Django Nani Documentation

Release 0.1.4

Jonas Obrist
contributors

December 14, 2013

Contents

Warning: The previously used `nani` package name is now deprecated, please upgrade your code to use `hvad`.

CHAPTER 1

About this project

django-hvad provides a high level API to maintain multilingual content in your database using the Django ORM.

CHAPTER 2

Before you dive into this

Please note that this documentation assumes that you are very familiar with Django and Python, if you are not, please familiarize yourself with those first.

While django-hvad tries to be as simple to use as possible, it's still recommended that you only use it if you consider yourself to be very strong in Python and Django.

CHAPTER 3

Notes on Django versions

django-hvad is tested on python 2.6 and 2.7 with django 1.2.7, 1.3.1 and 1.4. These should all work as expected, but for django 1.2.x you need you need to install django-cbv to use the class based views.

CHAPTER 4

Contents

4.1 Installation

4.1.1 Requirements

- Django 1.2.7 or higher
- Python 2.5 or a higher release of Python 2.x or PyPy 1.5 or higher, Python 3.x is not supported (yet).
- For Django 1.2.x you need django-cbv

4.1.2 Installation

Install django-hvad using pip by running `pip install django-hvad`. Then add `'hvad'` to your `INSTALLED_APPS` to make the admin templates work.

4.2 Quickstart

4.2.1 Define a multilingual model

Defining a multilingual model is very similar to defining a normal Django model, with the difference that instead of subclassing `django.db.models.Model` you have to subclass `hvad.models.TranslatableModel` and that all fields which should be translatable have to be wrapped inside a `hvad.models.TranslatedFields`.

Let's write an easy model which describes Django applications with translatable descriptions and information about who wrote the description:

```
from django.db import models
from hvad.models import TranslatableModel, TranslatedFields


class DjangoApplication(TranslatableModel):
    name = models.CharField(max_length=255, unique=True)
    author = models.CharField(max_length=255)

    translations = TranslatedFields(
```

```
        description = models.TextField(),
        description_author = models.CharField(max_length=255),
    )

    def __unicode__(self):
        return self.name
```

The fields `name` and `author` will not get translated, the fields `description` and `description_author` will.

4.2.2 Using multilingual models

Now that we have defined our model, let's play around with it a bit. The following code examples are taken from a Python shell.

Import our model:

```
>>> from myapp.models import DjangoApplication
```

Create an **untranslated** instance:

```
>>> hvad = DjangoApplication.objects.create(name='django-hvad', author='Jonas Obrist')
>>> hvad.name
'django-hvad'
>>> hvad.author
'Jonas Obrist'
```

Turn the **untranslated** instance into a **translated** instance with the language `'en'` (English):

```
>>> hvad.translate('en')
<DjangoApplication: django-hvad>
```

Set some translated fields and save the instance:

```
>>> hvad.description = 'A project to do multilingual models in Django'
>>> hvad.description_author = 'Jonas Obrist'
>>> hvad.save()
```

Get the instance again and check that the fields are correct:

```
>>> obj = DjangoApplication.objects.language('en').get(name='django-hvad')
>>> obj.name
u'django-hvad'
>>> obj.author
u'Jonas Obrist'
>>> obj.description
u'A project to do multilingual models in Django'
>>> obj.description_author
u'Jonas Obrist'
```

Note: I use `hvad.manager.TranslationQueryset.language()` here because I'm in an interactive shell, which is not necessarily in English, in your normal views, you can usually omit the call to that method, since the environment should already be in a valid language when in a request/response cycle.

Let's get all Django applications which have a description written by `'Jonas Obrist'` (in English):

```
>>> DjangoApplication.objects.language('en').filter(description_author='Jonas Obrist')
[<DjangoApplication: django-hvad>]
```

4.3 Models

4.3.1 Defining models

Models which have fields that should be translatable have to inherit `hvad.models.TranslatableModel` instead of `django.db.models.Model`. Their default manager (usually the `objects` attribute) must be an instance of `hvad.manager.TranslationManager` or a subclass of that class. Your inner `Meta` class on the model may not use any translated fields in it's options.

Fields to be translated have to be wrapped in a `hvad.models.TranslatedFields` instance which has to be assigned to an attribute on your model. That attribute will be the reversed ForeignKey from the *Translations Model* to your *Shared Model*.

If you want to customize your *Translations Model* using directives on a inner `Meta` class, you can do so by passing a dictionary holding the directives as the `meta` keyword to `hvad.models.TranslatedFields`.

A full example of a model with translations:

```
from django.db import models
from hvad.models import TranslatableModel, TranslatedFields


class TVSeries(TranslatableModel):
    distributor = models.CharField(max_length=255)

    translations = TranslatedFields(
        title = models.CharField(max_length=100),
        subtitle = models.CharField(max_length=255),
        released = models.DateTimeField(),
        meta={'unique_together': [('title', 'subtitle')]},
    )
```

4.3.2 New methods

translate

translate(*language_code*)
Returns this model instance for the language specified.

> **Warning:** This does **not** check if this language already exists in the database and assumes it doesn't! If it already exists and you try to save this instance, it will break!

Note: This method does not perform any database queries.

---

safe_translation_getter

safe_translation_getter(*name*, *default=None*)
Returns the value of the field specified by `name` if it's available on this instance in the currently cached language. It does not try to get the value from the database. Returns the value specified in `default` if no translation was cached on this instance or the translation does not have a value for this field.

This method is useful to safely get a value in methods such as `__unicode__()`.

Note: This method does not perform any database queries.

Example usage:

```
class MyModel(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=255)
    )

    def __unicode__(self):
        return self.safe_translation_getter('name', 'MyMode: %s' % self.pk)
```

lazy_translation_getter(*name*, *default=None*)
Returns the value of the field specified by `name` even thought it's not available on this instance in the currently cached language. Returns the value specified in `default` if no translation available on this instance or the translation does not have a value for this field.

This method is useful to get a value in methods such as `__unicode__()`.

Note: This method may perform database queries.

Example usage:

```
class MyModel(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=255)
    )

    def __unicode__(self):
        return self.lazy_translation_getter('name', 'MyMode: %s' % self.pk)
```

get_available_languages

get_available_languages()
Returns a list of available language codes for this instance.

Note: This method runs a database query to fetch the available languages.

4.3.3 Changed methods

save

save(*force_insert=False*, *force_update=False*, *using=None*)
This method runs an extra query when used to save the translation cached on this instance, if any translation was cached.

4.3.4 Working with relations

Foreign keys pointing to a *Translated Model* always point to the *Shared Model*. It is currently not possible to have a foreign key to a *Translations Model*.

Please note that `django.db.models.query.QuerySet.select_related()` used on a foreign key pointing to a *Translated Model* does not span to its *Translations Model* and therefore accessing a translated field over the relation causes an extra query.

If you wish to filter over a translated field over the relation from a *Normal Model* you have to use `hvad.utils.get_translation_aware_manager()` to get a manager that allows you to do so. That function takes your model class as argument and returns a manager that works with translated fields on related models.

4.4 Queryset API

4.4.1 TranslationQueryset

This is the queryset used by the `hvad.manager.TranslationManager`.

Performance consideration

While most methods on `hvad.manager.TranslationQueryset` querysets run using the same amount of queries as if they were untranslated, they all do slightly more complex queries (one extra join).

The following methods run two queries where standard querysets would run one:

- `hvad.manager.TranslationQueryset.create()`
- `hvad.manager.TranslationQueryset.update()` (only if both translated and untranslated fields are updated at once)

`hvad.manager.TranslationQueryset.get_or_create()` runs one query if the object exists, three queries if the object does not exist in this language, but in another language and four queries if the object does not exist at all. It will return `True` for created if either the shared or translated instance was created.

New methods

Methods described here are unique to django-hvad and cannot be used on normal querysets.

language

language(*language_code=None*)
Sets the language for the queryset to either the language code defined or the currently active language. This method should be used for all queries for which you want to have access to all fields on your model.

untranslated

untranslated()
Returns a `hvad.manager.FallbackQueryset` instance which by default does not fetch any translations. This is useful if you want a list of *Shared Model* instances, regardless of whether they're translated in any language.

Note: No translated fields can be used in any method of the queryset returned my this method. See *FallbackQueryset*

Note: This method is only available on the manager directly, not on a queryset.

delete_translations

delete_translations()
Deletes all *Translations Model* instances in a queryset, without deleting the *Shared Model* instances.

Not implemented public queryset methods

The following are methods on a queryset which are public APIs in Django, but are not implemented (yet) in django-hvad:

- `hvad.manager.TranslationQueryset.in_bulk()`
- `hvad.manager.TranslationQueryset.complex_filter()`
- `hvad.manager.TranslationQueryset.annotate()`
- `hvad.manager.TranslationQueryset.reverse()`
- `hvad.manager.TranslationQueryset.defer()`
- `hvad.manager.TranslationQueryset.only()`

Using any of these methods will raise a `NotImplementedError`.

4.4.2 FallbackQueryset

This is a queryset returned by *untranslated*, which can be used both to get the untranslated parts of models only or to use fallbacks. By default, only the untranslated parts of models are retrieved from the database.

Warning: You may not use any translated fields in any method on this queryset class.

New Methods

use_fallbacks

use_fallbacks(**fallbacks*)
Returns a queryset which will use fallbacks to get the translated part of the instances returned by this queryset. If `fallbacks` is given as a tuple of language codes, it will try to get the translations in the order specified. Otherwise the order of your LANGUAGES setting will be used.

Warning: Using fallbacks will cause **a lot** of queries! In the worst case 1 + (n * x) with n being the amount of rows being fetched and x the amount of languages given as fallbacks. Only ever use this method when absolutely necessary and on a queryset with as few results as possibel.

4.5 Forms

If you want to use your *Translated Model* in forms, you have to subclass `hvad.forms.TranslatableModelForm` instead of `django.forms.ModelForm`.

Please note that you should not override `hvad.forms.TranslatableModelForm.save()`, as it is a crucial part for the form to work.

4.6 Admin

When you want to use a *Translated Model* in the Django admin, you have to subclass `hvad.admin.TranslatableAdmin` instead of `django.contrib.admin.ModelAdmin`.

4.6.1 New methods

all_translations

all_translations(*obj*)
A method that can be used in `list_display` and shows a list of languages in which this object is available.

4.6.2 ModelAdmin APIs you should not change on TranslatableAdmin

Some public APIs on `django.contrib.admin.ModelAdmin` are crucial for `hvad.admin.TranslatableAdmin` to work and should not be altered in subclasses. Only do so if you have a good understanding of what the API you want to change does.

The list of APIs you should not alter is:

change_form_template

If you wish to alter the template to be used to render your admin, use the implicit template fallback in the Django admin by creating a template named `admin/<appname>/<modelname>/change_form.html` or `admin/<appname>/change_form.html`. The template used in django-hvad will automatically extend that template if it's available.

get_form

Use `hvad.admin.TranslatableAdmin.form` instead, but please see the notes regarding *Forms in admin*.

render_change_form

The only thing safe to alter in this method in subclasses is the context, but make sure you call this method on the superclass too. There's three variable names in the context you should not alter:

- `title`
- `language_tabs`
- `base_template`

get_object

Just don't try to change this.

queryset

If you alter this method, make sure to call it on the superclass too to prevent duplicate objects to show up in the changelist or change views raising `django.core.exceptions.MultipleObjectsReturned` errors.

4.6.3 Forms in admin

If you want to alter the form to be used on your `hvad.admin.TranslatableAdmin` subclass, it must inherit from `hvad.forms.TranslatableModelForm`. For more informations, see *Forms*.

4.6.4 ModelAdmin APIs not available on TranslatableAdmin

A list of public APIs on `django.contrib.admin.ModelAdmin` which are not implemented on `hvad.admin.TranslatableAdmin`.

- `list_display` [1]
- `list_display_links` [1]
- `list_filter` [1]
- `list_select_related` [1]
- `list_ediable` [1]
- `search_fields` [1]
- `date_hierarchy` [1]
- `actions` [1]

4.7 Release Notes

4.7.1 0.1.4 (Alpha)

Released on November 29, 2011

- Introduces `lazy_translation_getter()`

4.7.2 0.1.3 (Alpha)

Released on November 8, 2011

- A new setting was introduced to configure the table name separator, `NANI_TABLE_NAME_SEPARATOR`.

Note: If you upgrade from an earlier version, you'll have to rename your tables yourself (the general template is `appname_modelname_translation`) or set `NANI_TABLE_NAME_SEPARATOR` to the empty string in your settings (which was the implicit default until 0.1.0)

---

[1] This API can only be used with *Shared Fields*.

4.7.3 0.0.4 (Alpha)

In development

4.7.4 0.0.3 (Alpha)

Released on May 26, 2011.

- Replaced our ghetto fallback querying code with a simplified version of the logic used in Bert Constantins django-polymorphic, all credit for our now better FallbackQueryset code goes to him.
- Replaced all JSON fixtures for testing with Python fixtures, to keep tests maintainable.
- Nicer language tabs in admin thanks to the amazing help of Angelo Dini.
- Ability to delete translations from the admin.
- Changed hvad.admin.TranslatableAdmin.get_language_tabs signature.
- Removed tests from egg.
- Fixed some tests possibly leaking client state information.
- Fixed a critical bug in hvad.forms.TranslatableModelForm where attempting to save a translated model with a relation (FK) would cause IntegrityErrors when it's a new instance.
- Fixed a critical bug in hvad.models.TranslatableModelBase where certain field types on models would break the metaclass. (Many thanks to Kristian Oellegaard for the fix)
- Fixed a bug that prevented abstract TranslatableModel subclasses with no translated fields.

4.7.5 0.0.2 (Alpha)

Released on May 16, 2011.

- Removed language code field from admin.
- Fixed admin 'forgetting' selected language when editing an instance in another language than the UI language in admin.

4.7.6 0.0.1 (Alpha)

Released on May 13, 2011.

- First release, for testing purposes only.

4.8 Contact and support channels

- IRC: irc.freenode.net/#django-hvad
- Github: https://github.com/KristianOellegaard/django-hvad

4.9 How to contribute

4.9.1 Contributing Code

If you want to contribute code, one of the first things you should do is read the *Internal API Documentation*. It was written for developers who want to understand how things work.

Patches can be sent as pull requests on Github to https://github.com/KristianOellegaard/django-hvad.

Code Style

The **PEP 8** coding guidelines should be followed when contributing code to this project.

Patches **must** include unittests that fully cover the changes in the patch.

Patches **must** contain the necessary changes or additions to both the *internal* and *public* documentation.

If you need help with any of the above, feel free to *Contact and support channels* us.

4.9.2 Contributing Documentation

If you wish to contribute documentation, be it for fixes of typos and grammar or to cover the code you've written for your patch, or just generally improve our documentation, please follow the following style guidelines:

- Documentation is written using reStructuredText and Sphinx.
- Text should be wrapped at 80 characters per line. Only exception are over-long URLs that cannot fit on one line and code samples.
- The language does not have to be perfect, but please give your best.
- For section headlines, please use the following style:
  - # with overline, for parts
  - * with overline, for chapters
  - =, for sections
  - -, for subsections
  - ^, for subsubsections
  - ", for paragraphs

4.10 Internal API Documentation

4.10.1 About this part of the documentation

Warning: All APIs described in this part of the documentation which are not mentioned in the public API documentation are internal and are subject to change without prior notice. This part of the documentation is for developers who wish to work on django-hvad, not with it. It may also be useful to get a better insight on how things work and may proof helpful during troubleshooting.

4.10.2 Contents

This part of the documentation is grouped by file, not by topic.

General information on django-hvad internals

How it works

Model Definition The `hvad.models.TranslatableModelBase` metaclass scans all attributes on the model defined for instances of `hvad.models.TranslatedFields`, and if it finds one, sets the respective options onto meta.

`hvad.models.TranslatedFields` both creates the *Translations Model* and makes a foreign key from that model to point to the *Shared Model* which has the name of the attribute of the `hvad.models.TranslatedFields` instance as related name.

In the database, two tables are created:

- The table for the *Shared Model* with the normal Django way of defining the table name.
- The table for the *Translations Model*, which if not specified otherwise in the options (meta) of the *Translations Model* will have the name of the database table of the *Shared Model* suffixed by `_translations` as database table name.

Queries The main idea of django-hvad is that when you query the *Shared Model* using the Django ORM, what actually happens behind the scenes (in the queryset) is that it queries the *Translations Model* and selects the relation to the *Shared Model*. This means that model instances can only be queried if they have a translation in the language queried in, unless an alternative manager is used, for example by using `hvad.manager.FallbackManager.untranslated()`.

Due to the way the Django ORM works, this approach does not seem to be possible when querying from a *Normal Model*, even when using `hvad.utils.get_translations_aware_manager()` and therefore in that case we just add extra filters to limit the lookups to rows in the database where the *Translations Model* row existist in a specific language, using `<translations_accessor>__language_code=<current_language>`. This is suboptimal since it means that we use two different ways to query translations and should be changed if possible to use the same technique like when a *Translated Model* is queried.

A word on caching

Throughout this documentation, caching of translations is mentioned a lot. By this we don't mean proper caching using the Django cache framework, but rather caching the instance of the *Translations Model* on the instance of the *Shared Model* for easier access. This is done by setting the instance of the *Translations Model* on the attribute defined by the `translations_cache` on the *Shared Model*'s options (meta).

hvad.admin

hvad.admin.**translatable_modelform_factory**(*model*, *form=TranslatableModelForm*, *fields=None*, *exclude=None*, *formfield_callback=None*)

The same as `django.forms.models.modelform_factory()` but uses `type` instead of `django.forms.models.ModelFormMetaclass` to create the form.

TranslatableAdmin

class hvad.admin.**TranslatableAdmin**

A subclass of django.contrib.admin.ModelAdmin to be used for hvad.models.TranslatableModel subclasses.

query_language_key

The GET parameter to be used to switch the language, defaults to 'language', which results in GET parameters like ?language=en.

form

The form to be used for this admin class, defaults to hvad.forms.TranslatableModelForm and if overwritten should always be a subclass of that class.

change_form_template

We use 'admin/hvad/change_form.html' here which extends the correct template using the logic from django admin, see get_change_form_base_template(). This attribute should never change.

get_form(*self*, *request*, *obj=None*, ***kwargs*)

Returns a form created by translatable_modelform_factory().

all_translations(*self*, *obj*)

A helper method to be used in list_display to show available langauges.

render_change_form(*self*, *request*, *context*, *add=False*, *change=False*, *form_url=''*, *obj=None*)

Injects title, language_tabs and base_template into the context before calling the render_change_form() method on the super class. title just appends the current language to the end of the existing title in the context. language_tabs is the return value of get_language_tabs(), base_template is the return value of get_change_form_base_template().

queryset(*self*, *request*)

Calls hvad.manager.TranslationQueryset.language() with the current language from _language() on the queryset returned by the call to the super class and returns that queryset.

_language(*self*, *request*)

Returns the currently active language by trying to get the value from the GET parameters of the request using query_language_key or if that's not available, use django.utils.translations.get_language().

get_language_tabs(*self*, *request*, *available_languages*)

Returns a list of triples. The triple contains the URL for the change view for that language, the verbose name of the language and whether it's the current language, available or empty. This is used in the template to show the language tabs.

get_change_form_base_template(*self*)

Returns the appropriate base template to be used for this model. Tries the following templates:

- admin/<applabel>/<modelname>/change_form.html
- admin/<applabel>/change_form.html
- admin/change_form.html

hvad.descriptors

class hvad.descriptors.**NULL**

A pseudo type used internally to distinguish between None and no value given.

BaseDescriptor

class hvad.descriptors.**BaseDescriptor**
Base class for the descriptors, should not be used directly.

opts
The options (meta) of the model.

translation(*self*, *instance*)
Get the cached translation object on an instance, or get it from the database and cache it on the instance.

TranslatedAttribute

class hvad.descriptors.**TranslatedAttribute**
Standard descriptor for translated fields on the *Shared Model*.

name
The name of this attribute

opts
The options (meta) of the model.

__get__(*self*, *instance*, *instance_type=None*)
Gets the attribute from the translation object using `BaseDescriptor.translation()`. If no instance is given (used from the model instead of an instance) it returns the default value from the field.

__set__(*self*, *instance*, *value*)
Sets the value on the attribute on the translation object using `BaseDescriptor.translation()` if an instance is given, if no instance is given, raises an `AttributeError`.

__delete__(*self*, *instance*)
Deletes the attribute on the translation object using `BaseDescriptor.translation()` if an instance is given, if no instance is given, raises an `AttributeError`.

LanguageCodeAttribute

class hvad.descriptors.**LanguageCodeAttribute**
The language code descriptor is different than the other fields, since it's readonly. The getter is inherited from `TranslatedAttribute`.

__set__(*self*, *instance*, *value*)
Raises an attribute error.

__delete__(*self*, *instance*)
Raises an attribute error.

hvad.exceptions

exception hvad.exceptions.**WrongManager**
Raised when trying to access the related manager of a foreign key pointing from a normal model to a translated model using the standard manager instead of one returned by `hvad.utils.get_translation_aware_manager()`. Used to give developers an easier to understand exception than a `django.core.exceptions.FieldError`. This exception is raised by the `hvad.utils.SmartGetFieldByName` which gets patched onto the options (meta) of translated models.

hvad.fieldtranslator

hvad.fieldtranslator.**TRANSLATIONS**
Constant to identify *Shared Model* classes.

hvad.fieldtranslator.**TRANSLATED**
Constant to identify *Translations Model* classes.

hvad.fieldtranslator.**NORMAL**
Constant to identify normal models.

hvad.fieldtranslator.**MODEL_INFO**
Caches the model informations in a dictionary with the model class as keys and the return value of `_build_model_info()` as values.

hvad.fieldtranslator.**_build_model_info**(*model*)
Builds the model information dictionary for a model. The dictionary holds three keys: `'type'`, `'shared'` and `'translated'`. `'type'` is one of the constants `TRANSLATIONS`, `TRANSLATED` or `NORMAL`. `'shared'` and `'translated'` are a list of shared and translated fieldnames. This method is used by `get_model_info()`.

hvad.fieldtranslator.**get_model_info**(*model*)
Returns the model information either from the `MODEL_INFO` cache or by calling `_build_model_info()`.

hvad.fieldtranslator.**_get_model_from_field**(*starting_model*, *fieldname*)
Get the model the field `fieldname` on `starting_model` is pointing to. This function uses `get_field_by_name()` on the starting model's options (meta) to figure out what type of field it is and what the target model is.

hvad.fieldtranslator.**translate**(*querykey*, *starting_model*)
Translates a querykey (eg `'myfield__someotherfield__contains'`) to be language aware by spanning the translations relations wherever necessary. It also figures out what extra filters to the *Translations Model* tables are necessary. Returns the translated querykey and a list of language joins which should be used to further filter the queryset with the current language.

hvad.forms

TranslatableModelFormMetaclass

class hvad.forms.**TranslatableModelFormMetaclass**
Metaclass of `TranslatableModelForm`.

__new__(*cls*, *name*, *bases*, *attrs*)
The main thing happening in this metaclass is that the declared and base fields on the form are built by calling `django.forms.models.fields_for_model()` using the correct model depending on whether the field is translated or not. This metaclass also enforces the translations accessor and the master foreign key to be excluded.

TranslatableModelForm

class hvad.forms.**TranslatableModelForm**(*ModelForm*)

__metaclass__
`TranslatableMOdelFormMetaclass`

__init__(*self*, *data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=ErrorList*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
If this class is initialized with an instance, it updates `initial` to also contain the data from the *Translations Model* if it can be found.

save(*self*, *commit=True*)
Saves both the *Shared Model* and *Translations Model* and returns a combined model. The *Translations Model* is either altered if it already exists on the *Shared Model* for the current language (which is fetched from the `language_code` field on the form or the current active language) or newly created.

Note: Other than in a normal `django.forms.ModelForm`, this method creates two queries instead of one.

---

hvad.manager

This module is where most of the functionality is implemented.

hvad.manager.**FALLBACK_LANGUAGES**
The default sequence for fallback languages, populates itself from `settings.LANGUAGES`, could possibly become a setting on it's own at some point.

FieldTranslator

class hvad.manager.**FieldTranslator**
The cache mentioned in this class is the instance of the class itself, since it inherits dict.

Possibly this class is not feature complete since it does not care about multi-relation queries. It should probably use `hvad.fieldtranslator.translate()` after the first level if it hits the *Shared Model*.z

get(*self*, *key*)
Returns the translated fieldname for *key*. If it's already cached, return it from the cache, otherwise call `build()`

build(*self*, *key*)
Returns the key prefixed by `'master__'` if it's a shared field, otherwise returns the key unchanged.

ValuesMixin

class hvad.manager.**ValuesMixin**
A mixin class for `django.db.models.query.ValuesQuerySet` which implements the functionality needed by `TranslationQueryset.values()` and `TranslationQueryset.values_list()`.

_strip_master(*self*, *key*)
Strips `'master__'` from the key if the key starts with that string.

iterator(*self*)
Iterates over the rows from the superclass iterator and calls `_strip_master()` on the key if the row is a dictionary.

TranslationQueryset

class hvad.manager.**TranslationQueryset**
Any method on this queryset that returns a model instance or a queryset of model instances actually returns a

Translations Model which gets combined to behave like a *Shared Model*. While this manager is on the *Shared Model*, it is actually a manager for the *Translations Model* since the model gets switched when this queryset is instantiated from the `TranslationManager`.

override_classes
A dictionary of django classes to hvad classes to mixin when `_clone()` is called with an explicit *klass* argument.

_local_field_names
A list of field names on the *Shared Model*.

_field_translator
The cached field translator for this manager.

_real_manager
The real manager of the *Shared Model*.

_fallback_manager
The fallback manager of the *Shared Model*.

_language_code
The language code of this queryset.

translations_manager
The (real) manager of the *Translations Model*.

shared_model
The *Shared Model*.

field_translator
The field translator for this manager, sets `_field_translator` if it's `None`.

shared_local_field_names
Returns a list of field names on the *Shared Model*, sets `_local_field_names` if it's `None`.

_translate_args_kwargs(*self*, **args*, ***kwargs*)
Translates args (`django.db.models.expressions.Q` objects) and kwargs (dictionary of query lookups and values) to be language aware, by prefixing fields on the *Shared Model* with `'master__'`. Uses `field_translator` for the kwargs and `_recurse_q()` for the args. Returns a tuple of translated args and translated kwargs.

_translate_fieldnames(*self*, *fieldnames*)
Translate a list of fieldnames by prefixing fields on the *Shared Model* with `'master__'` using `field_translator`. Returns a list of translated fieldnames.

_recurse_q(*self*, *q*)
Recursively walks a `django.db.models.expressions.Q` object and translates it's query lookups to be prefixed by `'master__'` if they access a field on *Shared Model*.

Every `django.db.models.expressions.Q` object has an attribute `django.db.models.expressions.Q.children` which is either a list of other `django.db.models.expressions.Q` objects or a tuple where the key is the query lookup.

This method returns a new `django.db.models.expressions.Q` object.

_find_language_code(*self*, *q*)
Searches a `django.db.models.expressions.Q` object for language code lookups. If it finds a child `django.db.models.expressions.Q` object that defines a language code, it returns that language code if it's not `None`. Used in `get()` to ensure a language code is defined.

For more information about `django.db.models.expressions.Q` objects, see `_recurse_q()`.

Returns the language code if one was found or `None`.

_split_kwargs(*self*, ***kwargs*)
Splits keyword arguments into two dictionaries holding the shared and translated fields.

Returns a tuple of dictionaries of shared and translated fields.

_get_class(*self*, *klass*)
Given a `django.db.models.query.QuerySet` class or subclass, it checks if the class is a subclass of any class in `override_classes` and if so, returns a new class which mixes the initial class, the class from `override_classes` and `TranslationQueryset`. Otherwise returns the class given.

_get_shared_query_set(*self*)
Returns a clone of this queryset but for the shared model. Does so by using `_real_manager` and filtering over this queryset. Returns a queryset for the *Shared Model*.

language(*self*, *language_code=None*)
Specifies a language for this queryset. This sets the `_language_code` and filters by the language code.

If no language code is given, `django.utils.translations.get_language()` is called to get the current language.

Returns a queryset.

create(*self*, ***kwargs*)
Creates a new instance using the kwargs given. If `_language_code` is not set and language_code is not in kwargs, it uses `django.utils.translations.get_language()` to get the current language and injects that into kwargs.

This causes two queries as opposed to the one by the normal queryset.

Returns the newly created (combined) instance.

get(*self*, **args*, ***kwargs*)
Gets a single instance from this queryset using the args and kwargs given. The args and kwargs are translated using `_translate_args_kwargs()`.

If a language code is given in the kwargs, it calls `language()` using the language code provided. If none is given in kwargs, it uses `_find_language_code()` on the `django.db.models.expressions.Q` objects given in args. If no args were given or they don't contain a language code, it searches the `django.db.models.sql.where.WhereNode` objects on the current queryset for language codes. If none was found, it calls `language()` without an argument, which in turn uses `django.utils.translations.get_language()` to enforce a language to be used in this queryset.

Returns a (combined) instance if one can be found for the filters given, otherwise raises an appropriate exception depending on whether no or multiple objects were found.

get_or_create(*self*, ***kwargs*)
Will try to fetch the translated instance for the kwargs given.

If it can't find it, it will try to find a shared instance (using `_splitkwargs()`). If it finds a shared instance, it will create the translated instance. If it does not find a shared instance, it will create both.

Returns a tuple of a (combined) instance and a boolean flag which is `False` if it found the instance or `True` if it created **either** the translated or both instances.

filter(*self*, **args*, ***kwargs*)
Translates args and kwargs using `_translate_args_kwargs()` and calls the superclass using the new args and kwargs.

aggregate(*self*, **args*, ***kwargs*)
Loops through the passed aggregates and translates the fieldnames using `_translate_fieldnames()` and calls the superclass

latest(*self*, *field_name=None*)
Translates the fieldname (if given) using `field_translator` and calls the superclass.

in_bulk(*self*, *id_list*)
Not implemented yet.

delete(*self*)
Deletes the *Shared Model* using `_get_shared_query_set()`.

delete_translations(*self*)
Deletes the translations (and **only** the translations) by first breaking their relation to the *Shared Model* and then calling the delete method on the superclass. This uses two queries.

update(*self*, ***kwargs*)
Updates this queryset using kwargs. Calls `_split_kwargs()` to get two dictionaries holding only the shared or translated fields respectively. If translated fields are given, calls the superclass with the translated fields. If shared fields are given, uses `_get_shared_query_set()` to update the shared fields.

If both shared and translated fields are updated, two queries are executed, if only one of the two are given, one query is executed.

Returns the count of updated objects, which if both translated and shared fields are given is the sum of the two update calls.

values(*self*, **fields*)
Translates fields using `_translated_fieldnames()` and calls the superclass.

values_list(*self*, **fields*, ***kwargs*)
Translates fields using `_translate_fieldnames()` and calls the superclass.

dates(*self*, *field_name*, *kind*, *order='ASC'*)
Translates fields using `_translate_fieldnames()` and calls the superclass.

exclude(*self*, **args*, ***kwargs*)
Works like `filter()`.

complex_filter(*self*, *filter_obj*)
Not really implemented yet, but if filter_obj is an empty dictionary it just returns this queryset, since this is required to get admin to work.

annotate(*self*, **args*, ***kwargs*)
Not implemented yet.

order_by(*self*, **field_names*)
Translates fields using `_translated_fieldnames()` and calls the superclass.

reverse(*self*)
Not implemented yet.

defer(*self*, **fields*)
Not implemented yet.

only(*self*, **fields*)
Not implemented yet.

_clone(*self*, *klass=None*, *setup=False*, ***kwargs*)
Injects *_local_field_names*, *_field_translator*, *_language_code*, *_real_manager* and *_fallback_manager* into *kwargs*. If a *klass* is given, calls `_get_class()` to get a mixed class if necessary.

Calls the superclass with the new *kwargs* and *klass*.

iterator(*self*)
Iterates using the iterator from the superclass, if the objects yielded have a master, it yields a combined instance, otherwise the instance itself to enable non-cascading deletion.

Interestingly, implementing the combination here also works for `get()` and `__getitem__()`.

TranslationManager

class `hvad.manager.`**TranslationManager**
Manager to be used on `hvad.models.TranslatableModel`.

translations_model
The *Translations Model* for this manager.

language(*self*, *language_code=None*)
Calls `get_query_set()` to get a queryset and calls `TranslationQueryset.language()` on that queryset.

untranslated(*self*)
Returns an instance of `FallbackQueryset` for this manager.

get_query_set(*self*)
Returns an instance of `TranslationQueryset` for this manager. The queryset returned will have the *master* relation to the *Shared Model* marked to be selected when querying, using `select_related()`.

contribute_to_class(*self*, *model*, *name*)
Contributes this manager, the real manager and the fallback manager onto the class using `contribute_real_manager()` and `contribute_fallback_manager()`.

contribute_real_manager(*self*)
Creates a real manager and contributes it to the model after prefixing the name with an underscore.

contribute_fallback_manager(*self*)
Creates a fallback manager and contributes it to the model after prefixing the name with an underscore and suffixing it with `'_fallback'`.

FallbackQueryset

class `hvad.manager.`**FallbackQueryset**
A queryset that can optionally use fallbacks and by default only fetches the *Shared Model*.

_translation_fallbacks
List of fallbacks to use (or `None`).

iterator(*self*)
If `_translation_fallbacks` is set, it iterates using the superclass and tries to get the translation using the order of language codes defined in `_translation_fallbacks`. As soon as it finds a translation for an object, it yields a combined object using that translation. Otherwise yields an uncombined object. Due to the way this works, it can cause **a lot** of queries and this should be improved if possible.

If no fallbacks are given, it just iterates using the superclass.

use_fallbacks(*self*, **fallbacks*)
If this method gets called, `iterator()` will use the fallbacks defined here. If not fallbacks are given, `FALLBACK_LANGUAGES` will be used.

_clone(*self*, *klass=None*, *setup=False*, ***kwargs*)
Injects *translation_fallbacks* into *kwargs* and calls the superclass.

TranslationFallbackManager

class hvad.manager.**TranslationFallbackManager**

use_fallbacks(*self*, **fallbacks*)
Proxies to `FallbackQueryset.use_fallbacks()` by calling `get_query_set()` first.

get_query_set(*self*)
Returns an instance of `FallbackQueryset` for this manager.

TranslationAwareQueryset

class hvad.manager.**TranslationAwareQueryset**

_language_code
The language code of this queryset.

_translate_args_kwargs(*self*, **args*, ***kwargs*)
Calls `language()` using `_language_code` as an argument.

Translates *args* and *kwargs* into translation aware *args* and *kwargs* using `hvad.fieldtranslator.translate()` by iterating over the *kwargs* dictionary and translating it's keys and recursing over the `django.db.models.expressions.Q` objects in *args* using `_recurse_q()`.

Returns a triple of *newargs*, *newkwargs* and *extra_filters* where *newargs* and *newkwargs* are the translated versions of *args* and *kwargs* and *extra_filters* is a `django.db.models.expressions.Q` object to use to filter for the current language.

_recurse_q(*self*, *q*)
Recursively translate the keys in the `django.db.models.expressions.Q` object given using `hvad.fieldtranslator.translate()`. For more information about `django.db.models.expressions.Q`, see `TranslationQueryset._recurse_q()`.

Returns a tuple of *q* and *language_joins* where *q* is the translated `django.db.models.expressions.Q` object and *language_joins* is a list of extra language join filters to be applied using the current language.

_translate_fieldnames(*self*, *fields*)
Calls `language()` using `_language_code` as an argument.

Translates the fieldnames given using `hvad.fieldtranslator.translate()`

Returns a tuple of *newfields* and *extra_filters* where *newfields* is a list of translated fieldnames and *extra_filters* is a `django.db.models.expressions.Q` object to be used to filter for language joins.

language(*self*, *language_code=None*)
Sets the `_language_code` attribute either to the language given with *language_code* or by getting the current language from `django.utils.translations.get_language()`. Unlike `TranslationQueryset.language()`, this does not actually filter by the language yet as this happens in `_filter_extra()`.

get(*self*, **args*, ***kwargs*)
Gets a single object from this queryset by filtering by *args* and *kwargs*, which are first translated using `_translate_args_kwargs()`. Calls `_filter_extra()` with the *extra_filters* returned by `_translate_args_kwargs()` to get a queryset from the superclass and to call that queryset.

Returns an instance of the model of this queryset or raises an appropriate exception when none or multiple objects were found.

filter (*self*, **args*, ***kwargs*)
Filters the queryset by *args* and *kwargs* by translating them using `_translate_args_kwargs()` and calling `_filter_extra()` with the *extra_filters* returned by `_translate_args_kwargs()`.

aggregate (*self*, **args*, ***kwargs*)
Not implemented yet.

latest (*self*, *field_name=None*)
If a fieldname is given, uses `hvad.fieldtranslator.translate()` to translate that fieldname. Calls `_filter_extra()` with the *extra_filters* returned by `hvad.fieldtranslator.translate()` if it was used, otherwise with an empty `django.db.models.expressions.Q` object.

in_bulk (*self*, *id_list*)
Not implemented yet

values (*self*, **fields*)
Calls `_translated_fieldnames()` to translated the fields. Then calls `_filter_extra()` with the *extra_filters* returned by `_translated_fieldnames()`.

values_list (*self*, **fields*, ***kwargs*)
Calls `_translated_fieldnames()` to translated the fields. Then calls `_filter_extra()` with the *extra_filters* returned by `_translated_fieldnames()`.

dates (*self*, *field_name*, *kind*, *order='ASC'*)
Not implemented yet.

exclude (*self*, **args*, ***kwargs*)
Not implemented yet.

complex_filter (*self*, *filter_obj*)
Not really implemented yet, but if *filter_obj* is an empty dictionary it just returns this queryset, to make admin work.

annotate (*self*, **args*, ***kwargs*)
Not implemented yet.

order_by (*self*, **field_names*)
Calls `_translated_fieldnames()` to translated the fields. Then calls `_filter_extra()` with the *extra_filters* returned by `_translated_fieldnames()`.

reverse (*self*)
Not implemented yet.

defer (*self*, **fields*)
Not implemented yet.

only (*self*, **fields*)
Not implemented yet.

_clone (*self*, *klass=None*, *setup=False*, ***kwargs*)
Injects *_language_code* into *kwargs* and calls the superclass.

_filter_extra (*self*, *extra_filters*)
Filters this queryset by the `django.db.models.expressions.Q` object provided in *extra_filters* and returns a queryset from the superclass, so that the methods that call this method can directely access methods on the superclass to reduce boilerplate code.

TranslationAwareManager

class hvad.manager.**TranslationAwareManager**

get_query_set(*self*)
Returns an instance of `TranslationAwareQueryset`.

hvad.models

hvad.models.**create_translations_model**(*model*, *related_name*, *meta*, ***fields*)
A model factory used to create the *Translations Model*. Makes sure that the *unique_together* option on the options (meta) contain `('language_code', 'master')` as they always have to be unique together. Sets the `master` foreign key to *model* onto the *Translations Model* as well as the `language_code` field, which is a database indexed char field with a maximum of 15 characters.

Returns the new model.

TranslatedFields

class hvad.models.**TranslatedFields**
A wrapper for the translated fields which is set onto `TranslatableModel` subclasses to define what fields are translated.

Internally this is just used because Django calls the `contribute_to_class()` method on all attributes of a model, if such a method is available.

contribute_to_class(*self*, *cls*, *name*)
Calls `create_translations_model()`.

BaseTranslationModel

class hvad.models.**BaseTranslationModel**
A baseclass for the models created by `create_translations_model()` to distinguish *Translations Model* classes from other models. This model class is abstract.

TranslatableModelBase

class hvad.models.**TranslatableModelBase**
Metaclass of `TranslatableModel`.

__new__(*cls*, *name*, *bases*, *attrs*)

TranslatableModel

class hvad.models.**TranslatableModel**
A model which has translated fields on it. Must define one and exactly one attribute which is an instance of `TranslatedFields`. This model is abstract.

If initalized with data, it splits the shared and translated fields and prepopulates both the *Shared Model* and the *Translations Model*. If no *language_code* is given, `django.utils.translations.get_language()` is used to get the language for the *Translations Model* instance that gets initialized.

Note: When initializing a `TranslatableModel`, positional arguments are only supported for the shared fields.

---

objects
An instance of `hvad.manager.TranslationManager`.

_shared_field_names
A list of field on the *Shared Model*.

_translated_field_names
A list of field on the *Translations Model*.

classmethod contribute_translations(*cls*, *rel*)
Gets called from the `TranslatableModelBase` to set the descriptors of the fields on the *Translations Model* onto the model.

classmethod save_translations(*cls*, *instance*, ***kwargs*)
This classmethod is connected to the model's post save signal from the `TranslatableModelBase` and saves the cached translation if it's available.

translate(*self*, *language_code*)
Initializes a new instance of the *Translations Model* (does not check the database if one for the language given already exists) and sets it as cached translation. Used by end users to translate instances of a model.

safe_translation_getter(*self*, *name*, *default=None*)
Helper method to safely get a field from the *Translations Model*.

lazy_translation_getter(*self*, *name*, *default=None*)
Helper method to get the cached translation, and in the case the cache for some reason doesnt exist, it gets it from the database.

get_available_languages(*self*)
Returns a list of language codes in which this instance is available.

Extra information on _meta of Shared Models The options (meta) on `TranslatableModel` subclasses have a few extra attributes holding information about the translations.

translations_accessor The name of the attribute that holds the `TranslatedFields` instance.

translations_model The model class that holds the translations (*Translations Model*).

translations_cache The name of the cache attribute on this model.

Extra information on _meta of Translations Models The options (meta) on `BaseTranslationModel` subclasses have a few extra attributes holding information about the translations.

shared_model The model class that holds the shared fields (*Shared Model*).

hvad.utils

hvad.utils.**combine**(*trans*)
Combines a *Shared Model* with a *Translations Model* by taking the *Translations Model* and setting it onto the *Shared Model*'s translations cache.

hvad.utils.**get_cached_translation**(*instance*)
Returns the cached translation from an instance or None.

hvad.utils.**get_translation_aware_manager**(*model*)
Returns a manager for a normal model that is aware of translations and can filter over translated fields on translated models related to this normal model.

class hvad.utils.**SmartGetFieldByName**
Smart version of the standard get_field_by_name() on the options (meta) of Django models that raises a more useful exception when one tries to access translated fields with the wrong manager.

__init__(*self*, *real*)

__call__(*self*, *meta*, *name*)

hvad.utils.**permissive_field_by_name**(*self*, *name*)
Returns the field from the *Shared Model* or *Translations Model*, if it is on either.

hvad.compat.date

This module provides backwards compatiblity for Django 1.2 for the django.db.models.query.QuerySet.dates() API, which in Django 1.3 allows the fieldname to span relations.

DateQuerySet

class hvad.compat.date.**DateQuerySet**
Backport of django.db.models.query.DateQuerySet from Django 1.3.

DateQuery

class hvad.compat.date.**DateQuery**
Backport of django.db.models.sql.subqueries.DateQuery from Django 1.3.

4.11 Changelog

0.0.5 Started changelog

0.0.6 The behaviour of the fallbacks are now slightly changed - if you use .use_fallbacks() it will no longer return untranslated instances.

0.1 Fixed a bug where inlines would break in case the master didnt have the same id as the translation.

0.1.3 Introduces setting to configure the table name separator (unsurprisingly named NANI_TABLE_NAME_SEPARATOR). The default is _, to ensure schema compatibility with the deprecated django-multilingual-ng.

Note: Until version 0.1, no separator was used. If you want to upgrade to 0.1.1, you'll either have to rename the tables manually or set NANI_TABLE_NAME_SEPARATOR to '' in your settings.

0.1.4 Introduces `lazy_translation_getter()`

4.12 Glossary

Normal Model A Django model that does not have *Translated Fields*.

Shared Fields A field which is not translated, thus *shared* between the languages.

Shared Model The part of your model which holds the **untranslated** fields. Internally this is a separated model to your *Translations Model* as well as it's own database table.

Translated Fields A field which is translatable on a model.

Translated Model A Django model that subclasses `hvad.models.TranslatableModel`.

Translations Model The part of your model which holds the **translated** fields. Internally this is a (autogenerated) separate model with a ForeignKey to your *Shared Model*.

4.13 Indices and tables

- *genindex*
- *modindex*
- *search*
- *Glossary*

Python Module Index

h