

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Attila Saint Ouen |  | PAQUET Judicaël |  | Judicael.paquet@gmail.com  https://github.com/las93 |

Attila PHP

[Le nouvel ORM PHP]

Table des matières

1. [Présentation 2](#_Toc409550386)

[Installation](#_Toc409550387)

[Concept du routeur](#_Toc409550388)

1. [Présentation 2](#_Toc409550389)

[Installation](#_Toc409550390)

# Présentation

Attila est une nouvelle alternative d’ORM de qualité de développement open-source dans les environnements LAMP.

## Installation

L’installation de l’ORM est relativement simple. Vous pouvez télécharger la version .zip sur le Github <https://github.com/las93> ou ajouter las93/attila dans votre composer.json.

## Concept du routeur

Xxx

# Utilisation basique

Pour bien utiliser Attila, voici les premières étapes à respecter afin de proposer le meilleur des ORM à vos différents projets.

## XXX

XXX l’ORM

# Utilisation avancée

Voici la liste des utilisations avancées d’Attila qui pourront beaucoup vous aider dans vos différents projets.

## Les transactions

**Disponibilité : Attila 1.0.0-beta2**

Pour faire une transaction sous Attila, il faut utiliser les fonctions suivantes de l’ORM :

|  |
| --- |
| **// Commencer une transaction $oOrm->begin();  // Valider une transaction qui s’appliquera intégralement en base $oOrm->commit() ;   // Invalider une transaction qui ne s’appliquera pas en base. // Toutes les modifications seront annulées $oOrm->rollback() ;** |

## 4 triggers d’initialisation sur les entités

**Disponibilité : Attila 1.0.0-beta2**

Les entités ont deux triggers qui permettent d’être initialisé :

1/ La fonction initialize permet d’initialiser l’entité la première fois qu’elle est appelée et uniquement cette fois-là pendant tout le déroulement du script (également disponible sur le modèle) :

|  |
| --- |
| **public function** initialize() { ; } |

2/ La fonction onConstruct permet d’initialiser l’entité à chaque fois qu’elle est instanciée (également disponible sur le modèle)  :

|  |
| --- |
| **public function** onConstruct() { ; } |

3/ La fonction beforeSave permet d’initialiser l’entité avant chaque save effectué :

|  |
| --- |
| **public function** beforeSave() { ; } |

4/ La fonction afterFetch permet d’initialiser l’entité dès qu’elle a été remplit (par load) :

|  |
| --- |
| **public function** afterFetch() { ; } |

## Des filtres sur les résultats de l’ORM

**Disponibilité : Attila 1.0.0-beta2**

Attila vous propose de créer des filtres qui s’appliquent avant le retour des résultats au contrôleur.

|  |
| --- |
| **$cFilter = function($oArticle) {**  **$oArticle->title = ‘Monsite.com : ’.$oArticle->title;**  **}**  **$aResults = $oArticleModel->filter($cFilter)**  **->find();** |
|  |

## Automatisation des jointures many to many

**Disponibilité : Attila 1.0.0-beta2**

Attila vous propose de créer vos tables et vos entités de type many to many automatiquement. Pour cela il suffit de rajouter le champ « many\_to\_many » sur la clé primaire de la table en indiquant la table sur laquelle vous voulez faire ce type de jointure. Voici un exemple ci-dessous :

|  |
| --- |
| "tables": {  "attribute": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true,  **"many\_to\_many": "category"**  },  "name": {  "type": "varchar",  "value": 100,  "null": false  },  "type": {  "type": "enum",  "values": [ "predefined", "free" ],  "null": false  }  }  },  "category": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "name": {  "type": "varchar",  "null": false,  "value": 100  }  }  }  } |
|  |

Il va créer automatiquement la table (et l’entité correspondante ci-dessous sans que vous ayez besoin de la définir dans votre fichier de configuration :

|  |
| --- |
| "tables": {  "attribute\_category": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true  },  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true  }  }  }  } |
|  |

## Fonctions pour créer vos jointures dans les entités

**Disponibilité : Attila 1.0.0-beta2**

Si vous écrivez vous-même vos entités, il est également possible de créer vous-même vos jointures dans la fonction trigger initialize() comme ci-dessous des jointures « one to many » ou « one to one » :

|  |
| --- |
| **public function** initialize()  {  $this->hasMany('id', 'product', 'id\_main\_category', '\Venus\src\Helium\Entity');  }  **public function** initialize()  {  $this->hasOne('id', 'product', 'id\_main\_category', '\Venus\src\Helium\Entity');  } |

Dans ces fonctions, le premier paramètre correspond au champ de l’entité sur lequel on fera la jointure, le deuxième paramètre est le nom de l’entité destinataire sur laquelle on veut fait la jointure, le troisième paramètre est le nom du champ de l’entité destinataire sur lequel on fait la jointure et le quatrième paramètre est le namespace de l’entité destinataire.

Quand vous avez déclaré vos relations, vous pouvez appeler votre relation en ajoutant get\_ au nom de l’entité destinataire sur laquelle on fait la jointure :

|  |
| --- |
| **$oMyCategory**->get\_product(); |

Nous pouvons également définit une fonction qui prendra cette fonction grâce à la méthode getRelated() :

|  |
| --- |
| **public function** GetMaRelationAProduct($mParameters)  {  return $this->getRelated('product', $mParameters);  } |

## Fonctions pour créer vos jointures dans les modèles

**Disponibilité : Attila 1.0.0-beta2**

Attila permet de créer aussi des jointures « many to one » avec belongsTo() ou « many to many » avec hasManyToMany() à partir des modèles qui gèrent les listes de résultats. Voici les deux fonctions à ajouter pour offrir ces jointures à vos modèles :

|  |
| --- |
| **public function** initialize()  {  $this->belongsTo('id', 'product', 'id\_main\_category', '\Venus\src\Helium\Entity');  }  **public function** initialize()  {  $this->hasManyToMany('id', 'product', 'id\_main\_category', '\Venus\src\Helium\Entity', 'id\_ category', 'category');  } |

## Les clés étrangères virtuelles

**Disponibilité : Attila 1.0.0-RC1**

Attila permet de créer des clés étrangères virtuelles (à partir des entités). Même si vos bases ne contiennent pas de clés étrangères, Attila fera les tests nécessaires pour valider les insertions ou les suppressions. Vous pourrez déclarer ces clés étrangères des deux côtés :

|  |
| --- |
| **public function** initialize()  {  $this->hasMany('id', 'product', 'id\_main\_category', '\Venus\src\Helium\Entity', array (  ‘foreignKey’ => array(  ‘message’ => ‘ca ne marche pas’  )  ));  }  **public function** initialize()  {  $this->hasOne('id', 'product', 'id\_main\_category', '\Venus\src\Helium\Entity', array (  ‘foreignKey’ => array(  ‘action => self::CASCADE  )  ));  } |
|  |

## Le modèle propose la fonction count

**Disponibilité : Attila 1.0.0-RC1**

Attila permet compter le nombre de résultats d’une table grâce à la fonction count :

|  |
| --- |
| $oCategory = **new** Category;  var\_dump($oCategory->count()); |
|  |

Attila permet compter le nombre de résultats d’une table en faisant un distinct précis sur un champ grâce à la fonction count :

|  |
| --- |
| $oCategory = **new** Category;  var\_dump($oCategory->count(‘distinct’ => ‘name’)); |
|  |

Attila permet compter le nombre de résultats d’une table en retournant le nombre de résultat pour chaque ligne grâce au paramètre group :

|  |
| --- |
| $oCategory = **new** Category;  var\_dump($oCategory->count(‘group’ => ‘name’)); |
|  |

## Des clés étrangères

**Disponibilité : Attila 1.0.0-RC1**

Venus 2 permet de définir des clés étrangères complètes dans les fichiers de configurations de bases de données. Voici un exemple complet :

|  |
| --- |
| "id\_role": {  "type": "int",  "value": "10",  "null": false,  "join": "role",  "join\_by\_field": "id",  "join\_delete": "cascade",  "join\_update": "cascade",  "constraint": "role\_fk"  }, |

## Définir le moteur, l’auto-increment et le charset de la table

**Disponibilité : Attila 1.0.0-RC1**

Venus 2 permet de définir des compléments d’information sur les tables que vous voulez créer comme le moteur utilisé, l’auto-increment de départ et le charset par défaut de la table. Voici un exemple de table avec ce type de définition :

|  |
| --- |
| "board": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "name": {  "type": "varchar",  "value": "50",  "null": false  },  "id\_role": {  "type": "int",  "value": "10",  "null": false,  "join": "role",  "join\_by\_field": "id",  "join\_delete": "cascade",  "join\_update": "cascade",  "constraint": "role\_fk"  },  "id\_user": {  "type": "int",  "value": "10",  "null": false,  "join": "user",  "join\_by\_field": "id",  "join\_delete": "cascade",  "join\_update": "cascade",  "constraint": "user\_fk"  }  },  "engine": "InnoDB",  "auto\_increment": 3,  "default\_charset": "utf8"  } |

## Définir des index dans la table

**Disponibilité : Attila 1.0.0-RC1**

Venus 2 permet de définir des index de deux façons différentes (soit groupé, soit directement sur les champs en question). On ne pourra créer des index multiple que dans la première façon de faire en revanche. Voici deux exemples de table avec ce type de définition :

|  |
| --- |
| "user": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "id\_team": {  "type": "int",  "null": false,  "unsigned": true  }  },  "index": [  "id\_team\_name\_index": ["id\_team"]  ],  "engine": "InnoDB",  "auto\_increment": 1,  "default\_charset": "utf8"  }, |

ou

|  |
| --- |
| "user": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "id\_team": {  "type": "int",  "null": false,  "unsigned": true,  "key": “index”  }  },  "engine": "InnoDB",  "auto\_increment": 1,  "default\_charset": "utf8"  }, |

## Définir des champs uniques dans la table

**Disponibilité : Attila 1.0.0-RC1**

Attila permet de définir des champs uniques de deux façons différentes (soit groupé, soit directement sur les champs en question). On ne pourra créer des champs unqiues multiples que dans la première façon de faire en revanche. Voici deux exemples de table avec ce type de définition :

|  |
| --- |
| "user": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "id\_team": {  "type": "int",  "null": false,  "unsigned": true  }  },  "index": [  "id\_team\_name\_index": ["id\_team"]  ],  "engine": "InnoDB",  "auto\_increment": 1,  "default\_charset": "utf8"  }, |

ou

|  |
| --- |
| "user": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "id\_team": {  "type": "int",  "null": false,  "unsigned": true,  "key": “index”  }  },  "engine": "InnoDB",  "auto\_increment": 1,  "default\_charset": "utf8"  }, |

\* faire belongsTo (n-1) et hasManyToMany (n-n) sur les modèles.