onnet

2021
VIE

Odoo ORM: Common ORM

www.on.net.vn
www.arrowhitech.com

onnet
BY AHT TECH JSC

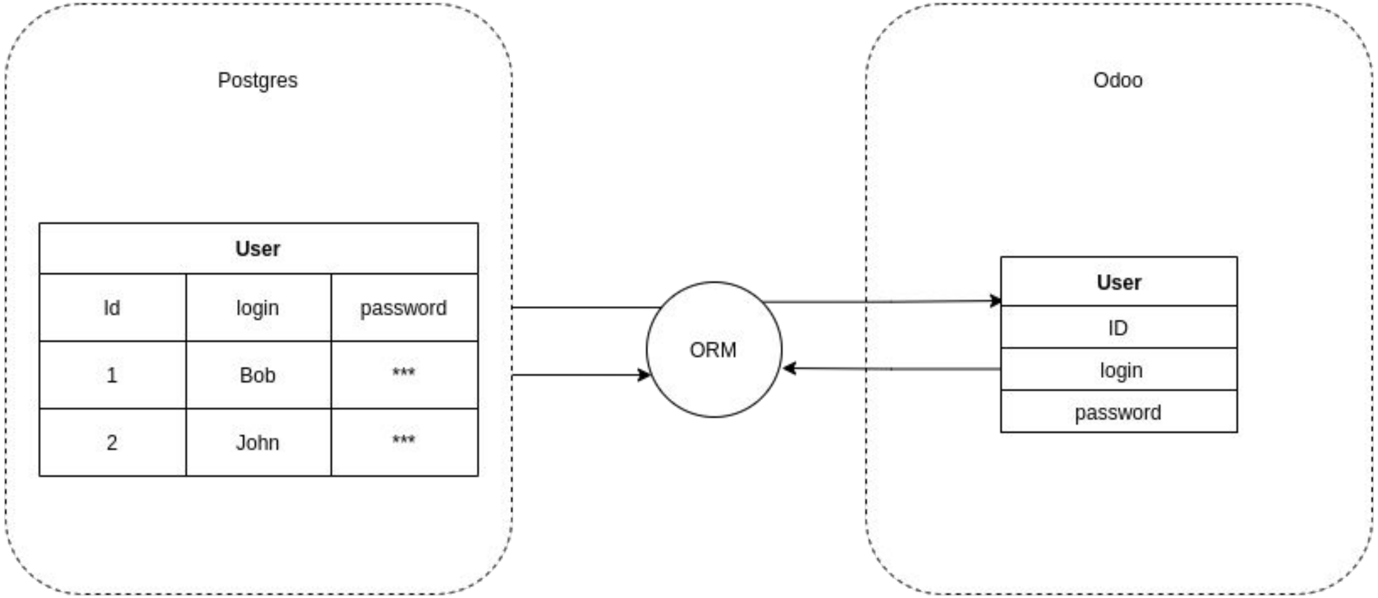
Table of contents

Introduction

Object–relational mapping (**ORM**, **O/RM**, and **O/R mapping tool**) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages.

This creates, in effect, a **"virtual object database"** that can be used from within the programming language.

Introduction



User		
Id	login	password
1	Bob	***
2	John	***

Recordsets

- An **ordered collection of records** of the same model.
- Interactions with **models** and **records** are performed through **recordsets**

Recordsets

Methods defined on a model are **executed** on a **recordset**, and their **self** is a recordset:

```
class AModel(models.Model):

    _name = 'a.model'

    def a_method(self):

        # self can be anything between 0 records and all records in the database
        self.do_operation()
```

Recordsets

Iterating on a **recordset** will yield new **sets of *a single record*** (“**singletons**”):

```
def do_operation(self):
    print(self) # => a.model(1, 2, 3, 4, 5)
    for record in self:
        print(record) # => a.model(1), then a.model(2), then a.model(3), ...
```

Recordsets: Field access

Recordsets provide an **“Active Record”** interface: model fields can be **read** and **written** directly from the record as **attributes**.

Field values can also be accessed like dict items, which is more elegant and safer than `getattr()` for dynamic field names. Setting a field’s value triggers an update to the database:

```
>>> record.name

Example Name

>>> record.company_id.name

Company Name

>>> record.name = "Bob"

>>> field = "name"

>>> record[field]

Bob
```

Recordsets: Record cache and prefetching

Odoo maintains a **cache** for the **fields** of the **records**, so that **not** every field access issues a **database request**, which would be terrible for performance. The following example queries the database only for the first statement:

```
record.name             # first access reads value from database
record.name             # second access gets value from cache
```

Recordsets: Record cache and prefetching

To avoid reading one field on one record at a time, Odoo ***prefetches*** records and fields following some heuristics to get good performance.

Consider the following example, where `partners` is a recordset of 1000 records. Without prefetching, the loop would make 2000 queries to the database. With prefetching, only one query is made:

```
for partner in partners:
    print partner.name       # first pass prefetches 'name' and 'lang'
    print partner.lang       # (and other fields) on all 'partners'
```

Environments

The **Environment** stores various **contextual** data used by the **ORM**: the database cursor (for database queries), the current user (for access rights checking) and the current context (storing **arbitrary** metadata). The environment also stores caches.

All recordsets have an environment, which is immutable, can be accessed using **env** and gives access to:

- the current **user** (user)
- the cursor **(cr)**
- the superuser flag **(su)**
- or the context **(context)**

Example:

```
>>> records.env
<Environment object ...>
>>> records.env.user
res.user(3)
>>> records.env.cr
<Cursor object ...)
```

Environments

When creating a recordset from an other recordset, the environment is inherited. The environment can be used to get an empty recordset in an other model, and query that model:

```
>>> self.env['res.partner']

res.partner()

>>> self.env['res.partner'].search([['is_company', '=', True], ['customer', '=', True]])

res.partner(7, 18, 12, 14, 17, 19, 8, 31, 26, 16, 13, 20, 30, 22, 29, 15, 23, 28, 74)
```

Environments: Altering the environment

Model.with_context(*[context][, **overrides]*) → records

Returns a new version of this recordset attached to an extended context.

The extended context is either the provided `context` in which `overrides` are merged or the *current* context in which `overrides` are merged e.g.:

```
# current context is {'key1': True}

r2 = records.with_context({}, key2=True)

# -> r2._context is {'key2': True}

r2 = records.with_context(key2=True)

# -> r2._context is {'key1': True, 'key2': True}
```

Environments: Altering the environment

Model.with_user(*user*)

Return a new version of this recordset attached to the given user, in non-superuser mode, unless **user** is the superuser (by convention, the superuser is always in superuser mode.)

Model.with_company(*company*)

Return a new version of this recordset with a modified context, such that:

```
result.env.company = company
result.env.companies = self.env.companies | company
```

Model.with_env(*env*)

Return a new version of this recordset attached to the provided environment

Model.sudo([*flag=True*])

Returns a new version of this recordset with superuser mode enabled or disabled, depending on `flag`. The superuser mode does not change the current user, and simply bypasses access rights checks.

Environments: SQL Execution

The **cr** attribute on **environments** is the cursor for the current database transaction and allows **executing SQL directly,** either for queries which are difficult to express using the ORM (e.g. complex joins) or for performance reasons:

```
self.env.cr.execute("some_sql", params)
```

Environments: SQL Execution

Model.invalidate_cache(*fnames=None*, *ids=None*)

Invalidate the record caches after some records have been modified. If both **fnames** and **ids** are **None**, the whole cache is cleared.

Parameters

- **fnames** – the list of modified fields, or None for all fields
- **ids** – the list of modified record ids, or None for all

Common ORM: create

Model.create(*vals_list*) → records

Creates new records for the model.

The new records are initialized using the values from the list of dicts `vals_list`, and if necessary those from `default_get()`.

Parameters

vals_list (*list*) – values for the model's fields, as a list of dictionaries: `[{'field_name': field_value, ...}, ...]`

For backward compatibility, vals_list may be a dictionary. It is treated as a singleton list [vals], and a single record is returned.

Returns

The created records

Common ORM: write

Model.write(*vals*)

Updates all records in the current set with the provided values.

Parameters

vals (*dict*) – fields to update and the value to set on them.

e.g: `{'foo': 1, 'bar': "Qux"}` will set the field `foo` to `1` and the field `bar` to `"Qux"` if those are valid (otherwise it will trigger an error).

Common ORM: copy

Model.copy(*default=None*)

Duplicate record `self` updating it with default values

Parameters

default (*dict*) – dictionary of field values to override in the original values of the copied record, e.g: `{'field_name': overridden_value, ...}`

Returns

new record

Common ORM: default_get

Model.default_get(*fields_list*) → default_values

Return default values for the fields in fields_list. Default values are determined by the context, user defaults, and the model itself.

Parameters

fields_list (*list*) – names of field whose default is requested

Returns

a dictionary mapping field names to their corresponding default values, if they have a default value.

Common ORM: name_create

Model.name_create(*name*) → record

Create a new record by calling create() with only one value provided: the display name of the new record.

The new record will be initialized with any default values applicable to this model, or provided through the context. The usual behavior of create() applies.

Parameters

name – display name of the record to create

Returns

the name_get() pair value of the created record

Common ORM: browse

Model.browse([*ids*]) → records

Returns a recordset for the ids provided as parameter in the current environment.

```
>> self.browse([7, 18, 12])
>> res.partner(7, 18, 12)
```

Parameters

ids (*int* or *list*(*int*) or *None*) – id(s)

Returns

recordset

Common ORM: search

Model.search(*args[, offset=0][, limit=None][, order=None][, count=False]*)[source]

Searches for records based on the args search domain.

Parameters

- **args** – A search domain. Use an empty list to match all records.
- **offset** (*int*) – number of results to ignore (default: none)
- **limit** (*int*) – maximum number of records to return (default: all)
- **order** (*str*) – sort string
- **count** (*bool*) – if True, only counts and returns the number of matching records (default: False)

Returns

at most limit records matching the search criteria

Common ORM: search_count

Model.search_count(*args*) → int

Returns the number of records in the current model matching the provided domain.

Common ORM: name_search

Model.name_search(*name='', args=None, operator='ilike', limit=100*) → records

Search for records that have **a display name** matching the given **name** pattern when compared with the given **operator**, while also matching the optional search domain (args).

This is used for example to provide suggestions based on a partial value for a relational field. Sometimes be seen as the inverse function of name_get(), but it is not guaranteed to be.

This method is equivalent to calling search() with a search domain based on display_name and then name_get() on the result of the search.

Parameters

- **name** (*str*) – the name pattern to match
- **args** (*list*) – optional search domain (see search() for syntax), specifying further restrictions
- **operator** (*str*) – domain operator for matching name, such as 'like' or '='.
- **limit** (*int*) – optional max number of records to return

Returns

list of pairs (id, text_repr) for all matching records.

Common ORM: read

Model.read(**[***fields***]**)

Reads the requested fields for the records in self, low-level/RPC method. In Python code, prefer browse().

Parameters

fields – list of field names to return (default is all fields)

Returns

a **list of dictionaries** mapping field names to their values, with one dictionary per record

Common ORM: filtered

Model.filtered(*func*)

Return the records in self satisfying func.

Parameters

func (*callable or str*) – a function or a dot-separated sequence of field names

Returns

recordset of records satisfying func, may be empty.

Example:

```
# only keep records whose company is the current user's
records.filtered(lambda r: r.company_id == user.company_id)

# only keep records whose partner is a company
records.filtered("partner_id.is_company")
```

-

Common ORM: mapped

Model.mapped(*func*)

Apply func on all records in self, and return the result as a list or a recordset (if func return recordsets). In the latter case, the order of the returned recordset is arbitrary.

Parameters

func (*callable or str*) – a function or a dot-separated sequence of field names

Returns

Returns a list of summing two fields for each record in the set:

```
records.mapped(lambda r: r.field1 + r.field2)
```

The provided function can be a string to get field values:

```
# returns a list of names
records.mapped('name')
# returns a recordset of partners
records.mapped('partner_id')
# returns the union of all partner banks, with duplicates removed
records.mapped('partner_id.bank_ids')
```

Q&A