SIMULATING NON RELATIONAL DATABASE USING DJANGO OBJECT-RELATIONAL MAPPING

Alexandr TRUHIN, Ana BALICA, Mihail KULEV

Universitatea Tehnică a Moldovei

Abstract: *Most of the shared Web hosting today comes up bundled to a relational database - this is the most often case that can solve a big set of problems. Nevertheless sometimes a different schema is being desired and designed. For this purpose we want to show how to build a non-relational database on top of a classic relational one using Object Relational Mapping (further ORM) in Django.*

Index Terms: *Django Web Framework, Django Models, NoSQL databases, ORM (Object-Relational Mapping), relational database.*

1. Introduction

The choice of a Web framework nowadays is wide enough. One of the popular ones today is Django - Python Web framework for perfectionists with deadlines. Although a database is not always required for a website, Django comes with an ORM in which one can describe database layout using just Python. The main idea behind using an ORM is being consistent across different database platforms. Via ORM the developer also takes advantage of the reusability of code, less amount of code, better version control.

Django models are ordinary classes that represent a description of your data. They contain the necessary fields and the expected behaviour. Nevertheless the simple model tools aren’t always enough to depict the data and the relations between it.

2. Motivation

In almost every project there are entities that have multiple parameters. Usual solution in this case is to have one or more related tables that have one column per one parameter/attribute. In some projects there are entities that can have a lot of attributes, or many attributes that are unknown at the stage of development. In that case it is not reasonable to create one column for each property as classical relational databases scale vertically badly. There is an overcome for this problem in OLAP cubes, but they are complex and not so popular among small size projects. One way to solve this problem is to store all additional attributes in some specific format (for example JSON), but in this case searching and sorting will be much slower. The ”right” way to deal with such a situation is to use NoSQL database (for ex. MongoDB). In such databases vertical scaling is done without any loss of performance as each row is an independent entity. But such databases are in progress to become popular, and not any server has it. In such case we can simulate NoSQL relations by providing one more abstraction layer that will handle operations with relational DB.

3. Django models

Using standard Django models we’ll have for each model one table with all necessary fields. It works well when necessary fields are known a priori. But when not, it will cause exponential space usage growth. It will happen because when you’ll need to add one new field to one specific row, you’ll have to change the structure of the whole table. Simply explained - relational DBs have very bad horizontal scalability.

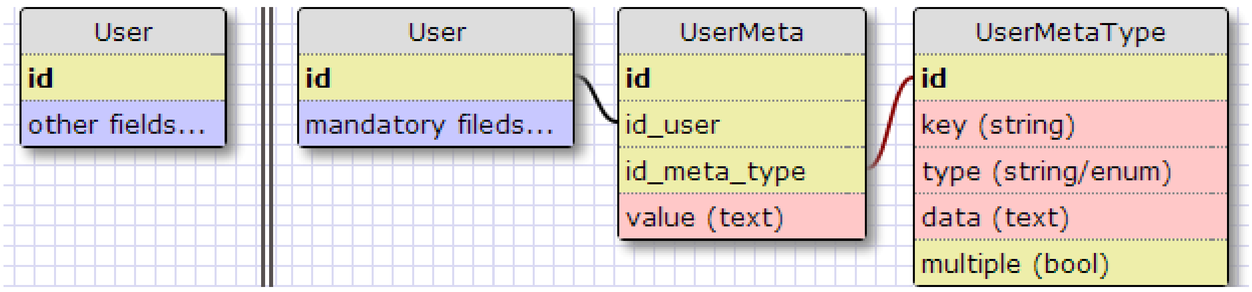


Fig. 1 Default Django Model’s DB Schema (on the left) and Extended Django Model’s DB Schema to support non-relational like structure

Instead of scaling horizontally, we can make use of relational DBs advantage - good vertical scaling. We'll create two more additional tables that will store user meta data (attributes) and meta types (Fig. 1). If all user's attributes can be generalized by one type, then it may be a good idea to skip users meta type table.

UserMetaType table is used for keeping meta types such as string, number, choise and others. It stores information about attribute name (key), available choices for each type and if a given attribute can have multiple values for one user. *UserMeta* keeps attribute value and knows its attribution to user and meta type.

This structure is handled on data layer by our new extended class. Using this structure is the same as using simple object: we can assign values to attributes, we can rewrite them and delete. This is done for simplicity. In python it's easy to do that by rewriting default attribute access functions. So when we make some actions with attributes of our extended model object, python interpreter sends this request to these functions. This object has only one real attribute - user. It keeps user object to access its attributes as a proxy and to know to whom to set new attributes.

```
class UserExtended():
  def __init__(self, user_id=0):
    try:
      self.user =
User.objects.get(id=user_id)
    except User.DoesNotExist:
      self.user = User()

  def __setattr__(self, key, value):
    if key == "user":
      self.__dict__[key] = value
    elif hasattr(self.user, key):
      setattr(self.user, key, value)
      self.user.save()
    else:
      return self.setMeta(key, value)
```

```
  def __getattr__(self, key):
    if hasattr(self.user, key):
      return getattr(self.user, key)
    else:
      return self.getMeta(key)
  def __delattr__(self, key):
    if key == "user":
      del self.__dict__[key]
    else:
      self.delMeta(key)

  def __del__(self):
    self.user.delete()
UserMeta.objects.filter(user=self.us
er).delete()
```

Fig. 2 UserExtended basic class structure

If default user object has no necessary attributes, then internal class methods are used to store, retrieve and remove attributes into *UserMeta* table. *setMeta* method checks if passed value corresponds to attribute type (stored in UserMetaType). *getMeta* method may return a value, or an array of values if given attribute type has *multiple* parameter set to true.

```
def setMeta(self, key, value):
  try:
    user_meta =
UserMeta.objects.get(user=self.user,
key=key)
  except UserMeta.DoesNotExist:
    user_meta =
UserMeta(user=self.user, key=key) #
new meta
  except:
    raise AttributeError("unknown
error")
  user_meta.value = value
  user_meta.save()
```

```
def getMeta(self, key):
  try:
    meta =
UserMeta.objects.get(user=self.user,
key=key)
    return meta.value
  except:
    raise AttributeError("error")

def delMeta(self, key):
  try:
    return
UserMeta.objects.filter(user=self.use
r, key=key).delete()
  except:
    return False
```

Fig. 3 Internal generalized class methods used to store and retrieve attributes values from DB

Now we can easily work on user object (Fig.4). Given structure may have other helper methods, as for example a method for retrieving all available attributes of one user. But those are out of scope for this paper. Anyway it's important to understand that this is just a skeleton of the idea and for real use cases it will require many more helper methods.

```
user = UserExtended(user_id)  Load user
user = UserExtended()         Load empty user
user.age                      Get user age attribute (string or list of
strings)
user.hasattr('age')           Check if attribute age is defined
user.age = 25                 Update user age attribute
user.type = ['student', 'alumni'] Update user type attributes (if multiple)
del user.age                  Delete user age attribute
del user                      Delete user and all its meta
```
Fig. 4 Example of using basic methods on a user

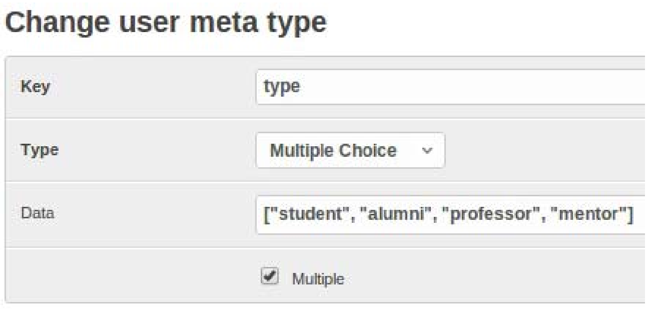
3. Admin panel

Django has in its core a large suite of functionalities, including an admin panel, which is simply added to the installed apps. The admin page will list all our data. If we have a standard use case using plain Django Models, then this is it - nothing more to be done. But since we decided to enter and manipulate the data in a different manner, then it is necessary to make the corresponding changes.

First of all it is important to have a correct way of saving data field in *UserMetaTypes*. On the admin page we added some help notes, where it is said that data field is used just in the case we define our type as Multiple Choice. Moreover our data should be listed in one row without spaces by listing all the possible choices and separating them by commas. In the *UserMetaTypeAdmin* class we are overwriting the *save_model* method by saving this data as a JSON object.

```
def save_model(self, request, obj, form, change):
  if obj.data:
    try:
      json.loads(obj.data)
    except ValueError:
      obj.data = json.dumps(obj.data.split(','))
  obj.save()
```
Fig. 5 The body of save_model method

And this is how it looks like.


Change user meta type

Key	type
Type	Multiple Choice
Data	["student", "alumni", "professor", "mentor"]
	✓ Multiple

Fig. 6 Admin interface “Change user meta type”

Second thing to pay attention to is the UserAdmin implementation.

```
class UserAdmin(admin.ModelAdmin):
  list_display = search_fields = ('name', 'surname')
  form = UserAdminForm
```

Fig. 7 A part of UserAdmin class

Obviously here we state that we want our database tuples to be represented by their name and surname, and making those also search fields. Though the ModelForm can be created dynamically to represent our Add/Create page, we built a custom one.

The custom form initializes the UserExtended class, extracts all the UserMeta and UserMetaTypes for a specific record. If a new User record is created then a dictionary of all defined UserMetaTypes will appear. If User already exists then not only the UserMetaTypes will be returned, but all the data related to this record.

The templates are also overwritten. There are 3 sections: basic (static) information about the user, Meta and Add Meta. Meta component contains the available data about the particular record - the name of the meta type, the input box and the data queried from the database. Add Meta has a list of all other

UserMetaType objects that we previously defined. By clicking on one of them, the UserMetaType is added to the Meta section with a blank input box. If one wants to delete a UserMetaType for the User, which is being edited at the moment, then there is a close button on the right of each input.

In order to achieve the following, we have overwritten a specific template in the admin app, used some extra styling and a bit of jQuery to manipulate the DOM. Here is presented a short snippet that creates the list of UserMetaTypes.

```
{% if meta.type == 'choice' %}
  <ul>{% for data in meta.data %}
    <li><label for="{{ data }}">
      <input type="checkbox"  name="{{ key }}" value="{{ data }}" id="{{
data }}" {% if data in meta.value%}checked="checked"{% endif %}> {{
data|normalize }}
    </label></li>
  {% endfor %}</ul>
{% elif meta.type == 'textarea' %}
  <textarea cols="20" id="{{ key }}" type="text">{{ meta.value
}}</textarea>
{% else %}
  <input name="{{ key }}" value="{{ meta.value }}" class="vTextField"
type="text" id="{{ key }}">
{% endif %}
```

Fig. 8 The template for displaying input boxes

All those are created at once, by checking the type of meta data and building a unique type of input: checkbox, text area or a single input row. Next we are hiding all the meta data that is blank for User, create *click* events for all the meta keys in the Add Meta section to be discovered in the Meta section and also create a *click* event for the *close button* to hide the meta and to add the deleted Meta for the meta fieldset.

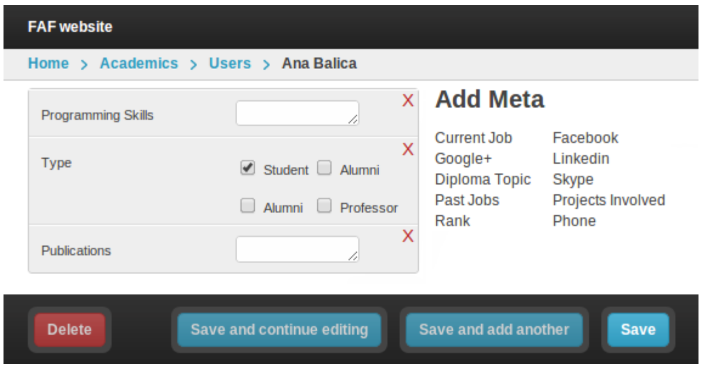


Fig. 9 Admin page “Change user”

To save the changed data we check if the meta type is not in the POST request and delete it using method defined in the *UserExtended* class. Otherwise the data (updated or not) will be saved in the database.

4. Conclusion

As a result we have achieved an easy scalable database that doesn’t require a schema update with every new attribute. We have kept the same level of performance and provided an intuitive interface for non-technical people to add, change, remove records in the database.

References

1.*The Django admin site*, Available: https://docs.djangoproject.com/en/dev/ref/contrib/admin/
2.*Django advanced templates*, Available: http://www.djangobook.com/en/2.0/chapter09.html
3.*Python implementing descriptors*, Available: http://docs.python.org/2/reference/datamodel.html#descriptors
4.*NoSQL database article*, Available: http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html