Odoo utilise plusieurs décorateurs dans son framework pour personnaliser le comportement des méthodes et des champs dans les modèles. Voici une explication des décorateurs les plus couramment utilisés dans Odoo, classés par catégories.

1. Décorateurs pour les Méthodes

- **@api.model** : Ce décorateur est utilisé pour marquer une méthode comme une méthode de modèle. Cela signifie que la méthode ne recevra que l'objet self comme paramètre, qui représente l'ensemble du modèle. Les méthodes décorées avec @api.model peuvent être appelées sur des objets singleton ou multi-enregistrements.

```
@api.model
def create(self, vals):
```

- **@api.multi** : Ce décorateur a été utilisé pour indiquer que la méthode peut traiter plusieurs enregistrements (une liste de self), mais à partir d'Odoo 12.0, ce décorateur est devenu obsolète. Il est maintenant recommandé d'utiliser uniquement @api.model ou @api.model_create_multi lorsque cela est nécessaire.
- **@api.one** : Ce décorateur a également été utilisé dans les versions précédentes pour traiter un seul enregistrement à la fois et retourner un seul résultat, mais il est désormais obsolète depuis Odoo 12.0.
- **@api.depends** : Ce décorateur est utilisé pour indiquer que le résultat d'une méthode dépend de la valeur d'autres champs. Odoo utilise cette information pour déterminer quand la méthode doit être recalculée.

```
@api.depends('field1', 'field2')
def _compute_total(self):
    for record in self:
        record.total = record.field1 + record.field2
```

- **@api.onchange** : Ce décorateur est utilisé pour déclencher une méthode lorsqu'un ou plusieurs champs spécifiés sont modifiés dans l'interface utilisateur. Il est souvent utilisé pour mettre à jour d'autres champs de manière dynamique en fonction des valeurs modifiées.

```
@api.onchange('field1')
def _onchange_field1(self):
    if self.field1 > 10:
        self.field2 = 20
```

- **@api.constrains** : Ce décorateur est utilisé pour définir une méthode qui vérifie des contraintes sur certains champs. Si la contrainte est violée, une exception est levée, ce qui empêche l'enregistrement d'être validé.

```
@api.constrains('field1', 'field2')
def _check_values(self):
    if self.field1 > self.field2:
        raise ValidationError("Field1 cannot be greater than Field2.")
```

- **@api.returns** : Ce décorateur est utilisé pour indiquer le type de valeur que la méthode retourne. Il est rarement utilisé dans les modules standard, mais peut être utile pour la documentation ou la gestion de certains cas spécifiques.

```
@api.returns('self', lambda value: value.id)
def some_method(self):
    return self
```

- **@api.model_create_multi** : Utilisé pour définir une méthode de création capable de traiter plusieurs enregistrements à la fois. Il est utilisé principalement dans les méthodes create pour accepter une liste de dictionnaires de valeurs.

```
@api.model_create_multi
def create(self, vals_list):
    return super(MyModel, self).create(vals_list)
```

- **@api.autovacuum** : Ce décorateur est utilisé pour définir une méthode qui doit être exécutée automatiquement par le processus de nettoyage automatique d'Odoo.

-

```
@api.autovacuum
def _cleanup_old_data(self):
    # Custom logic for cleaning up old data
```

- **@api.returns** : Il est utilisé pour annoter une méthode en spécifiant son type de retour. Cela peut être utile pour la documentation et la gestion des types, mais il n'est pas couramment utilisé.

2. Décorateurs pour les Champs

- **@api.depends** : Mentionné précédemment, ce décorateur est utilisé non seulement pour les méthodes mais aussi pour les champs calculés. Il indique à Odoo quels champs déclenchent le recalcul du champ calculé.

```
total = fields.Float(compute='_compute_total', store=True)
@api.depends('field1', 'field2')
def _compute_total(self):
    self.total = self.field1 + self.field2
```

- **@api.onchange** : Comme pour les méthodes, ce décorateur peut être utilisé pour déclencher un changement de valeur sur un champ en fonction de la modification d'un autre champ.

```
discount = fields.Float()
@api.onchange('price')
def _onchange_price(self):
    if self.price > 100:
        self.discount = 10
```

- **@api.constrains** : Utilisé pour vérifier des contraintes sur les champs.
- **@api.depends_context** : Ce décorateur est une extension du décorateur @api.depends qui permet de marquer les méthodes dépendant du contexte, en particulier des clés de contexte.

```
@api.depends_context('lang')
def _compute_name_in_lang(self):
```

```
    for record in self:
        record.name = self.env['res.lang'].name
```

3. Autres Décorateurs

- **@tools.ormcache** : Ce décorateur est utilisé pour mettre en cache les résultats des méthodes dans la mémoire du serveur Odoo. Il est utile pour des méthodes qui sont fréquemment appelées avec les mêmes paramètres et dont le résultat ne change pas fréquemment.

```
from odoo.tools import ormcache

@ormcache('self.env.uid', 'partner_id')
def get_partner_name(self, partner_id):
    return self.env['res.partner'].browse(partner_id).name
```

- **@job** : Utilisé pour marquer une méthode comme étant exécutable en arrière-plan (background job). Ce décorateur est utilisé dans les modules de gestion de files d'attente de tâches.

```
from odoo.addons.queue_job.job import job

@job
def my_long_running_job(self):
    # Long-running task
```

- **@api.v8 et @api.v7** : Ces décorateurs sont utilisés pour la compatibilité descendante avec les API des anciennes versions d'Odoo. Ils permettent de maintenir des méthodes compatibles avec les anciennes versions tout en introduisant des nouvelles.

Conclusion

Ces décorateurs sont essentiels pour contrôler et personnaliser le comportement des modèles, méthodes et champs dans Odoo. En comprenant leur utilisation, vous pouvez mieux structurer et optimiser votre code pour tirer le meilleur parti du framework Odoo.

chacun avec un exemple et explications

Voici une explication détaillée de chaque décorateur Odoo avec un exemple pour illustrer leur utilisation.

1. @api.model

Description:

Utilisé pour les méthodes qui ne dépendent pas de l'enregistrement spécifique sur lequel elles sont appelées. Ces méthodes reçoivent un self représentant le modèle, et non un enregistrement particulier.

Exemple:

```
from odoo import models, api

class MyModel(models.Model):
    _name = 'my.model'

    @api.model
    def create_default_record(self):
        # Crée un enregistrement avec des valeurs par défaut
        return self.create({'name': 'Default Name'})
```

Explication:

Ici, la méthode create_default_record est une méthode de modèle qui crée un nouvel enregistrement avec une valeur par défaut. Elle peut être appelée sans faire référence à un enregistrement spécifique.

---

2. @api.multi

Description:

Destiné à être utilisé sur des méthodes qui peuvent être appelées sur plusieurs enregistrements (self est une liste d'enregistrements). Ce décorateur est obsolète à partir d'Odoo 12.

Exemple:

```
from odoo import models, api

class MyModel(models.Model):
    _name = 'my.model'
    @api.multi
    def compute_total(self):
        for record in self:
            record.total = record.amount1 + record.amount2
```

Explication:

La méthode compute_total est destinée à être utilisée sur plusieurs enregistrements. Odoo gère l'itération sur chaque enregistrement dans self.

---

3. @api.depends

Description:

Utilisé pour spécifier que la méthode ou le champ dépend de la valeur d'autres champs. Cela aide Odoo à savoir quand recalculer ou réévaluer une méthode ou un champ.

Exemple:

```
from odoo import models, fields, api

class MyModel(models.Model):
    _name = 'my.model'
    amount1 = fields.Float()
    amount2 = fields.Float()
    total = fields.Float(compute='_compute_total')
    @api.depends('amount1', 'amount2')
    def _compute_total(self):
```

```
        for record in self:
            record.total = record.amount1 + record.amount2
```

Explication:

La méthode _compute_total est recalculée automatiquement chaque fois que amount1 ou amount2 change.

---

4. @api.onchange

Description:

Déclenche une méthode lorsqu'un champ spécifié est modifié dans l'interface utilisateur. Utilisé pour mettre à jour d'autres champs en fonction des valeurs modifiées.

Exemple:

```
from odoo import models, fields, api
class MyModel(models.Model):
    _name = 'my.model'
    price = fields.Float()
    discount = fields.Float()
    @api.onchange('price')
    def _onchange_price(self):
        if self.price > 100:
            self.discount = 10
```

Explication:

Si price dépasse 100, la méthode _onchange_price est appelée automatiquement, et discount est mis à jour à 10.

---

5. @api.constrains

Description:

Utilisé pour définir des contraintes sur des champs. Si une contrainte est violée, une exception est levée, empêchant l'enregistrement d'être validé.

Exemple:

```
from odoo import models, fields, api
from odoo.exceptions import ValidationError
class MyModel(models.Model):
    _name = 'my.model'
    amount1 = fields.Float()
    amount2 = fields.Float()
    @api.constrains('amount1', 'amount2')
    def _check_amounts(self):
        if self.amount1 > self.amount2:
            raise ValidationError("Amount1 cannot be greater than Amount2.")
```

Explication:

La méthode _check_amounts vérifie que amount1 n'est pas supérieur à amount2. Si cette condition est violée, une erreur de validation est levée.

---

6. @api.model_create_multi

Description:

Utilisé pour les méthodes de création qui peuvent traiter plusieurs enregistrements à la fois, particulièrement utile pour optimiser les opérations en masse.

Exemple:

```
from odoo import models, api

class MyModel(models.Model):
    _name = 'my.model'
    @api.model_create_multi
    def create(self, vals_list):
        return super(MyModel, self).create(vals_list)
```

Explication:

Cette méthode create peut gérer une liste de valeurs (vals_list) et créer plusieurs enregistrements en une seule opération.

---

7. @api.returns

Description:

Spécifie le type de retour d'une méthode pour aider à la documentation et à la gestion des types.

Exemple:

```
from odoo import models, api

class MyModel(models.Model):
    _name = 'my.model'
    @api.returns('self', lambda value: value.id)
    def some_method(self):
        return self
```

Explication:

Ce décorateur indique que la méthode some_method retourne un enregistrement (self), et le type de retour peut être géré en fonction de l'ID de l'enregistrement.

---

8. @api.autovacuum

Description:

Indique qu'une méthode doit être exécutée automatiquement par le processus de nettoyage automatique d'Odoo.

Exemple:

```
from odoo import models, api

class MyModel(models.Model):
    _name = 'my.model'

    @api.autovacuum
    def _cleanup_old_data(self):
        # Logic to clean up old data
        pass
```

Explication:

Cette méthode _cleanup_old_data est appelée automatiquement par le processus de nettoyage d'Odoo pour effectuer des tâches de maintenance.

---

9. @tools.ormcache

Description:

Utilisé pour mettre en cache les résultats des méthodes dans la mémoire du serveur, utile pour des méthodes fréquemment appelées avec les mêmes paramètres.

Exemple:

```
from odoo import models
from odoo.tools import ormcache

class MyModel(models.Model):
    _name = 'my.model'

    @ormcache('self.env.uid', 'partner_id')
```

```
    def get_partner_name(self, partner_id):
        return self.env['res.partner'].browse(partner_id).name
```

Explication:

Le résultat de la méthode get_partner_name est mis en cache en fonction de l'utilisateur (self.env.uid) et de l'ID du partenaire (partner_id), réduisant ainsi les appels répétitifs à la base de données.

---

10. @job

Description:

Utilisé pour marquer une méthode comme une tâche en arrière-plan (background job). Utilisé principalement dans les modules de gestion de files d'attente de tâches.

Exemple:

```
from odoo import models
from odoo.addons.queue_job.job import job
class MyModel(models.Model):
    _name = 'my.model'
    @job
    def my_long_running_job(self):
        # Long-running task logic
        pass
```

Explication:

La méthode my_long_running_job est enregistrée comme une tâche qui peut être exécutée en arrière-plan, ce qui est utile pour des processus longs ou intensifs.

---

11. @api.depends_context

Description:

Ce décorateur est utilisé pour marquer les méthodes ou champs qui dépendent du contexte, notamment des clés spécifiques dans le contexte.

Exemple:

```
from odoo import models, fields, api

class MyModel(models.Model):
    _name = 'my.model'
    name_in_lang = fields.Char(compute='_compute_name_in_lang')
    @api.depends_context('lang')
    def _compute_name_in_lang(self):
        for record in self:
            record.name_in_lang = self.env['res.lang'].name
```

Explication:

La méthode _compute_name_in_lang est recalculée en fonction du contexte de la langue (lang). Cela signifie que la valeur du champ name_in_lang changera selon la langue active dans le contexte utilisateur.

Ces décorateurs permettent de personnaliser et d'optimiser le comportement des modèles, méthodes, et champs dans Odoo, en offrant un cadre puissant pour le développement d'applications métiers.

Voici une liste des méthodes courantes disponibles pour les modèles Odoo, accompagnée d'explications et d'exemples pour chaque méthode.

1. **`create`**

Description:

Cette méthode est utilisée pour créer un nouvel enregistrement dans la base de données.

Exemple:

```
from odoo import models, fields

class MyModel(models.Model):
    _name = 'my.model'
    name = fields.Char()
    @api.model
    def create(self, vals):
        # Ajout d'une logique avant la création
        vals['name'] = vals.get('name', 'Default Name')
        return super(MyModel, self).create(vals)
```

Explication:

La méthode `create` reçoit un dictionnaire de valeurs (`vals`) pour l'enregistrement à créer. Vous pouvez personnaliser les valeurs avant de passer au traitement par Odoo.

---

2. **`write`**

Description:

Utilisée pour mettre à jour les enregistrements existants avec de nouvelles valeurs.

Exemple:

```
from odoo import models, fields
class MyModel(models.Model):
    _name = 'my.model'
    name = fields.Char()
    def write(self, vals):
        # Ajout d'une logique avant la mise à jour
        if 'name' in vals and not vals['name']:
            vals['name'] = 'Updated Name'
        return super(MyModel, self).write(vals)
```

Explication:

La méthode `write` reçoit un dictionnaire de valeurs à mettre à jour. Vous pouvez modifier ces valeurs avant de les enregistrer.

---

3. **`unlink`**

Description:

Cette méthode est utilisée pour supprimer des enregistrements.

Exemple:

```
from odoo import models, api
from odoo.exceptions import UserError
class MyModel(models.Model):
    _name = 'my.model'
    name = fields.Char()
    @api.multi
    def unlink(self):
        for record in self:
            if record.name == 'Protected':
                raise UserError("Cannot delete protected records.")
        return super(MyModel, self).unlink()
```

Explication:

La méthode `unlink` est appelée pour supprimer les enregistrements. Vous pouvez ajouter des vérifications avant de procéder à la suppression réelle.

---

4. **`search`**

Description:

Permet de rechercher des enregistrements dans la base de données en utilisant des critères de recherche.

Exemple:

```
from odoo import models

class MyModel(models.Model):
    _name = 'my.model'
    def find_records(self, name):
        return self.search([('name', '=', name)])
```

Explication:

La méthode `search` prend une liste de tuples définissant les critères de recherche et renvoie les enregistrements correspondants.

---

5. **`browse`**

Description:

Permet de récupérer des enregistrements à partir de leurs ID.

Exemple:

```
from odoo import models

class MyModel(models.Model):
    _name = 'my.model'
    def get_record(self, record_id):
        return self.browse(record_id)
```

Explication:La méthode `browse` prend un ID (ou une liste d'IDs) et renvoie les enregistrements correspondants.

6. **`read`**

Description:

Permet de lire les valeurs des champs d'enregistrements spécifiques.

Exemple:

```
from odoo import models
class MyModel(models.Model):
    _name = 'my.model'
    def get_record_values(self, record_id):
        record = self.browse(record_id)
        return record.read(['name'])
```

Explication:

La méthode `read` prend une liste de noms de champs et renvoie les valeurs de ces champs pour les enregistrements spécifiés

7. **`name_get`**

Description:

Utilisée pour définir comment les enregistrements doivent être affichés dans les sélecteurs et autres éléments de l'interface utilisateur.

Exemple:

```
from odoo import models, fields
class MyModel(models.Model):
    _name = 'my.model'
    name = fields.Char()
    code = fields.Char()
    def name_get(self):
```

```
        result = []
        for record in self:
            name = f"{record.code} - {record.name}"
            result.append((record.id, name))
        return result
```

Explication:

La méthode `name_get` personnalise l'affichage des enregistrements. Elle renvoie une liste de tuples avec l'ID de l'enregistrement et le nom affiché.

8. **`default_get`**

Description:

Permet de définir des valeurs par défaut pour les champs d'un nouvel enregistrement.

Exemple:```python

```
from odoo import models, fields
class MyModel(models.Model):
    _name = 'my.model'
    name = fields.Char()
    @api.model
    def default_get(self, fields_list):
        defaults = super(MyModel, self).default_get(fields_list)
        defaults.update({'name': 'Default Value'})
        return defaults
```

```
```

Explication:

La méthode `default_get` est appelée pour obtenir des valeurs par défaut pour les champs lors de la création d'un nouvel enregistrement.

9. **`fields_get`**

Description:

Permet de récupérer des informations sur les champs d'un modèle, comme leur type et leurs propriétés.

Exemple:

```
from odoo import models

class MyModel(models.Model):
    _name = 'my.model'
    def get_fields_info(self):
        return self.fields_get()
```

Explication:

La méthode `fields_get` renvoie un dictionnaire contenant des informations sur les champs du modèle, utile pour des personnalisations ou des rapports.

---

10. **`name_search`**

Description:

Permet d'effectuer une recherche de texte dans les noms des enregistrements pour les utiliser dans des sélecteurs ou des filtres.

Exemple:

```
from odoo import models, fields

class MyModel(models.Model):
    _name = 'my.model'
    name = fields.Char()
    def name_search(self, name='', args=None, operator='ilike', limit=100):
        return super(MyModel, self).name_search(name, args, operator, limit)
```

Explication:

La méthode `name_search` est utilisée pour rechercher des enregistrements en fonction du texte saisi dans les champs de nom. Vous pouvez personnaliser les critères de recherche et le nombre de résultats.

Ces méthodes permettent de gérer et de personnaliser les enregistrements dans Odoo, en vous offrant des moyens de manipuler, rechercher, créer, mettre à jour et supprimer des données. Elles sont essentielles pour le développement et la personnalisation des modules Odoo.