django-hierarkey Documentation

Release 1.1

Raphael Michel

Mar 07, 2022

Contents

This package allows you to attach a key-value store to a model, e.g. to store preferences of a user or a customer. The package supports arbitrary datatypes, defaults and model hierarchies, i.e. you can define a different model instance as your instance's parent and the values of the parent instance will be used as default values for the child instances.

CHAPTER 1

Documentation content

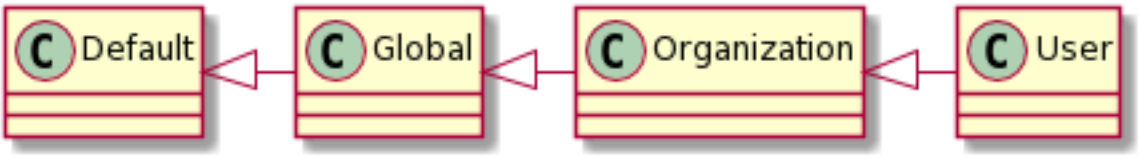
1.1 Concepts

Hierarkey is built to store key-value pairs. For example, you could use it to store arbitrary preferences of your users that you do not want to create model fields for because you have too many of them or because you generate them dynamically.

1.1.1 Storage hierarchy

Hierarkey is built as a hierarchical store. For example, in your application users might be associated with an organization. In that case, you can store key-value pairs with the user, the organization or globally for your application.

If you fetch a value for a user, the value set for the user will be returned. If the value has never been set for the user, the value set for the organization will be returned. If the value has never been set for the organization, the globally configured value will be returned. If no value has ever been set, the hardcoded default will be returned. If no default exists for the key, `None` will be returned.



Note that this is only an example, you could build this hierarchy in any way you want.

1.1.2 Storage and data types

The key-value pairs are stored into one database model per level of your hierarchy. The values will be stored in a `TextField`, so they will need to be serialized first. When querying values, they need to be deserialized to get the original data type. If you have hardcoded a default value for a key, it will have a type associated. In this case, if you don't need to pass a type when querying, the hardcoded type will be used for deserialization. Otherwise, as hierarkey

is not able to detect the data type from the saved data, you need to pass the desired data type to the query function as `as_type` (see *the API reference*).

Currently, the following data types are supported out of the box:

- `str`, `bool`, `int`, `float`
- `list` and `dict` with members that are serializable
- `Decimal`
- `datetime`, `time`, `date`
- Any Django model instances (only the primary key is stored, so this behaves like an unconstrained foreign key)
- `django.core.files.File`

Hierarkey is built in a way that allows you to easily add custom defaults and *your own data types*.

1.2 Getting started

1.2.1 Install hierarkey

You only need to install the hierarkey package, for example using pip:

```
$ pip install django-hierarkey
```

That's it!

1.2.2 Attach a storage to a single models

As a first example, we will attach a storage object to a single model. For example, you could use this to store settings for every user. If you want to use the hierarchical features of hierarkey, just skip to the next example.

First of all, in your `models.py`, we will create a `Hierarkey` object:

```
from hierarkey.models import Hierarkey

hierarkey = Hierarkey(attribute_name='settings')
```

The attribute name defines the name that you will use later to access the storage. This allows you to use multiple separated storage hierarchies within one model, if you want to. Next, add a decorator to the model you want to associate the storage with:

```
@hierarkey.add()
class User(models.Model):
    ....
```

Note: The parentheses in the call to the `@hierarkey.add()` decorator are required. Refer to the *API documentation* for more information on its parameters.s

Now, you need to create a database migration and apply it:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

1.2.3 Build a hierarchical storage

To build a hierarchy, we need multiple models. In our example, we will have a three-level hierarchy, consisting of global settings, organization settings and user settings. You can use more levels if you want to, or you can omit the global settings. There can only be one level of global settings.

As in the previous example, we first create a `Hierarkey` object:

```
from hierarkey.models import GlobalSettingsBase, Hierarkey

hierarkey = Hierarkey(attribute_name='settings')
```

Next, we define a class, and attach the global settings. This class can be empty and is not needed except for consistency in the hierarkey API. You can define it like this:

```
@hierarkey.set_global()
class GlobalSettings(GlobalSettingsBase):
    pass
```

Then, we add our other two layers. The organization layer works the same way as in the first example above. In contrast, for the user model we have to specify the name of the parent relation:

```
@hierarkey.add()
class Organization(models.Model):
    ...

@hierarkey.add(parent_field='organization')
class User(models.Model):
    organization = models.ForeignKey(Organization)
    ...
```

Now, you need to create a database migration and apply it:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

1.2.4 Set default values

You can add default values by calling a method on the `Hierarkey` object specifying the key, the value and the type:

```
hierarkey.add_default('key', 'value', bool)
```

1.2.5 Access the settings storage

On an instance of your model, you can access the settings storage under the name you specified as the `attribute_name` further above.

You can read and write the values in the key-value store in three ways. First, by attribute access:

```
print(user.settings.theme)
user.settings.theme = 'dark'
del user.settings.theme
```

Note: Attribute access is unsupported for key starting with an underscore.

Second, by item access:

```
print(user.settings['theme'])
user.settings['theme'] = 'dark'
del user.settings['theme']
```

And third, using explicit methods:

```
print(user.settings.get('theme'))
user.settings.set('theme', 'dark')
user.settings.delete('theme')
```

All changes are written to the database instantly, while values are read eagerly and are being cached.

Deserialization will only be automatically performed for keys that have a default value specified in code. If you want to deserialize other keys, you need to use the explicit getter methods and specify the type yourself:

```
user.settings.get('theme', as_type=int)
```

To access the global settings, you can instantiate the global settings class you defined before:

```
GlobalSettings().settings.get(...)
```

1.2.6 Next steps

You can now continue reading either about *Forms* or in the *API Reference*.

1.3 API Reference

Everything not listed on this page is considered a private API and should not be called from outside.

1.3.1 Model definition

class hierarkey.models.**Hierarkey**(*attribute_name*)

The Hierarkey object represents one complete key-value store hierarchy. It can have one global and multiple object-level storages attached and holds default values as well as custom type serialization info.

Parameters **attribute_name** – The name for the attribute on the model instances that will allow access to the storage, e.g. `settings`.

add(*cache_namespace: str = None*, *parent_field: str = None*) → type

Decorator. Attaches a global key-value store to a Django model.

Parameters

- **cache_namespace** – Optional. A custom namespace used for caching. By default this is constructed from the name of the class this is applied to and the `attribute_name` of this `Hierarkey` object.
- **parent_field** – Optional. The name of a field of this model that refers to the parent in the hierarchy. This must be a `ForeignKey` field.

add_default(*key: str, value: Optional[str], default_type: type = <class 'str'>*) → None
Adds a default value and a default type for a key.

Parameters

- **key** – Key
- **value** – *Serialized* default value, i.e. a string or `None`.
- **default_type** – The type to deserialize values for this key to, defaults to `str`.

add_type(*type: type, serialize: Callable[[Any], str], unserialize: Callable[[str], Any]*) → None
Adds serialization support for a new type.

Parameters

- **type** – The type to add support for.
- **serialize** – A callable that takes an object of type `type` and returns a string.
- **unserialize** – A callable that takes a string and returns an object of type `type`.

set_global(*cache_namespace: str = None*) → type
Decorator. Attaches the global key-value store of this hierarchy to an object.

Parameters **cache_namespace** – Optional. A custom namespace used for caching. By default this is constructed from the name of the class this is applied to and the `attribute_name` of this `Hierarkey` object.

class hierarkey.models.**GlobalSettingsBase**
Base class for objects with a global settings storage attached. This class does not add any functionality, it only makes global settings behave more consistent to object-level settings.

1.3.2 Storage access

class hierarkey.proxy.**HierarkeyProxy**
If you add a hierarkey storage to a model, the model will get a new attribute (e.g. `settings`) containing a key-value store that is managed by this class.

This class allows access to settings via attribute access, item access or using the documented methods. You should not instantiate this class yourself.

delete(*key: str*) → None
Deletes a setting from this object's storage.

The write to the database is performed immediately and the cache in the cache backend is flushed. The cache within this object will be updated correctly.

flush() → None
Discards both the state within this object as well as the cache in Django's cache backend.

freeze() → dict
Returns a dictionary of all settings set for this object, including any values of its parents or hardcoded defaults.

get(*key: str, default=None, as_type: type = None, binary_file: bool = False*)
Get a setting specified by `key`. Normally, settings are strings, but if you put non-strings into the settings object, you can request deserialization by specifying `as_type`. If the key does not have a harcdoded default type, omitting `as_type` always will get you a string.

If the setting with the specified name does not exist on this object, any parent object up to the global settings layer (if configured) will be queried. If still no value is found, a default value set in ths source

code will be returned if one exists. If not, the value of the `default` argument of this method will be returned instead.

If you receive a `File` object, it will already be opened. You can specify the `binary_file` flag to indicate that it should be opened in binary mode.

set(*key: str*, *value: Any*) → None
Stores a setting in the database and connects it to its object.

The write to the database is performed immediately and the cache in the cache backend is flushed. The cache within this object will be updated correctly.

1.3.3 Forms

class hierarkey.forms.**HierarkeyForm**(**args*, *obj*, *attribute_name*, ***kwargs*)
This is a custom subclass of `django.forms.Form` that you can use to set values for any keys. See the Forms chapter of the documentation for more details.

get_new_filename(*name: str*) → str
Returns the file name to use based on the original filename of an uploaded file. By default, the file name is constructed as:

```
<model_name>-<attribute_name>/<primary_key>/<original_basename>.<random_nonce>
↪.<extension>
```

save() → None
Saves all changed values to the database.

1.4 Forms

Hierarkey provides a form base class to manipulate values of a key-value store. You just define your form as you normally would:

```
from django import forms
from hierarkey.forms import HierarkeyForm

class MySettingsForm(HierarkeyForm):
    theme = forms.ChoiceField(
        choices=(('light', 'light'), ('dark', 'dark')
    )
    ...
```

You can use any form field that results in a data type that can be serialized and deserialized by hierarkey. This includes most form fields defined by Django and even *custom types*.

To save the data, you can call the `save()` method on the form. Note that the form takes two additional arguments: The object that you want to update the values for (i.e. a `GlobalSettings` object or a model instance) and the `attribute_name` the storage is located at. When using a class-based `FormView`, you could integrate it like this:

```
from django.views.generic import FormView

class MySettingsView(FormView):
    form_class = MySettingsForm

    def get_form_kwargs(self):
```

(continues on next page)

(continued from previous page)

```
        kwargs = super().get_form_kwargs()
        kwargs['attribute_name'] = 'settings'
        kwargs['obj'] = User.objects.get(...)
        return kwargs

    def form_valid(self, form):
        form.save()
        return super().form_valid(form)
```

Note: Initial values for the form will be taken from the settings storage, i.e. from the given object and the global and hardcoded defaults. Initial values set directly on the form or field layer are not supported.

1.5 Custom types

You can add support for serializing and deserializing custom types like this:

```
class MyMessageType:
    def __init__(self, msg):
        self.msg = msg

hierarkey.add_type(
    MyMessageType,
    lambda v: v.foo,
    lambda v: MyMessageType(v)
)

...

# Serialize
user.settings.set('myproperty', MyMessageType('Hello'))

# Deserizalize
# will return MyMessageType('Hello')
user.settings.get('myproperty', as_type=MyMessageType)
```

1.6 File handling

Hierarkey has rudimentary support for saving a `file` into the key-value storage. In this case, not the *content* of the file will be saved in the key-value store. Instead, only the *name* of the file within the configured Django storage backend will be saved.

You need to save the file to the storage backend yourself and then pass the `File` object to hierarkey. When you access the key in the store, the `file://` prefix will automatically be detected and hierarkey will use your default storage backend to open the file for you. The `binary_file` flag of the `get()` method allows you to open the file in binary mode.

When you use our *forms support*, this is done automatically for you. You can just specify a normal `django.forms.FileField` field on the model and `HierarkeyForm` will deal with storing the file to the default storage backend as well as deleting and replacing files. The filename will be automatically generated based on the pri-

mary key of your model, the key in the storage, and a random nonce. You can change this behaviour by overriding *get_new_filename()* on your form.

CHAPTER 2

Author and License

This package has been created by Raphael Michel and is published under the terms of the Apache License 2.0.

Index