column(name="id_pessoa")`. Lembrando que a anotação `@Column` pode estar presente tanto no comentário do campo da classe quanto no comentário do método *getter* deste mesmo campo. O relacionamento entre as classes `Telefone` e `Pessoa` é muitos para um (um telefone possui um proprietário, e um proprietário possui vários telefones).

No momento da criação de um telefone, seu proprietário não será recuperado, mas um proxy para este proprietário será armazenado na variável `$proprietario`. Somente no momento da primeira chamada ao método `getProprietario()` a busca será efetivamente realizada, de acordo com o `fetch=FetchType.LAZY`.

Ao criar um objeto telefone um proprietário é criado, e ao atualizar um telefone seu proprietário também é atualizado, de acordo com `cascade=CascadeType.SAVE`. Lembrando que

ao apagar um telefone *seu proprietário não é apagado*, por conta das configurações do cascateamento.

@ManyToMany - define um relacionamento NxM (muitos para muitos) entre duas classes. Um relacionamento NxM é mapeado, no modelo relacional, com uma tabela intermediária entre a tabela N e a M que guarda as chaves primárias de ambas como chaves estrangeiras e efetivamente constitui o relacionamento entre as duas. Para isto, **toda anotação `@ManyToMany` deve ser acompanhada de uma anotação `@JoinTable`**, que é descrita a seguir.

Possui os parâmetros:

- mappedBy (especial) - uma das classes anotadas com `@ManyToMany` deve possuir o parâmetro `mappedBy` setado com o nome do campo da outra classe que participa do relacionamento. Deve-se lembrar que **somente uma delas** deve possuir o parâmetro `mappedBy`. A classe que deve possuí-lo é a mais fraca no relacionamento mas, na prática, o lado do qual o `mappedBy` é setado não influencia no funcionamento da persistência.

- targetEntity (obrigatório) - nome da outra classe constituinte do relacionamento (*case sensitive*).

- fetch (opcional) - determina a forma com que os dados trazidos do banco serão carregados no objeto. Pode assumir os valores enumerados `FetchType.FETCH` (padrão, carrega o objeto correspondente ao relacionamento por completo) ou `FetchType.LAZY` (carrega apenas um proxy para o objeto, que será carregado somente sob demanda).

- cascade (opcional) - determina a forma com que as inclusões, alterações e exclusões se propagam a cada um dos objetos aninhados. Pode assumir os valores enumerados `CascadeType.NONE` (padrão, não propaga nenhuma alteração do objeto continente para o objeto aninhado), `CascadeType.SAVE` (propaga apenas inserções e atualizações do objeto continente para o objeto contido), `CascadeType.CREATE` (propaga apenas criações do objeto continente para o objeto contido), `CascadeType.UPDATE` (propaga apenas alterações do objeto continente para o objeto contido), `CascadeType.DELETE` (propaga apenas deleções do objeto continente para o objeto contido) e `CascadeType.ALL` (propaga todas as ações do objeto continente para o objeto contido).

@JoinTable - anotação que define em qual tabela o relacionamento NxM descrito em `@ManyToMany` deve ser persistido. Além da tabela, devem ser informadas as chaves primárias da tabela N e da tabela M (chaves estrangeiras na tabela de relacionamento).

Possui os seguintes parâmetros:

- name (obrigatório) - nome da tabela que representa o relacionamento NxM.

- joinColumns (obrigatório) – nome da chave estrangeira que mapeia a tabela N.

- inverseJoinColumns (obrigatório) – nome da chave estrangeira que mapeia a tabela M.

Exemplo:

```php
<?php
 /**
  * @Entity
  * @Table(name="pessoa")
  */
 class Pessoa {

       /**
        * @var Collection
        */
       private $cursos;

       /**
        * @ManyToMany(fetch=FetchType.LAZY,
        *           cascade=CascadeType.SAVE,
        *           mappedBy="alunos",
        *           targetEntity="Curso")
        * @JoinTable(name="pessoa_has_curso",
        *           joinColumns="id_pessoa",
        *           inverseJoinColumns="id_curso")
        * @return Collection
        */
       public function getCursos() {
             return $this->cursos;
       }

 }
 ?>
```

No exemplo acima, temos um relacionamento NxM (muitos para muitos) entre as classes `Pessoa` e `Curso` (uma pessoa participa de vários cursos e um curso possui vários alunos).

A anotação `@ManyToMany` nos informa várias coisas, tais como: o método `getCursos()` retorna uma coleção (`Collection`) de objetos do tipo `Curso`, de acordo com `targetEntity="Curso"`. O relacionamento é mapeado pela propriedade `alunos` da classe `Curso` (na outra classe do relacionamento), o que indica que a classe `Pessoa` é a classe mais fraca deste relacionamento, de acordo com `mappedBy="alunos"`. A coleção de cursos deve ser recuperada no momento da criação de um objeto `Pessoa`, de acordo com `fetch=FetchType.FETCH`. Todas as operações executadas em uma `Pessoa` devem ser

cascateadas aos cursos freqüentados por ela, de acordo com `cascade=CascadeType.ALL`.

A anotação `@JoinTable` informa que: o nome da tabela do relacionamento NxM é `pessoa_has_curso`, de acordo com `name="pessoa_has_curso"`. A coluna que representa a chave estrangeira para a tabela `pessoa` é a coluna de nome `id_pessoa`, de acordo com `joinColumns="id_pessoa"`. A coluna que representa a chave estrangeira para a tabela `curso` é a coluna de nome `id_curso`, como representado por `inverseJoinColumns="id_curso"`[7].

@Inheritance (especial) - define relações de especialização e generalização (herança) entre classes.

Possui um parâmetro:

- type (obrigatório) – especifica o tipo de persistência das entidades participantes da herança. Pode assumir os valores enumerados:

`InheritanceType.TABLE_PER_CLASS`

Cada uma das classes na hierarquia da herança está associada a uma tabela da base de dados, até mesmo as entidades abstratas, Por padrão o framework assumirá que as tabelas das classes filhas possuam um chave estrangeira para a tabela mãe do nível mais alto com o mesmo nome. Esse comportamento pode ser alterado usando a anotação `@JoinColumn(name="<nome_da_coluna>")` no bloco de comentários de classe nas classes filhas.

`InheritanceType.TABLE_PER_SUBCLASS`

Apenas as entidades concretas estão associadas a tabelas na base de dados, o que significa que os dados das entidades abstratas deve ser salvo repetidas vezes para cada entidade filha

`InheritanceType.SINGLE_TABLE`

Apenas a entidade mãe possui vinculo com uma tabela na base de dados e é responsável por persistir as informações de todas as entidades filhas.

Observações importantes: a anotação `@Inheritance` sempre deve ser colocada na classe-mãe (ou na classe mais geral) em uma herança. Caso o tipo de herança seja o `InheritanceType.TABLE_PER_CLASS`, todas as classes da herança devem possuir a anotação `@Table`. Caso o tipo seja `InheritanceType.TABLE_PER_SUBCLASS`, apenas as classes concretas da herança devem possuir a anotação `@Table`. Caso seja `InheritanceType.SINGLE_TABLE`, somente a classe mais geral (aquela que não possui ancestrais) deve ser anotada com `@Table`, já que ela é responsável pela persistência de todas as suas filhas.

[7] Se o mapeamento tiver que ser feito na entidade Curso, o valor do parâmetro joinColumns deverá ser "id_curso" e o do parâmetro inverseJoinColumns deverá ser "id_pessoa".

```php
/**
 * @Entity
 * @Inheritance(type=InheritanceType.TABLE_PER_CLASS)
 * @Table(name="pessoa")
 */
class Pessoa {
    /**
     * @var string
     */
    private $nome;

    /**
     * Declaração do id, outras propriedades e seus métodos acessores
     * ...
     */
}

/**
 * @Entity
 * @Table(name="pessoa_fisica")
 */
class PessoaFisica extends Pessoa {
    /**
     * @var string
     */
    private $cpf;

    /**
     * Declaração do id, outras propriedades e seus métodos acessores
     * ...
     */
}
```

No exemplo acima, temos duas classes, `Pessoa` e `PessoaFisica`, sendo que a última extende a primeira. A classe `Pessoa`, por ser a mais geral na herança, é anotada com `@Inheritance`, tipo `InheritanceType.TABLE_PER_CLASS`. Isto significa que toda classe participante da herança (neste caso somente `PessoaFisica`) possui uma tabela própria correspondente para persistência.

4. Utilizando o EntityFilter para realizar buscas simples

Em diversos momentos no seu código será necessário recuperar, na base de dados, entidades que atendam a um conjunto de critérios. Para que isso seja possível o Engine.DB fornece uma API de criação de critérios, tornando possível a filtragem dos resultados.

Inicialmente é necessário obter uma instância do `EntityFilter` para a entidade.

```
$filter = DAOFactory::getDAO()->getFilterFor('<nome_da_entidade>');
```

`<nome_da_entidade>` - Nome da classe sobre a qual serão aplicados os critérios de filtragem.

À partir do momento em obtivemos o filtro, existem várias operações que podemos realizar sobre ele.

Para setar um limite máximo de registros a recuperar, devemos utilizar o método `$filter->limit($number)`, onde `$number` é a quantidade máxima de registros que se deseja.

Para setar um offset (quantidade de registros a ignorar na busca antes de iniciar a obtenção do primeiro registro), devemos utilizar o método `$filter->offset($number)`, onde `$number` é a quantidade de registros a ignorar.

Caso queiramos utilizar um mecanismo de fetching diferente do padrão (`FetchType.LAZY`, mas atenção, o fetching padrão temporariamente será o `FetchType.FETCH`, enquanto o `FetchType.LAZY` não for implementado), devemos utilizar o método `$filter->setFetchMode($alias, $type)`, onde `$alias` deve ser uma string contendo o caminho para a entidade referenciada e `$type` o valor enumerado contendo o tipo de fetching desejado para a entidade, `FetchType.LAZY` ou `FetchType.FETCH`.

Para obtermos a query SQL para o filtro configurado, devemos chamar `$filter->getSqlString()`. O valor retornado é uma string contendo o código SQL necessário para a busca dos valores desejados. Este método é utilizado internamente quando chamamos `getUnique()` e `getList()`, que serão explicados mais adiante.

Segue adiante a descrição de outros métodos contidos na classe `EntityFilter`.

`$filter->getClass()` – retorna a classe do filtro sendo utilizado.

`$filter->getAlias()` e `$filter->setAlias()` – utilizados para recuperar/setar o apelido (alias) para a entidade para a qual se solicitou o filtro.

`$filter->getEntityData()` e `$filter->setEntityData()` – utilizados para recuperar/setar o objeto `EntityData` (objeto de metadados de entidade) referente à entidade para a qual solicitou-se o filtro.

Em seguida, para utilizarmos efetivamente do filtro, devemos adicionar os critérios obtidos através da utilização do método `$filter->add(<parâmetro>)`, onde o parâmetro deverá ser obtido com a utilização das classes `FilterParameter`, `FilterCondition`, `FilterGroupBy`, `FilterOrderBy`.

- **`FilterParameter::eq`** – adiciona uma constraint "igual a" à propriedade fornecida.

Ex: `$filter->add(FilterParameter::eq("codigo",123456))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` igual a 123456.

- **`FilterParameter::ne`** – adiciona uma constraint "diferente de" à propriedade dada.

Ex: `$filter->add(FilterParameter::ne("codigo",123456))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` diferente de 123456.

- **`FilterParameter::lt`** - adiciona uma constraint de "menor que" à propriedade dada.

Ex: `$filter->add(FilterParameter::lt("codigo",123456))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` menor que 123456.

- **`FilterParameter::gt`** - adiciona uma constraint de "maior que" à propriedade dada.

Ex: `$filter->add(FilterParameter::gt("codigo",123456))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` maior que 123456.

- **`FilterParameter::le`** - adiciona uma constraint de "menor ou igual a" à propriedade dada.

Ex: `$filter->add(FilterParameter::le("codigo",123456))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` menor ou igual a 123456.

- **`FilterParameter::ge`** - adiciona uma constraint de "maior ou igual a" à propriedade dada.

Ex: `$filter->add(FilterParameter::ge("codigo",123456))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` maior ou igual a 123456.

- **`FilterParameter::like`** - adiciona uma constraint de "semelhante a" à propriedade fornecida. Pode-se utilizar o caractere coringa '%' para determinar um trecho do dado pesquisado, ou ainda seu início ou término.

Ex: `$filter->add(FilterParameter::like("nome",”Seelaz%”))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `nome` iniciando por Seelaz.

- **`FilterParameter::isNull`** - adiciona uma constraint de “é nulo” à propriedade fornecida.

Ex: `$filter->add(FilterParameter::isNull("codigo"))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` nulo.

- **`FilterParameter::isNotNull`** - adiciona uma constraint de “não é nulo” à propriedade fornecida.

Ex: `$filter->add(FilterParameter::isNotNull("codigo"))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `codigo` não-nulo.

- **`FilterParameter::eqProperty`** - adiciona uma constraint de "igual a" entre duas propriedades.

Ex**:** `$filter->add(FilterParameter::eqProperty("dataInicio",”dataFim”))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `dataInicio` igual ao campo `dataFim`.

- **`FilterParameter::neProperty`** - adiciona uma constraint de "diferente de" entre duas propriedades.

Ex: `$filter->add(FilterParameter::neProperty("dataInicio",”dataFim”))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `dataInicio` diferente do campo `dataFim`.

- FilterParameter::ltProperty - adiciona uma constraint de "menor que" entre duas propriedades.

Ex: `$filter->add(FilterParameter::ltProperty("dataInicio",”dataFim”))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `dataInicio` menor que o campo `dataFim`.

- FilterParameter::gtProperty - adiciona uma constraint de "maior que" entre duas propriedades.

Ex: `$filter->add(FilterParameter::gtProperty("dataInicio",”dataFim”))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `dataInicio` maior que o campo `dataFim`.

- FilterParameter::leProperty - adiciona uma constraint de "menor ou igual a" entre duas propriedades.

Ex: `$filter->add(FilterParameter::leProperty("dataInicio",”dataFim”))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `dataInicio` menor ou igual ao campo `dataFim`.

- FilterParameter::geProperty - adiciona uma constraint de "maior ou igual a" entre duas propriedades.

Ex: `$filter->add(FilterParameter::geProperty("dataInicio",”dataFim”))`

Trás todas as entidades `<nome_da_entidade>` que possuírem o campo `dataInicio` maior ou igual ao campo `dataFim`.

Para critérios mais complexos:

- FilterParameter::andConditions - cria uma operação ‘AND’ entre duas condições.

Ex:

`$filter->add(FilterParameter::andConditions(<condição1>,<condição2>))`

- `<condição_1>` e `<condição_2>` aceitam parâmetros simples utilizando `FilterParameter`. Para agrupar mais condições é necessário o uso do

`FilterCondition::add(<condição>).`

- `FilterParameter::orConditions` - cria uma operacao 'OR' entre duas condições

Ex:

```
$filter->add(FilterParameter::orConditions (<condição1>,<condição2>))
```

- `<condição_1>` e `<condição_2>` aceitam parâmetros simples utilizando `FilterParameter`. Para agrupar mais condições é necessário o uso do `FilterCondition::add(<condição>).`

`FilterGroupBy`, `FilterOrderBy` são utilizados nas chamadas dos métodos:

```
$filter->addOrderBy(FilterOrderBy::asc("<nome_da_propriedade>"));
$filter->addOrderBy(FilterOrderBy::desc("<nome_da_propriedade>"));
$filter->addGroupBy(new FilterGroupBy("<nome_da_propriedade>")).
```

Para determinar-se critérios para entidades compostas é necessário criar *aliases* (apelidos) para que seja possível referenciar os atributos de cada uma dessas entidades.

```
$filter->createAlias("<propriedade>","<apelido>",<join_type>)
```

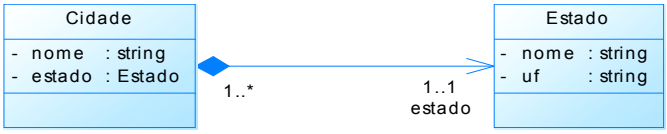
Onde:

`<propriedade>` – nome da propriedade na classe.

`<apelido>` – palavra que representará a entidade composta.

`<join_type>` – Utiliza-se para modificar a estratégia de junção na busca pela entidade. Aceita como parâmetro os valores enumerados `JoinType.INNER` (padrão, executa um `INNER JOIN` contra o banco) e `JoinType.LEFT` (executa um `LEFT JOIN` contra o banco).

Vejamos por exemplo esse caso:



Para conseguirmos a listagem de todas as cidades do estado de Goiás poderíamos fazer assim:

```
$filter = DAOFactory::getDAO()->getFilterFor('Cidade');
$filter->createAlias("estado","e");
$filter->add(FilterParameter::eq("e.uf","GO"));
```

Uma vez definidos os critérios, é possível determinar se o resultado deve ser único ou uma lista através das chamadas a um dos dois métodos: `getUnique()`, se o resultado esperado for único; ou `getList()`, que obtém um *array* com as entidades que atendem aos critérios fornecidos.

Segue abaixo um exemplo de como utilizar os dois métodos:

```
$obj = $filter->getUnique(); //gera uma exception caso resultado não seja único.
$list = $filter->getList();
```

5. Executando SQL Nativo

Em algum momento você pode vir a precisar de executar algumas queries no banco. O engine permite que sejam executadas queries diretamente do driver configurado, oferecendo ainda métodos para controle de transações e recuperação de resultados.

Para se obter o driver basta somente utilizar o seguinte código:

```
$driver = DAOFactory::getDAO()->getDriver();

//A partir da obtenção do driver voce poderá utilizar os seguintes métodos:

//Inicia a conexão com o bannco de dados;
$driver->connect();
//Termina a conexão com o bannco de dados;
$driver->disconnect();

//Inicia uma transação;
$driver->begin();
//Comita uma transação;
$driver->commit();
//Desfaz as alterações da ultima transação;
$driver->rollback();

//Executa uma query e retorna o Resource resultante;
$driver->run($sql);
//Executa uma query e retorna um array com os resultados indexados pelos nomes das colunas ou aliases utilizados;
$driver->fetchAssoc($sql);
```

Esses comandos devem permitir que a grande maioria das operações desejadas sejam possiveis de se realizar. A sintaxe deve seguir os padrões do SGBD utlizado pois o engine não oferece um dialeto “Genérico” (Ainda).

6. Depurando Queries

Uma outra situação que com certeza surgirá é a necessidade de se saber o que o engine está enviando ao banco. É possível definir um arquivo de log para visualização posterior. Para isso utilize o seguinte código:

```
import('engine.mvc.Logger');
define('ENGINE_DEBUG_VERBOSE',20);
define('ENGINE_DEBUG_LOG',true);
$logger = new Logger("app.log");
```

Com isso além de ter as Query discriminadas nas mensagens de erro, um log será criado no diretório do arquivo que inclui o engine com algumas informações adicionais. **Mas atenção**, remova ou comente essas linhas ao colocar a aplicação em produção ou então qualquer um poderá visualizar o SQL gerado nas mensagens de erro.