Django Hvad Documentation

Release 1.8.0

Jonas Obrist
contributors

Aug 18, 2017

Contents

CHAPTER 1

About this project

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

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

CHAPTER 2

Notes on Django versions

The guideline is hvad supports all Django versions that are supported by the Django team. This holds true for Long-Term Support releases as well. Support for new versions will usually be introduced when they reach the beta stage.

Thus, django-hvad 1.5 is tested on the following configurations:

- Django 1.7.11, running Python 2.7, 3.3 or 3.4.
- Django 1.8.9, running Python 2.7, 3.3, 3.4 or 3.5.
- Django 1.9.2, running Python 2.7, 3.4 or 3.5.

All tests are run against MySQL and PostgreSQL.

CHAPTER 3

Contents

Installation

Requirements

- Django 1.8 or higher.
- Python 2.7 or PyPy 1.5 or higher, Python 3.4 or higher.

Installation

Packaged version

This is the recommended way. Install django-hvad using pip by running:

```
pip install django-hvad
```

This will download the latest version from pypi and install it.

Then add `'hvad'` to your `INSTALLED_APPS`, and proceed to *Quickstart*.

Latest development version

If you need the latest features from a yet unreleased version, or just like living on the edge, install django-hvad using pip by running:

```
pip install https://github.com/kristianoellegaard/django-hvad/tarball/master
```

This will download the development branch from github and install it.

Then add `'hvad'` to your `INSTALLED_APPS`, and proceed to *Quickstart*.

Quickstart

Define a multilingual model

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

Create a translated instance

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 instance:

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

This is the most straightforward way to create a new instance with translated fields. Doing it this way avoids the possibility of creating instances with no translation at all, something one usually wants to avoid.

Once we have an instance, we can add new translations. Let's add some French:

```
>>> hvad.translate('fr')
<DjangoApplication: django-hvad>
>>> hvad.name
'django-hvad'
>>> hvad.description
>>> hvad.description = 'Un projet pour gérer des modèles multilingues sous Django'
>>> hvad.description_author = 'Julien Hartmann'
>>> hvad.save()
```

Note: The *translate()* method creates a brand new translation in the specified language. Please note that it does not check the database, and that if the translation already exists, a database integrity exception will be raised when saving.

Querying translatable models

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'
```

We use *language()* to tell hvad we want to use translated fields, in English. This is one of the three ways to query a translatable model. It only ever considers instance that have a translation in the specified language and match the filters in that language.

Other ways are *untranslated()*, which uses a fallback algorithm to fetch the best translation within a list of languages, and direct, vanilla use of the queryset, which does not know about translations or translated fields at all.

Back to our instance, get it again, in other languages:

```
>>> obj = DjangoApplication.objects.language('fr').get(name='django-hvad')
>>> obj.description
u'Un projet pour gérer des modèles multilingues sous Django'
>>>
>>> DjangoApplication.objects.language('ja').filter(name='django-hvad')
[]
```

See how, in the second query, the fact that no translation exist in Japanese for our object had it filtered out of the query.

Note: We set an explicit language when calling *language()* because we are in an interactive shell, which is not necessarily in English. In your normal views, you can usually omit the language simply writing `MyModel.objects.language().get(...)`. This will use `get_language()` to get the language the environment is using at the time of the query.

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

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

Notice how the second query only considers French translations and returns an empty set.

Next, we will have a more detailed look at how to *work with translatable models*.

Models

Defining models

Defining models with django-hvad is done by inheriting *`TranslatableModel`*. Model definition works like in regular Django, with the following additional features:

- Translatable fields can be defined on the model, by wrapping them in a *`TranslatedFields`* instance, and assigning it to an attribute on the model. That attribute will be used to access the *translations* of your model directly. Behind the scenes, it will be a reversed ForeignKey from the *Translations Model* to your *Shared Model*.
- Translatable fields can be used in the model options. For options that take groupings of fields (`unique_together` and `index_together`), each grouping may have either translatable or non-translatable fields, but not both.
- Special field `language_code` is automatically created by hvad, and may be used for defining `unique_together` constraints that are only unique per language.

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(),
    )
    class Meta:
        unique_together = [('title', 'subtitle')]
```

Note: The `Meta` class of the model may not use the translatable fields in `order_with_respect_to`.

Note: TranslatedFields cannot contain a field named `master`, as this name is reserved by hvad to refer to the *Shared Model*. Also, special field `language_code` can be overriden in order to set it to be a different type of field, or change its options.

New and Changed Methods

translate

translate(*language_code*)
Prepares a new translation for this instance for the language specified.

Note: This method does not perform any database queries. It assumes the translation does not exist. If it does exist, trying to save the instance will raise an `IntegrityError`.

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 never performs 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', str(self.pk))
```

lazy_translation_getter

Changed in version 0.4.

lazy_translation_getter(*name*, *default=None*)
Tries to get the value of the field specified by `name` using *`safe_translation_getter()`*. If this fails, tries to load a translation from the database. If none exists, returns the value specified in `default`.

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

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, unless they were prefetched before (if the instance was retrieved with a call to `prefetch_related('translations')`).

---

save

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Overrides `save()`.

This method runs an extra query to save the translation cached on this instance, if any translation was cached.

It accepts both translated and untranslated fields in `update_fields`.

- If only untranslated fields are specified, the extra query will be skipped.
- If only translated fields are specified, the shared model update will be skipped. Note that this means signals will not be triggered.

Working with relations

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

Please note that `select_related()` used on a foreign key pointing from a *normal model* to a *translatable model* does not span to its *translations* and therefore accessing a translated field over the relation will cause an extra query. Foreign keys from a translatable model do not have this restriction.

If you wish to filter over a translated field over the relation from a *Normal Model* you have to use *`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.

Advanced Model Definitions

Abstract Models

New in version 0.5.

`Abstract models` can be used normally with hvad. Untranslatable fields of the base models will remain untranslatable, while translatable fields will be translatable on the concrete model as well:

```
class Place(TranslatableModel):
    coordinates = models.CharField(max_length=64)
    translations = TranslatedFields(
        name = models.CharField(max_length=255),
    )
    class Meta:
        abstract = True

class Restaurant(Place):
```

```
    score = models.PositiveIntegerField()
    translations = TranslatedFields()    # see note below
```

Note: The concrete models **must** have a *TranslatedFields* instance as one of their attributes. This is required because this attribute will be used to access the translations. It can be empty.

Proxy Models

New in version 0.4.

Proxy models can be used normally with hvad, with the following restrictions:

- The __init__ method of the proxy model will not be called when it is loaded from the database.
- As a result, the pre_init and post_init signals will not be sent for the proxy model either.

The __init__ method and signals for the concrete model will still be called.

Multi-table Inheritance

Unfortunately, multi-table inheritance is not supported, and unlikely to be. Please read #230 about the issues with multi-table inheritance.

Custom Managers and Querysets

Custom Manager

Vanilla managers, using vanilla querysets can be used with translatable models. However, they will not have access to translations or translatable fields. Also, such a vanilla manager cannot server as a default manager for the model. The default manager **must** be translation aware.

To have full access to translations and translatable fields, custom managers must inherit *TranslationManager* and custom querysets must inherit either *TranslationQueryset* (enabling the use of *language()*) or *FallbackQueryset* (enabling the use of *use_fallbacks()*). Both are described in the *dedicated section*.

Custom Querysets

Once you have a custom queryset, you can use it to override the default ones in your manager. This is where it is more complex than a regular manager: *TranslationManager* uses three types of queryset, that can be overriden independently:

- *queryset_class* must inherit *TranslationQueryset*, and will be used for all queries that call the *language()* method.
- *fallback_class* must inherit *FallbackQueryset*, and will be used for all queries that call the *untranslated()* method.
- *default_class* may be any kind of queryset (a TranslationQueryset, a FallbackQueryset or a plain QuerySet). It will be used for all queries that call neither language nor untranslated. It defaults to being a regular, translation-unaware QuerySet for compatibility, see next section about overriding it.

As a convenience, it is possible to override the queryset at manager instanciation, avoiding the need to subclass the manager:

```
class TVSeriesTranslationQueryset(TranslationQueryset):
    def is_public_domain(self):
        threshold = datetime.now() - timedelta(days=365*70)
        return self.filter(released__gt=threshold)

class TVSeries(TranslatableModel):
    # ... (see full definition in previous example)
    objects = TranslationManager(queryset_class=TVSeriesTranslationQueryset)
```

Overriding Default Queryset

New in version 0.6.

By default, the *TranslationManager* returns a vanilla, translation-unaware QuerySet when a query is done without either *language()* or *untranslated()*. This conservative behavior makes it compatible with third party modules. It is, however, possible to set it to be translation-aware by overriding it:

```
class MyModel(TranslatableModel):
    objects = TranslationManager(default_class=TranslationQueryset)
```

This deeply changes key behaviors of the manager, with many benefits:

- The call to language() can be omitted, filtering on translations is implied in all queries. It is still possible to use it to set another language on the queryset.
- As a consequence, all third-party modules will only see objects in current language, unless they are hvad-aware.
- They will also gain access to translated fields.
- Queries that use prefetch_related() will prefetch the translation as well (in current language).
- Acessing a translatable model from a ForeignKey or a GenericForeignKey will also load and cache the translation in current language.

In other terms, all queries become translation-aware by default.

Warning: Some third-party modules may break if they rely on the ability to see all objects. MPTT, for instance, will corrupt its tree if some objects have no translation in current language. Use caution when combining this feature with other manager-altering modules.

Custom Translation Models

New in version 1.5.

It is possible to have *translations* use a custom base class, by specifying a base_class argument to *TranslatedFields*. This may be useful for advanced manipulation of translations, such as customizing some model methods, for instance from_db():

```
class BookTranslation(models.Model):
    @classmethod
    def from_db(cls, db, fields, values):
        obj = super(BookTranslation, self).from_db(cls, db, field, values)
        obj.loaded_at = timezone.now()
        return obj

    class Meta:
```

```
        abstract = True

class Book(TranslatableModel):
    translations = TranslatedFields(
        base_class = BookTranslation,
        name = models.CharField(max_length=255),
    )
```

In this example, the `Book`'s translation model will have `BookTranslation` as its first base class, so every translation will have a `loaded_at` attribute when loaded from the database. Keep in mind this attribute will *not* be available on the book itself, but can be accessed through `get_cached_translation(book).loaded_at`.

Such classes are inserted into the translations inheritance tree, so if some other model inherits `Book`, its translations will also inherit `BookTranslation`.

Next, we will detail the *translation-aware querysets* provided by hvad.

Queryset API

If you do not need to do fancy things such as custom querysets and are not in the process of optimizing your queries yet, you can skip straight to next section, to start using your translatable models to *build some forms*.

The queryset API is at the heart of hvad. It provides the ability to filter on translatable fields and retrieve instances along with their translations. They come in two flavors:

- The *TranslationQueryset*, for working with instances translated in a specific language. It is the one used when calling *`TranslationManager.language()`*.
- The *FallbackQueryset*, for working with all instances regardless of their language, and eventually loading translations using a fallback algorithm. It is the one used when calling *`TranslationManager.untranslated()`*.

Note: It is possible to *override the querysets* used on a model's manager.

TranslationQueryset

The TranslationQueryset works on a translatable model, limiting itself to instances that have a translation in a specific language. Its API is almost identical to the regular Django `QuerySet`.

New and Changed Methods

language

`language`(*language_code=None*)

Sets the language for the queryset to either the given language code or the currently active language if None. Language resolution will be deferred until the query is evaluated.

This filters out all instances that are not translated in the given language, and makes translatable fields available on the query results.

The special value `'all'` disables language filtering. This means that objects will be returned once per language in which they match the query, with the appropriate translation loaded.

Note: support for *select_related()* in combination with `language('all')` is experimental. Please check the generated queries and open an issue if you have any problem. Feedback is appreciated as well.

---

fallbacks

fallbacks(**languages*)

New in version 0.6.

Enables fallbacks on the queryset. When the queryset has fallbacks enabled, it will try to use fallback languages if an object has not translation available in the language given to *language()*.

The *languages* arguments specified the languages to use, priorized from first to last. Special value *None* will be replaced with current language as returned by `get_language()`. If called with an empty argument list, the `LANGUAGES` setting will be used.

If an instance has no translation in the *language()*-specified language, nor in any of the languages given to `fallbacks()`, an arbitrary translation will be picked.

Passing the single value `None` alone will disable fallbacks.

Note: This feature requires Django 1.6 or newer.

---

delete_translations

delete_translations()

Deletes all *Translations Model* instances matched by a queryset, without deleting the *Shared Model* instances.

This can be used to target specific translations of specific objects for deletion. For instance:

```
# Delete English translation of all objects that have field == "foo"
MyModel.objects.language('en').filter(field='foo').delete_translations()

# Delete all translations but English for object with id 42
MyModel.objects.language('all').exclude(language_code='en').filter(pk=42).delete_
↪translations()
```

Warning: It is an error to delete all translations of an instance. This will cause the object to be unreachable through translation-aware queries and invisible in the admin panel.

If you delete all translations and re-create one immediately after, remember to enclose the whole process in a transaction to avoid the possibility of leaving the object unreachable.

select_related

select_related(**fields*)

Inherited from `select_related()`.

The `select_related` method also selects translations of translatable models when it encounters some.

Note: support for `select_related` in combination with `language('all')` is experimental. Please check the generated queries and open an issue if you have any problem. Feedback is appreciated as well.

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:

- *bulk_create()*
- *update_or_create()*
- *complex_filter()*
- *defer()*
- *only()*

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

Performance consideration

While most methods on *TranslationQueryset* 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:

- *create()*
- *update()* (only if both translated and untranslated fields are updated at once)

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.

FallbackQueryset

Deprecated since version 1.4.

This is a queryset returned by *untranslated()*, which can be used both to get the untranslated parts of models only or to use fallbacks for loading a translation based on a priority list of languages. By default, only the untranslated parts of models are retrieved from the database, and accessing translated field will trigger an additional query for each instance.

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

Warning: If you have a default `ordering` defined on your model and it includes any translated field, you must specify an ordering on every query so as not to use the translated fields specified by the default ordering.

New Methods

use_fallbacks

Changed in version 0.5.

use_fallbacks (**fallbacks*)
Deprecated since version 1.4.

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, replacing the special *None* value with the current language at query evaluation, as returned by `get_language()`. Otherwise the order of your LANGUAGES setting will be used, prepended with current language.

This method is now deprecated, and one should use *TranslationQueryset.fallbacks()* for an equivalent feature.

> **Warning:** Using fallbacks with a version of Django older than 1.6 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 possible.
>
> Changed in version 0.5: Fallbacks were reworked, so that when running on Django 1.6 or newer, only one query is needed.

Not implemented public queryset methods

The following are methods on a queryset which are public APIs in Django, but are not implemented on fallback querysets.

- `aggregate()`
- `annotate()`
- `defer()`
- `only()`

Next, we will use our models and queries to *build some forms*.

Forms

Although Django's `ModelForm` can work with translatable models, they will only know about untranslatable fields. Don't worry though, django-hvad's got you covered with the following form types:

- *TranslatableModelForm* is the translation-enabled counterpart to Django's `ModelForm`.
- *Translatable formsets* is the translation-enabled counterpart to Django's `model formsets`, for editing several instances at once.
- *Translatable inline formsets* is the translation-enabled counterpart to Django's `inline formsets`, for editing several instances attached to another object.
- *Translation formsets* allows building a formset of all the translations of a single instance for editing them all at once. For instance, in a tabbed view.

TranslatableModelForm

TranslatableModelForms work like `ModelForm`, but can display and edit translatable fields as well. Their use is very similar, except the form must subclass *`TranslatableModelForm`* instead of `ModelForm`:

```
class ArticleForm(TranslatableModelForm):
    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter']
```

Notice the difference from `Django's example`? There is none but for the parent class. This `ArticleForm` will allow editing of one `Article` in one language, correctly introspecting the model to know which fields are translatable.

The form can work in either normal mode, or **enforce** mode. This affects the way the form chooses a language for displaying and committing.

- A form is in normal mode if it has no language set. This is the default. In this mode, it will use the language of the `instance` it is given, defaulting to current language if not `instance` is specified.
- A form is in **enforce** mode if is has a language set. This is usually achieved by calling *translatable_modelform_factory*. When in **enforce** mode, the form will always use its language, disregarding current language and reloading the `instance` it is given if it has another language loaded.
- The language can be overriden manually by providing a `custom clean() method`.

In all cases, the language is not part of the form seen by the browser or sent in the POST request. If you need to change the language based on some user input, you must override the `clean()` method with your own logic, and set `cleaned_data['language_code']` with it.

All features of Django forms work as usual.

TranslatableModelForm factory

Similar to Django's `ModelForm factory`, hvad eases the generation of uncustomized forms by providing a factory:

```
BookForm = translatable_modelform_factory('en', Book, fields=('author', 'title'))
```

The translation-aware version works exactly the same way as the original one, except it takes the language the form should use as an additional argument.

The returned form class is in **enforce** mode.

Note: If using the `form=` parameter, the given form class must inherit *TranslatableModelForm*.

TranslatableModel Formset

Similar to Django's `ModelFormset factory`, hvad provides a factory to create formsets of translatable models:

```
AuthorFormSet = translatable_modelformset_factory('en', Author)
```

This formset allows edition a collection of `Author` instances, all of them being in English.

All arguments supported by Django's `modelformset_factory()` can be used.

For instance, it is possible to override the queryset, the same way it is done for a regular formset. In fact, it is recommended for performance, as the default queryset will not prefetch translations:

```
BookForm = translatable_modelformset_factory(
    'en', Book, fields=('author', 'title'),
    queryset=Book.objects.language('en').all(),
)
```

Here, using *language()* ensures translations will be loaded at once, and allows filtering on translated fields is needed.

The returned formset class is in **enforce** mode.

Note: To override the form by passing a `form=` argument to the factory, the custom form must inherit *TranslatableModelForm*.

TranslatableModel Inline Formset

Similar to Django's `inline formset factory`, hvad provides a factory to create inline formsets of translatable models:

```
BookFormSet = translatable_inlineformset_factory('en', Author, Book)
```

This creates an inline formset, allowing edition of a collection of instances of `Book` attached to a single instance of `Author`, all of those objects being editted in English. It does not allow editting other languages; for this, please see *translationformset_factory*.

Any argument accepted by Django's `inlineformset_factory()` can be used with `translatable_inlineformset_factory` as well.

The returned formset class is in **enforce** mode.

Note: To override the form by passing a `form=` argument to the factory, the custom form must inherit *TranslatableModelForm*.

Translations Formset

Basic usage

The translation formset allows one to edit all translations of an instance at once: adding new translations, updating and deleting existing ones. It works mostly like regular `BaseInlineFormSet` except it automatically sets itself up for working with the *Translations Model* of given *TranslatableModel*.

Example:

```
from django.forms.models import modelform_factory
from hvad.forms import translationformset_factory
from myapp.models import MyTranslatableModel

MyUntranslatableFieldsForm = modelform_factory(MyTranslatableModel)
MyTranslationsFormSet = translationformset_factory(MyTranslatableModel)
```

Now, `MyUntranslatableFieldsForm` is a regular, Django, translation-unaware form class, showing only the untranslatable fields of an instance, while `MyTranslationsFormSet` is a formset class showing only the translatable fields of an instance, with one form for each available translation (plus any additional forms requested with the `extra` parameter - see `modelform_factory()`).

Custom Translation Form

As with regular formsets, one may specify a custom form class to use. For instance:

```
class MyTranslationForm(ModelForm):
    class Meta:
        fields = ['title', 'content', 'slug']

MyTranslationFormSet = translationformset_factory(
    MyTranslatableModel, form=MyTranslationForm, extra=1
)
```

Note: The translations formset will use a `language_code` field if defined, or create one automatically if none was defined.

One may also specify a custom formset class to use. It must inherit *`BaseTranslationFormSet`*.

Wrapping it up: editing the whole instance

A common requirement, being able to edit the whole instance at once, can be achieved by combining a regular, translation unaware `ModelForm` with a translation formset in the same view. It works the way one would expect it to. The following code samples highlight a few gotchas.

Creating the form and formset for the object:

```
FormClass = modelform_factory(MyTranslatableModel)
TranslationsFormSetClass = translationformset_factory(MyTranslatablemodel)

self.object = self.get_object()
form = FormClass(instance=self.object, data=request.POST)
formset = TranslationsFormSetClass(instance=self.object, data=request.POST)
```

Checking submitted form validity:

```
if form.is_valid() and formset.is_valid():
    form.save(commit=False)
    formset.save()
    self.object.save_m2m()  # only if our model has m2m relationships
    return HttpResponseRedirect('/confirm_edit_success.html')
```

Note: When saving the formset, translations will be recombined with the main object, and saved as a whole. This allows custom `save()` defined on the model to be called properly and signal handlers to be fed a full instance. For this reason, we use `commit=False` while saving the form, avoiding a useless query.

Warning: You must ensure that `form.instance` and `formset.instance` reference the same object, so that saving the formset does not overwrite the values computed by **form**.

A common way to use this view would be to render the `form` on top, with the `formset` below it, using JavaScript to show each translation in a tab.

Next, we will take a look at the *administration panel*.

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`.

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. Entries are linked to their corresponding admin page.

Note: You should add *prefetch_related('translations')* to your queryset if you use this in `list_display`, else one query will be run for every item in the list.

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 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.

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*.

ModelAdmin APIs not available on TranslatableAdmin

A list of public APIs on `django.contrib.admin.ModelAdmin` which are not implemented on *`hvad.admin.TranslatableAdmin`* for handling translatable fields, these APIs should continue to work as usual for non-translatable fields.

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

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

REST Framework

New in version 1.2.

What would be a modern application without dynamic components? Well, it would not be so modern to begin with. This is why django-hvad provides fully tested and integrated support for Django REST framework.

The philosophy is the same one that is used for Django's *forms*, hvad providing the following extensions:

- *TranslatableModelSerializer* is the translation-enabled counterpart to ModelSerializer.
- *HyperlinkedTranslatableModelSerializer* is the translation-enabled counterpart to HyperlinkedModelSerializer.
- *TranslationsMixin* can be plugged into a *ModelSerializer* to add a dictionary of all available translations. Writing is supported as well.

Note: Support for REST framework requires Django REST Framework version 3.1 or newer.

TranslatableModelSerializer

`hvad.contrib.restframework.TranslatableModelSerializer`

TranslatableModelSerializer works like ModelSerializer, but can serialize and deserialize translatable fields as well. Their use is very similar, except the serializer must subclass `hvad.contrib.restframework.TranslatableModelSerializer`:

```
class BookSerializer(TranslatableModelSerializer):
    class Meta:
        model = Book
        fields = ['title', 'author', 'review']
```

Notice the difference from a regular serializer? There is none. This `BookSerializer` will allow serializing and deserializing one `Book` in one language, correctly introspecting the model to know which fields are translatable.

It is also possible to include the language on the serializer. This is done by default (if no `fields` is specified), or you may include `'language_code'` as part of the field list.

Like *TranslatableModelForm*, `TranslatableModelSerializer` can work in either normal mode, or **enforce** mode. The semantics of both mode are exactly the same as with forms, selecting the way a language is chosen for serializing and deserializing.

- A serializer is in normal mode if it has no language set. This is the default. In this mode, it will use the language of the `instance` it is given, defaulting to current language if no `instance` is specified.
- A serializer is in **enforce** mode if is has a language set. This is achieved by giving it a `language=` argument at instanciation. When in **enforce** mode, the serializer will always use its own language, disregarding current language and reloading the `instance` it is given if it has another language loaded.
- The language can be overriden manually by providing a custom `validate()` method. This method should set the desired language in `data['language_code']`. Please refer to REST framework documentation for details on the `validate()` method.

When the serializer is in normal mode, it is possible to send `'language_code'` as part of the serialized representation. More on this below. In **enforce** mode however, including a language code in a POST, PATCH or PUT request is an error that will raise a `ValidationError` as appropriate.

All features of regular REST framework serializers work as usual.

Examples

Adding the language to the serialized data, in **normal** mode:

```
class BookSerializer(TranslatableModelSerializer):
    class Meta:
        model = Book
        fields = ['title', 'author', 'language_code']

# Now language appears in serialized representation
serializer = BookSerializer(instance=Book.objects.language('ja').get(pk=1))
# => {"title": "", "author": "12", "language_code": "ja" }

# It can also be set explicitly in POST/PUT/PATCH data
print(data['language_code']) # 'fr'
serializer = BookSerializer(data=data)
if serializer.is_valid():
    obj = serializer.save()
    assert obj.language_code == 'fr'
```

Setting a serializer in **enforce** mode:

```
# In enforce mode, serialized data will always use the enforced language
serializer = BookSerializer(instance=Book.objects.untranslated().get(pk=1), language=
↪'en')
assert serializer.data['language_code'] == 'en'

# In enforce mode, language is implicit
assert 'language_code' not in request.data
serializer = BookSerializer(data=request.data, language='fr')
if serializer.is_valid():
    obj = serializer.save()
    assert obj.language_code == 'fr'

# In enforce mode, language must not be provided in data
assert 'language_code' in request.data
serializer = BookSerializer(data=request.data, language='fr')
assert not serializer.is_valid()
```

Manually overriding deserialized language:

```
class UserBookSerializer(TranslatableModelSerializer):
    def validate(self, data):
        # assuming you made a custom User model that has an associated
        # preferences object including the user's preferred language
        data = super(UserBookSerializer, self).validate(data)
        data['language_code'] = self.context['request'].user.preferences.language
        return data

    class Meta:
        model = Book
```

HyperlinkedTranslatableModelSerializer

`hvad.contrib.restframework.HyperlinkedTranslatableModelSerializer`

The `HyperlinkedTranslatableModelSerializer` is equivalent to `TranslatableModelSerializer`,

except it outputs hyperlinks instead of ids. There is not much to add here, everything that applies to *TranslatableModelSerializer* also applies to `HyperlinkedTranslatableModelSerializer`, except it uses REST framework's HyperlinkedModelSerializer semantics.

TranslationsMixin

`hvad.contrib.restframework.TranslationsMixin`

This mixin is another approach to handling translations for your REST api. With *TranslatableModelSerializer*, a relevant language is made visible, which is perfect for translation-unaware client-side applications. `TranslationsMixin` takes the other approach: it exposes all translations at once, letting the client-side application choose or handle translations the way it wants. This is most useful for admin-type applications.

Use is very simple: mix it into a regular serializer:

```
from rest_framework.serializers import ModelSerializer

class BookSerializer(TranslationsMixin, ModelSerializer):
    class Meta:
        model = Book

obj = Book.objects.untranslated().prefetch_related('translations').get(pk=1)
serializer = BookSerializer(instance=obj)
pprint(serializer.data)
# {'author': '1',
#  'id': 1,
#  'translations': {'en': {'title': 'The Little Prince'},
#                   'fr': {'title': 'Le Petit Prince'}}}
```

Note: For performance, you should always prefetch the translations like in the above example, otherwise the serializer will have to fetch them for each object independently, resulting in a large number of queries.

Writing is supported as well. It takes a dictionary of translations, the very same format it outputs. Existing translations will be updated, missing translations will be created. Any existing translation that is not in the data will be deleted.

For convenience, you can include both the translations dictionary and translated fields in the same serializer. This can be handy if only some parts of your application care about all the translations. For instance, a book listing might just want the title in the preferred language, while the book editing dialog allows editing all languages. In this case, direct translated fields will be read-only, use the translations dictionary for updating.

It is possible to override the representation of translations. This is done by specifying a custom serializer on the meta:

```
from rest_framework import serializers

class BookTranslationSerializer(serializers.ModelSerializer):
    class Meta:
        exclude = ['subtitle', 'cover']

class BookSerializer(TranslationsMixin, serializers.ModelSerializer):
    class Meta:
        model = Book
        translations_serializer = BookTranslationSerializer
```

In case advanced customisation of translations is required, be aware that your custom translation serializer is handed the full object. This allows building computed fields using both translated and untranslated data.

However, it can interfer with some field types, most notable related fields, which expect the actual translation model. Hvad handles this automatically in its default translation serializer. You can inherit this handling by making your own translation serializer a subclass of `hvad.contrib.restframework.NestedTranslationSerializer`.

Frequent Questions

Why "django-*hvad*"?

The project first started as "django-*nani*", created by Jonas Obrist. The word *nani* is the romanized form of "", which means *What?*.

When Kristian Øllegaard took responsibility for updating and maintaining the project, including a major refactor of the internals, the project was renamed to *hvad*, which is the Danish word for *What?*.

If we were to continue the trend, we would rename to django-*quoi*, but that is very unlikely unless Django introduces major breaking changes in a future version.

How do I get the right language from the request?

In most cases, you will be using *language()* with no arguments in your views and forms. When used with no arguments, it defaults to using the current language, as returned by Django's `get_language()`.

Therefore, having hvad use the right language is mostly a matter of having Django setting it right. Fortunately, Django provide the tools to do this, in the form of the `LocaleMiddleware`. Here is a short guide to making it work.

First, the middleware must be enabled. This is done by adding `'django.middleware.locale.LocaleMiddleware'` to `MIDDLEWARE_CLASSES` in you settings file.

- It must come after `SessionMiddleware`.
- If you use the `CacheMiddleware`, then the `LocaleMiddleware` must come after that too.
- Right after those, as close to the top as possible, should the `LocaleMiddleware` come:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

Now, the middleware will try to determine the user's language preference. There is a detailed explanation of how it proceeds in Django documentation.

Hvad will happily follow the language discovered by the middleware. Although this will usually be enough, you may sometimes want to force the language. Either on a specific request by explicitly passing a language code to *language()*, or by changing the current language. The later is done through activate().

How about multilingual URI?

We will assume the URI we want to be multilingual are made of two kind of components: static components, and dynamic components. We want to translate both kind:

- Static components, through ugettext_lazy().
- Dynamic components, from our translatable models.

Static components

This is thoroughly documented in Django's URL i18n documentation and does not actually involve hvad, so this will be a short guide. It requires the LocaleMiddleware to be properly *configured*, so please do that first.

With this middleware active, each request will set a current language before looking up the URI in your urlconf.py. This makes it possible to use ugettext_lazy() in your patterns, like this:

```
from django.conf.urls import url
from django.utils.translation import ugettext_lazy as _

urlpatterns = [
    url(_(r'^en/news/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>.*)'),
        views.NewsView, name='news-detail'),
]
```

The pattern would then appear in the list of translatable string, making it possible to add, for instance, a translation that would read ^fr/actualites/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>.*)

Note: Notice the language code at the beginning. Although not required, prefixing your URI with it makes the life much easier to the LocaleMiddleware.

Dynamic components

We translated the static parts of the URI with Django mechanics. What now? Well, if we touch nothing, everything will work fine: the language of the user will be used for URI resolution, and then hvad's *language()* will follow the same. Database queries will filter on the user's language by default, and your view will 404 if nothing is found in that language.

Now, in some instances, the language might not be known. Because your URI does not include a language code, or because you want to find objects regardless of the user's language. Maybe based on a translatable slug. This can be done by querying with language('all'):

```
from django.views.generic.base import TemplateView

class NewsView(TemplateView):
    def get(self, request, *args, **kwargs):
        slug = kwargs['slug']
```

```
        obj = News.objects.language('all').get(published=True, slug=slug)

        context = self.get_context_data(news=obj, language=obj.language_code)
        return self.render_to_response(context)
```

This view will find the news given its slug, regardless of which language it is in. It will display it in the language it is found with. It would be possible to force it to be in the user's preferred language by adding another query:

```
obj = News.objects.language('all').get(published=True, slug=slug)
try:
    # Try to replace obj with a version in current user's language
    obj = News.objects.language().get(pk=obj.pk)
except News.DoesNotExist:
    # No translation for user's language, stick with that of the slug
    pass
```

Note: Note those examples assume slugs are unique amongst all news of all languages.

How do I use hvad with MPTT?

Note: Since version 0.5, hvad no longer uses a custom metaclass, making the old metaclass workaround unneeded.

The mptt application implements Modified Preorder Tree Traversal for Django models. If you have any model in your project that is organized in a hierarchy of items, you should be using it.

MPTT and hvad can cooperate pretty well by merging the `TranslationManager` from hvad with the `MPTTManager` from MPTT. Doing so is relatively straightforward:

```
class FolderManager(TranslationManager, MPTTManager):
    use_for_related_fields = True

class Folder(MPTTModel, TranslatableModel):
    # ...
    objects = FolderManager()
```

The same principle would work with a custom queryset too, but MPTT does not define one.

How do I separate translatable fields in admin?

This comes froms #68.

We need to separate the fields in fieldsets. Unfortunately, technical restrictions on Django < 1.6 make support for translated fields directly on ModelAdmin difficult. Therefore, it must be worked around by defining a custom `get_fieldsets()` as such:

```
class MyModelAdmin(TranslatableAdmin):
    # ... other admin stuff
    def get_fieldsets(self, request, obj=None):
        return (
            (_('Common fields'), {
                'fields': ('owner', 'is_published',),
```

```
            }),
            (_('Translated fields'), {
                'fields': ('name', 'slug', 'description',),
            }),
        )
```

The model admin will then be generated with two fieldsets, one for common fields and one for translated fields. At this point though, language tabs still appear at the top, with both fieldsets beneath. This can be changed by providing a custom template for rendering the form. This is a 2-step process. First, we specify a custom template on the admin:

```
class MyModelAdmin(TranslatableAdmin):
    # ... ohter admin stuff
    change_form_template = 'myapp/change_form.html'
```

Then we create the template, by extending the base admin change form. Only, we place the language tabs where we want them to be:

```
{% extends "admin/change_form.html" %}

{% block field_sets %}
    {% for fieldset in adminform %}
        {% include "admin/includes/fieldset.html" %}
        {% if forloop.first %}
            {% include "admin/hvad/includes/translation_tabs.html" %}
        {% endif %}
    {% endfor %}
{% endblock %}
```

In that example, the language tabs will end up in between the first and second fieldsets. We are mostly done, all we miss is some CSS rules to have the tabs look right. We may simply copy-paste the `extrahead` block straight from `hvad/templates/admin/hvad/change_form.html`.

Note: Remember that language tabs are links to other pages. This means that clicking them without saving the form will not save anything, not even common fields. Basically, a new, fresh form will be built from DB values. If adding new object, common fields will be blanked as well.

Release Notes

1.8.0 - current release

Released on April 28, 2017

Python and Django versions supported:

- Support for Django 1.10 was added.
- Django 1.7 is no longer supported.
- So, as a reminder, supported Django versions for this release are: 1.8 LTS, 1.9, 1.10.x (for x 1) and 1.11.

New features:

- Automatic loading of translations on attribute access can now be disabled, by setting `HVAD["AUTOLOAD_TRANSLATIONS"]` to `False`. This will prevent hvad from initiating database queries. Accessing translatable attributes with no translation loaded will then raise an `AttributeError`.

- It is possible to automatically install *TranslationQueryset* as the default queryset for all translatable models, by setting HVAD["USE_DEFAULT_QUERYSET"] to True. Specifically, it changes *default_class* to be a *TranslationQueryset* instead of a QuerySet. See the section about *overriding the default queryset* for advantages and caveats of doing so.
- Field declaration for internal language_code attribute can be overriden. — #332.

Compatibility warnings:

- All settings have been moved to a unique HVAD dictionary. Please update your django settings accordingly.
- Deprecated class FallbackQueryset has been removed. Using it along with *FallbackQueryset.use_fallbacks()* did not work on Django 1.9 and newer and was deprecated on older versions. Using it without that method made it behave like a regular queryset. So as a summary,
  - Code using *.untranslated().use_fallbacks()* must be replaced with *.language().fallbacks()*.
  - All other uses of *FallbackQueryset* can be safely replaced with a regular QuerySet.
- Translated admin no longer shows objects lacking a translation. This was already the case on Django 1.9 and newer, and this behavior now extends to all Django versions. Such objects should not happen anyway, and throw a warning when encountered.

Fixes:

- Increase speed of translated attribute access by ~30%, by avoiding a method call when a translation is loaded.
- Attempting to use a reserved name for a translated field now raises an ImproperlyConfigured exception instead of silently ignoring the field.
- Instances created by serializers using TranslatableModelMixin in normal, non-*enforcing* mode can no longer be created without a translation. — #322.

1.7.0

Released on February 8, 2017

New features:

- Support for defer() and only() was added. Note that deferring all translated fields will **not** result in translation being skipped, because 1) it needs to be inspected for language resolution and 2) loaded translation language must be fixed at query time. Support is limited to immediate fields at the moment, ie it is not possible to defer fields of additionals models loaded through select_related().

Compatibility warnings:

- Internal admin method TranslatableAdmin.get_available_languages() is deprecated and will be removed. Use *TranslatableModel.get_available_languages()* instead.
- Internal admin method TranslatableAdmin.get_language_tabs() signature changed.

Fixes:

- Do not consider annotations when looking up translatable query fields. Fixes errors that could arise when using some annotation names. — #303.
- Accept special value __all__ for form field list, as a synonym for None, meaning include all known fields. — #313.
- Fix translation deletion links that were improperly generated when using inline change forms. — #317.

1.6.0

Released on September 6, 2016

Python and Django versions supported:

- Support for Django 1.10 was added. It requires version 1.10.1 or better.
- So, as a reminder, supported Django versions for this release are: 1.7, 1.8 LTS, 1.9, 1.10.x (for x 1).

Fixes:

- No longer set `master` to `NULL` before clearing translations when using `delete_translations()`. This only triggers one query instead of two, and allows enforcing non-null foreign key at the database level.
- Django system checks are now run in the test suite in addition to hvad's tests.

1.5.1

Released on May 23, 2016

Fixes:

- Filter out m2m and generic fields in `update_translation()` so it does not bite when using (unsupported) m2m fields or generic relations in a translation — #285.

1.5.0

Released on February 2, 2016

Python and Django versions supported:

- Django 1.4 LTS is no longer supported.
- So, as a reminder, supported Django versions for this release are: 1.7, 1.8 LTS, 1.9.

New features:

- It is now possible to specify a *custom translation base* model, allowing advanced translation manipulation, such as controlling their loading with `from_db()`.
- Translated model's `save()` method now accepts translated field names in `update_fields`. Also, if only translated fields, or only untranslated fields are specified in `update_fields`, the extra query will be skipped.
- Support for third parameter on `ModelAdmin`'s `get_object()` method was added.
- Experimental support for using *language('all')* together with `select_related()` is being introduced. Please check the generated queries if you use it. Feedback is appreciated.

Compatibility Warnings:

- Saving of translations now happens in the model's `save()` method. It used to happen in the `post_save` signal.
- `TranslationsMixin` now splits the update into `update` and `update_translation` methods. The former is called once per save, and uses the latter as many times as required to update all translations.

Fixes:

- Translation deletion URIs are no longer broken on Django 1.9 — #279.
- REST framework translation support now uses `update_fields` to reduce the number of queries when updating an object.

- REST framework translation support no longer breaks when using `PrimaryKeyRelatedField` and `TranslationsMixin` together — #278.
- Admin no longer uses deprecated `patterns` function — #268.

1.4.0

Released on November 10, 2015

Python and Django versions supported:

- Support for Python 3.5 was added.
- Support for Django 1.9 was added.
- Django 1.6 is no longer officially supported.
- Django 1.4 LTS has reached its end of life, and support will be dropped in hvad 1.5.
- So, as a reminder, supported versions for this release are: 1.4 LTS, 1.7, 1.8 LTS, 1.9.

Compatibility Warnings:

- As a result of the annotations fix (see below), applications that worked around `annotate()` 's shortcomings on translation querysets are likely to break, as `annotate()` has been fixed. The workarounds should be simply removed.
- Method *`FallbackQueryset.use_fallbacks()`* is **not** supported on Django 1.9 and newer (and deprecated on other versions, see below). Please use *TranslationQueryset.fallbacks()* instead.
- Translated admin no longer shows objects lacking a translation, starting from Django 1.9. This behavior will be extended to all Django versions in the next release. Such objects should not happen anyway, and throw a warning when encountered.
- Translation model building has been refactored. It is functionally equivalent to its previous implementation (it passes the exact same test suite), but code depending on the internals and inner implementation details could break.

Deprecation List:

- Method *`FallbackQueryset.use_fallbacks()`* is now deprecated on Django 1.6 and newer. The plan is to completely drop *`FallbackQueryset`* in the near future, and let *`TranslationManager.untranslated()`* default to returning a plain Django queryset, thus enabling `MyModel.objects.untranslated()` to give access to all features a plain Django queryset supports.

  For queries that need fallbacks, the `use_fallbacks()` method has long been superseded by *TranslationQueryset.fallbacks()*, which is better tested, uses simpler code yet supports more features. Please update your queries accordingly.

  `MyModel.objects.untranslated().use_fallbacks('en', 'ja', 'fr')` should be rewritten as `MyModel.objects.language('en').fallbacks('ja', 'fr')`, or even `MyModel.objects.language().fallbacks()` to have the query use your application's language settings automatically.

Fixes:

- Annotations added to a *`TranslationQueryset`* using the `annotate()` method no longer end up on the translation cache with a `master__` prefix.
- Specifying translation fields in `unique_together` on translatable models no longer causes Django to generate incorrect migrations. — #260.

- When no `Meta` options are set on a *TranslatableModelForm*, the auto-created one now correctly inherits that of its first base class that has one set — #262.
- Using `language('all')` together with `values()` no longer breaks — #264.

1.3.0

Released on July 29, 2015

This release is a collection of fixes and improvements, some of which may introduce minor compatibility issues. Please make sure you fix any deprecation warnings before upgrading to avoid those issues.

Python and Django versions supported:

- Django 1.5 is no longer officially supported.
- Django 1.6 has reached its end of life, and support will be dropped in hvad 1.4.
- As a reminder, Django 1.4 is still supported, so supported versions for this release are: 1.4, 1.6, 1.7, 1.8.

New Features:

- Russian and Latvian translations are now included, thanks to Juris Malinens — #248.

Compatibility Warnings: deprecated features pending removal in 1.3 have been removed. Most notably:

- Calling `save()` on an invalid form now raises an assertion exception.
- Classes `TranslatableModelBase`, `TranslationFallbackManager`, `TranslatableBaseView` and method `TranslationManager.using_translations()` no longer exist.
- Deprecated view methods and context modifiers now raise an assertion exception.

Fixes:

- Lift Django restrictions on translated fields in `Meta.unique_together` and `Meta.index_together` — #252.
- Properly forward model validation methods to translation validation methods, so that model validation detects constraint violations on the translation as well. Fixes duplicate detection in admin for unique constraints on translations — #251.
- Detect name clash between translated and non-translated fields — #240.
- Validate that at least one translation is provided when deserializing objects in `TranslationsMixin` — #256.
- Fix handling of model edition from an admin popup in Django 1.7 and newer — #253.
- Generate proper ORM structures for fallbacks. Avoids table relabeling breaking queries, for instance when using `update()` or feeding a queryset to another queryset — #250.

1.2.2

Released on June 3, 2015

Fixes:

- Properly handle `language_code` in `Meta.unique_together` and `Meta.index_together` — #244.

1.2.1

Released on April 29, 2015

Fixes:

- Make passing the `model` argument to queryset's `__init__` optional. Still allow it to be passed either as a positional or named argument — #241.

1.2.0

Released on March 19, 2015

This is a feature release, to push REST framework support onto the main package.

Python and Django versions supported:

- Due to this version being released early, end of support for Django 1.5 has been postponed until next release.

New features:

- Support for Django REST framework is now included. It requires REST framework version 3.1 or newer — #220.

1.1.1

Released on March 5, 2015

Fixes:

- Backwards compatibility issue in get_field implementation — #233.
- Admin no longer breaks on models using another `pk` field than `id` — #231.

1.1.0

Released on February 17, 2015

Python and Django versions supported:

- hvad now supports Django 1.8.
- Django 1.5 has reached its end of life, and support will be dropped in hvad 1.2. Note however that Django 1.4 will still be supported.

New features:

- It is now possible to use translated fields in the `unique_together` and `index_together` settings on *TranslatableModel*. They cannot be mixed in a single constraint though, as table-spanning indexes are not supported by SQL databases.
- The `annotate()` method is now supported. Support is still basic for now: annotations may not access more than one level of relation.

Compatibility warnings:

- Internal module `hvad.fieldtranslator` was no longer used, and was incompatible with Django 1.8. It has been removed.
- Deprecated `using_translations()` has been removed. It can be safely replaced by *`language()`*.

- Deprecated `TranslationFallbackManager` has been removed. Please use manager's *`untranslated()`* method instead.
- Deprecated `TranslatableModelBase` metaclass has been removed. Since release 0.5, hvad does not trigger metaclass conflicts anymore – #188.
- Overriding the language in `QuerySet.get()` and `QuerySet.filter()` was deprecated in release 0.5, and has now been removed. Either use the *`language()`* method to set the correct language, or specify *`language('all')`* to filter manually through `get` and `filter` – #182.
- `TranslatableModel`'s Internal attribute `_shared_field_names` has been removed.

Deprecation list:

- Passing `unique_together` or `index_together` as a `meta` option on *`TranslatedFields`* is now deprecated and will be unsupported in release 1.3. Put them in the model's `Meta` instead, alongside normal fields.
- Calling `save()` on an invalid *TranslatableModelForm* is a bad practice and breaks on regular Django forms. This is now deprecated, and relevant checks will be removed in release 1.3. Please check the form is valid before saving it.
- Generic views in `hvad.views` have been refactored to follow Django generic view behaviors. As a result, several non-standard methods are now deprecated. Please replace them with their Django equivalents — check #225.

1.0.0

Released on December 19, 2014

Python and Django versions supported:

- Django 1.3 is no longer supported.
- Python 2.6 is no longer supported. Though it is likely to work for the time being, it has been dropped from the tested setups.

New features:

- *TranslatableModelForm* has been refactored to make its behavior more consistent. As a result, it exposes two distinct language selection modes, *normal* and *enforce*, and has a clear API for manually overriding the language — #221.
- The new features of `modelform_factory()` introduced by Django 1.6 and 1.7 are now available on *translatable_modelform_factory* as well — #221.
- *TranslationQueryset* now has a *fallbacks()* method when running on Django 1.6 or newer, allowing the queryset to use fallback languages while retaining all its normal functionalities – #184.
- Passing additional `select` items in method `extra()` is now supported. — #207.
- It is now possible to use *TranslationQueryset* as default queryset for translatable models. — #207.
- A lot of tests have been added, hvad now has 100% coverage on its core modules. Miscellaneous glitches found in this process were fixed.
- Added MySQL to tested database backends on Python 2.7.

Compatibility warnings:

- *TranslatableModelForm* has been refactored to make its behavior more consistent. The core API has not changed, but edge cases are now clearly specified and some inconsistencies have disappeared, which could create issues, especially:

  - Direct use of the form class, without passing through the *factory method*. This used to have an unspecified behavior regarding language selection. Behavior is now well-defined. Please ensure it works the way you expect it to.

Fixes:

- *TranslatableModelForm*'s `clean()` can now return *None* as per the new semantics introduced in Django 1.7. — #217.
- Using `Q object` logical combinations or `exclude()` on a translation-aware manager returned by *`get_translation_aware_manager()`* no longer yields wrong results.
- Method `get_or_create()` now properly deals with Django 1.6-style transactions.

0.5.2

Released on November 8, 2014

Fixes:

- Admin does not break anymore on M2M fields on latest Django versions. — #212.
- Related fields's `clear()` method now works properly (it used to break on MySQL, and was inefficient on other engines) — #212.

0.5.1

Released on October 24, 2014

Fixes:

- Ecountering a regular (un-translatable) model in a deep *select_related* does not break anymore. — #206.
- Language tabs URI are now correctly generated when changelist filters are used. — #203.
- Admin language tab selection is no longer lost when change filters are active. — #202.

0.5.0

Released on September 11, 2014

New features:

- New *translationformset_factory* and its companion *`BaseTranslationFormSet`* allow building a formset to work on an instance's translations. Please have at look at its detailed *documentation* – #157.
- Method *`language()`* now accepts the special value `'all'`, allowing the query to consider all translations – #181.
- Django 1.6+'s new `datetimes()` method is now available on *`TranslationQueryset`* too – #175.
- Django 1.6+'s new `earliest()` method is now available on *`TranslationQueryset`*.
- Calls to *`language()`*, passing `None` to use the current language now defers language resolution until the query is evaluated. It can now be used in form definitions directly, for instance for passing a custom queryset to `ModelChoiceField` – #171.
- Similarly, *`use_fallbacks()`* can now be passed `None` as one of the fallbacks, and it will be replaced with current language at query evaluation time.

- All queryset classes used by *TranslationManager* can now be customized thanks to the new *fallback_class* and *default_class* attributes.
- Abstract models are now supported. The concrete class must still declare a *TranslatedFields* instance, but it can be empty – #180.
- Django-hvad messages are now available in Italian – #178.
- The Meta.ordering model setting is now supported on translatable models. It accepts both translated and shared fields – #185, #12.
- The select_related() method is no longer limited to 1 level depth – #192.
- The select_related() method semantics is now consistent with that of regular querysets. It supports passing None to clear the list and mutiple calls mimic Django behavior. That is: cumulative starting from Django 1.7 and substitutive before – #192.

Deprecation list:

- The deprecated nani module was removed.
- Method using_translations() is now deprecated. It can be safely replaced by *language()* with no arguments.
- Setting NANI_TABLE_NAME_SEPARATOR was renamed to HVAD_TABLE_NAME_SEPARATOR. Using the old name will still work for now, but issue a deprecation warning, and get removed in next version.
- CSS class nani-language-tabs in admin templates was renamed to hvad-language-tabs. Entities will bear both classes until next version.
- Private _real_manager and _fallback_manager attributes of *TranslationQueryset* have been removed as the indirection served no real purpose.
- The TranslationFallbackManager is deprecated and will be removed in next release. Please use manager's *untranslated()* method instead.
- The TranslatableModelBase metaclass is no longer necessary and will be removed in next release. hvad no longer triggers metaclass conflicts and TranslatableModelBase can be safely dropped – #188.
- Overriding the language in QuerySet.get() and QuerySet.filter() is now deprecated. Either use the *language()* method to set the correct language, or specify *language('all')* to filter manually through get and filter – #182.

Fixes:

- Method latest() now works when passed no field name, properly getting the field name from the model's Meta.get_latest_by option.
- *FallbackQueryset* now leverages the better control on queries allowed in Django 1.6 and newer to use only one query to resolve fallbacks. Old behavior can be forced by adding HVAD_LEGACY_FALLBACKS = True to your settings.
- Assigning value to translatable foreign keys through its _id field no longer results in assigned value being ignored – #193.
- Tests were refactored to fully support PostgreSQL – #194

0.4.1

Released on June 1, 2014

Fixes:

- Translations no longer remain in database when deleted depending on the query that deleted them – #183.

- *get_available_languages()* now uses translations if they were prefetched with prefetch_related(). Especially, using *all_translations()* in list_display no longer results in one query per item, as long as translations were prefetched – #179, #97.

0.4.0

Released on May 19, 2014

New Python and Django versions supported:

- django-hvad now supports Django 1.7 running on Python 2.7, 3.3 and 3.4.
- django-hvad now supports Django 1.6 running on Python 2.7 and 3.3.

New features:

- *TranslationManager*'s queryset class can now be overriden by setting its *queryset_class* attribute.
- Proxy models can be used with django-hvad. This is a new feature, please use with caution and report any issue on github.
- *TranslatableAdmin*'s list display now has direct links to each available translation.
- Instance's translated fields are now available to the model's save() method when saving a *TranslatableModelForm*.
- Accessing a translated field on an untranslated instance will now raise an AttributeError with a helpful message instead of letting the exception bubble up from the ORM.
- Method *in_bulk()* is now available on *TranslationQueryset*.

Deprecation list:

- Catching ObjectDoesNotExist when accessing a translated field on an instance is deprecated. In case no translation is loaded and none exists in database for current language, an AttributeError is raised instead. For the transition, both are supported until next release.

Removal of the old 'nani' aliases was postponed until next release.

Fixes:

- Fixed an issue where *TranslatableAdmin* could overwrite the wrong language while saving a form.
- *lazy_translation_getter()* now tries translations in LANGUAGES order once it has failed with current language and site's main LANGUAGE_CODE.
- No more deprecation warnings when importing only from hvad.
- *TranslatableAdmin* now generates relative URLs instead of absolute ones, enabling it to work behind reverse proxies.
- django-hvad does not depend on the default manager being named 'objects' anymore.
- Q objects now work properly with *TranslationQueryset*.

0.3

New Python and Django versions supported:

- django-hvad now supports Django 1.5 running on Python 2.6 and 2.6.

Deprecation list:

- Dropped support for django 1.2.

- In next release, the old 'nani' module will be removed.

0.2

The package is now called 'hvad'. Old imports should result in an import error.

Fixed django 1.4 support

Fixed a number of minor issues

0.1.4 (Alpha)

Released on November 29, 2011

- Introduces *lazy_translation_getter()*

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)

0.0.4 (Alpha)

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.

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.

0.0.1 (Alpha)

Released on May 13, 2011.

- First release, for testing purposes only.

Contact and support channels

- Github: https://github.com/KristianOellegaard/django-hvad

How to contribute

Running the tests

Common Setup

- `virtualenv env`
- `source env/bin/activate`
- `pip install django sphinx djangorestframework`

Postgres Setup

Additional to the steps above, install `psycopg2` using pip and have a postgres server running that you have access to with a user that can create databases.

Mysql Setup

Additional to the steps above, install `mysql-python` using pip and have a mysql server running that you have access to with a user that can create databases.

Run the test

- `python runtests.py`

Optionally, prefix it with a environment variable called `DATBASE_URL`, for example for a Postgres server running on `myserver.com` on port `5432` with the user `username` and password `password` and database name `hvad`:

- `DATABASE_URL=postgres://username:password@myserver.com:5432/hvad python runtests.py`

If in doubt, you can check `.travis.yml` for some examples.

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.

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

Internal API Documentation

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.

Contents

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

General information on django-hvad internals

How it works

Model Definition

Function *hvad.models.prepare_translatable_model()* is invoked by Django metaclass using class_prepared signal. It scans all attributes on the model defined for instances of *hvad.models.TranslatedFields*, and if it finds one, sets the respective options onto meta.

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 *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 *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_translation_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 languages.

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 *untranslated()* on the queryset returned by the call to the super class and returns that queryset. This allows showing all objects, even if they have no translation in current language, at the cost of more database queries.

_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 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

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. If no translation is cached yet, use the `get_language()` function to get the current language, load it from the database and cache it on the instance.

If no translation is cached, and no translation exists for current language, raise an `AttributeError`.

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 field object itself, allowing introspection of the model.

Starting from Django 1.7, calling `getattr()` on a translated field before the App Registry is initialized raises an `AttributeError`.

__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*)
Uses Django's internal fields_for_model to get translated fields for model and fields declarations, then lets Django handle the other fields. Once it is done, it merges the translated fields, preserving order.

Special handling is done to:

- Prevent `language_code` from being used in any way by a field. This is because the form uses the `language_code` key in the `cleaned_data` dictionary.
- Prevent `master` from being recognized as a translated field. It is still a valid field name though.
- Prevent the translations accessor from being used as a field.

TranslatableModelForm

class `hvad.forms.`**BaseTranslatableModelForm**(*BaseModelForm*)

The actual class supporting the features and methods, but lacking metaclass sugar. Inherited by *TranslatableModelForm* to attach the metaclass. Details are documented on that class.

class `hvad.forms.`**TranslatableModelForm**(*BaseTranslatableModelForm*)

Main form for editing *TranslatableModel* instances. As with regular django `Form` classes, it can be used either directly or by passing it to *translatable_modelform_factory()*.

As an extension to regular forms, it handles translation and can be bound to a language. Binding to a language is done by setting *language* on the class (not the instance), either by inheriting it manually or using the factory function. Once bound to a language, the form is in **enforce** mode: all manipulations will be done using that language exclusively.

__metaclass__

TranslatableModelFormMetaclass

language

The language the form is bound to. This is a class attribute. If present, the form is in **enforce** mode and will only deal with the specified language. See each method for the exact effects.

__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, that has a translation loaded, it updates `initial` to also contain the data from the *Translations Model*.

If the form is not bound to a language, it will use the data from the instance. If the instance has no translation loaded, an attempt will be made at loading the current language, and if that fails the fields will be blank.

If the form is in **enforce** mode and the instance does not have the correct translation loaded, then:

- it will attempt to load it from the database.
- if that fails, it will try to use the loaded translation on the instance.
- if that fails (instance is untranslated), it will use default values.

This process results in new translations being pre-populated with data from another language. Simply pass an instance in that language, or an untranslated instance if the behavior is not desired.

clean(*self*)

If the form is in **enforce** mode, namely if it has a `language` property, apply the it to `cleaned_data`. As usual, the special value `None` is replaced by current language.

If the form is not bound to a language, this method does nothing. It is then possible to either use *save()* in unbound mode or set the language code manually in `cleaned_data['language_code']`.

Note: A missing language is not the same as `None`. While `None` will be replaced by current language and applied to `cleaned_data`, a missing language will not apply any language at all.

_post_clean(*self*)

Loads a translation appropriate to the form mode. It is the very same that will be loaded by *save()*. Doing it twice is needed because:

- it must be done in `_post_clean` so that the correct translation is available for modifications. For instance, if the view updates some translated fields in between the call to `is_valid()` and `save()`, or if a form defines a custom `save()`.
- it must also be done in `save` to ensure the language is correctly enforced when in **enforce** mode.

This double check has no cost: unless the instance is changed by the view, the `save()` check will see the translation is correct and do nothing.

save(*self*, *commit=True*)

Saves both the *Shared Model* and *Translations Model* and returns a combined model.

The target language is determined as follows:

- If a language is defined in `cleaned_data`, that language is used.
- Else, if the instance has a translation loaded, its language is used.
- Else, the current language is used.

Once the language is determined, the following happen:

- If the object does not exist, it is created.
- If the object exists but not in the target language, its shared fields are updated and a new translation is created.
- If the object exists in the target language, it is updated.

Note: The **enforce** mode has no direct impact on this method. Rather, it affects the behavior of *clean()*, which places relevant language (or lack thereof) in `cleaned_data`.

---

hvad.forms.**translatable_modelform_factory**(*language*, *model*, *form=TranslatableModelForm*, ***kwargs*)

Attaches a language and a model class to the specified form and returns the resulting class. Additional arguments are any arguments accepted by Django's `modelform_factory()`, including `fields` and `exclude`.

Having a language attached, the returned form is in **enforce** mode.

hvad.forms.**translatable_modelformset_factory**(*language*, *model*, *form=TranslatableModelForm*, ***kwargs*)

Creates a formset class, allowing edition a collection of instances of `model`, all of them in the specified `language`. Additional arguments are any argument accepted by Django's `modelformset_factory()`.

Having a language attached, the returned formset is in **enforce** mode.

hvad.forms.**translatable_inlineformset_factory**(*language*, *parent_model*, *model*, *form=TranslatableModelForm*, ***kwargs*)

Creates an inline formset, allowing edition of a collection of instances of `model` attached to an instance of `parent_model`, all of those objects being in the specified `language`. Additional arguments are any argument accepted by Django's `inlineformset_factory()`.

Having a language attached, the returned formset is in **enforce** mode.

BaseTranslationFormSet

class hvad.forms.**BaseTranslationFormSet**(*BaseInlineFormSet*)

instance
An instance of a *TranslatableModel* that the formset works on the translations of. Its untranslatable fields will be used while validating and saving the translations.

order_translations(*self*, *qs*)
Is given a queryset over the *Translations Model*, that it should alter and return. This is used for adding **order_by** clause that will define the order in which languages will show up in the formset.

Default implementation orders by **language_code**. If overriding this method, the default implementation should not be called.

clean(*self*)
Performs translation-specific cleaning of the form. Namely, it combines each form's translation with *instance* then calls full_clean() on the full object.

It also ensures the last translation of an object cannot be deleted (unless adding a new translation at the same time).

_save_translation(*self*, *form*, *commit=True*)
Saves one of the formset's forms to the database. It is used by both *save_new()* and *save_existing()*. It works by combining the form's translation with *instance*'s untranslatable fields, then saving the whole object, triggering any custom save() method or related signal handlers.

save_new(*self*, *form*, *commit=True*)
Saves a new translation. Called from save().

save_existing(*self*, *form*, *instance*, *commit=True*)
Saves an existing, updated translation. Called from save().

add_fields(*self*, *form*, *index*)
Adds a **language_code** field if it is not defined on the translation form.

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 `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.

SkipMasterSelectMixin

class hvad.manager.**SkipMasterSelectMixin**
A mixin class for specialized querysets such as `DateQuerySet` and `DateTimeQuerySet` which forces *TranslationQueryset* not to add the related lookup on the *master* field. This is required as those specialized querysets use DISTINCT, and added the related lookup brings along all fields on the *Shared Model*, breaking the lookup.

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.

_language_code
The language code of this queryset, or one of the following special values:

- `None`: `get_language()` will be called to get the current language.
- `'all'`: no language filtering will be applied, a copy of an instance will be returned for every translation that matched the query.

_language_fallbacks
A tuple of fallbacks used for this queryset, if fallbacks have been activated by *fallbacks()*, or *None* otherwise.

A `None` value in the tuple will be replaced with current language at query evaluation.

_hvad_switch_fields
A tuple of attributes to move from the *Translations Model* to the *Shared Model* instance before returning objects to the caller. It is mostly used by `extra()` so additional values collected by the `select` argument are available on the final instance.

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 (`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 `Q` object and translates it's query lookups to be prefixed by `'master__'` if they access a field on *Shared Model*.

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

This method returns a new `Q` object.

_find_language_code(*self*, *q*)
Searches a `Q` object for language code lookups. If it finds a child `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 `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 `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_queryset(*self*)
Returns a clone of this queryset but for the shared model. Does so by creating a `QuerySet` on *shared_model* and filtering over this queryset. Returns a queryset for the *Shared Model*.

_add_language_filter(*self*)
Apply the language filter to current query. Language is retrieved from *_language_code*, or `get_language()` if None. If *fallbacks()* have been set, apply the additional join as well.

Special value `'all'` will prevent any language filter from being applied, resulting in the query considering all translations, possibly returning the same instance mutiple times if several of its translations match. In that case, each instance will be *combined* with one of the matching translations.

Applied filters include the base language filter on the language_code field, as well as any related model translation set up by *select_related()*.

_add_select_related(*self*, *language_code*)
New in version 0.5.

Applies the related selections to current query. This includes the basic selection of `master`, any relation specified through *select_related()* and the translations of any translatable models it navigates through.

language(*self*, *language_code=None*)
Specifies a language for this queryset. This sets the *_language_code*, but no filter are actually applied until *_add_language_filter()* is called. This allows for query-time resolution of the None value. It is an error to call *language()* multiple times on the same queryset.

The following special values are accepted:

- `None`, or no value: `get_language()` will be called to get the current language.
- `'all'`: no language filtering will be applied, a copy of an instance will be returned for every translation that matched the query, each copy being *combined* with one of the matching translations.

Returns a queryset.

Note: Support for using `language('all')` and *select_related()* on the same queryset is experimental. Please check the generated queries and open an issue if you have any problem. Feedback is appreciated as well.

---

fallbacks(*self*, **languages*)
New in version 0.6.

Activates fallbacks for this queryset. This sets the *_language_fallbacks* attribute, but does not apply any join or filtering until *_add_language_filter()* is called. This allows for query-time resolution of the `None` values in the list.

The following special cases are accepted:

- `None` as a single argument will disable fallbacks on the queryset.
- An empty argument list will use `LANGUAGES` setting as a fallback list.
- A `None` value a language will be replaced by the current language at query evalution time, by calling `get_language()`

Returns a queryset.

Note: Using `fallbacks` and *select_related()* on the same queryset is not supported and will raise a `NotImplementedError`.

Note: This feature requires Django 1.6 or newer.

---

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 `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.

Note: It is an error to call `create` with no `language_code` on a queryset whose *_language_code* is `'all'`. Doing so will raise a `ValueError`.

---

bulk_create(*self*, *objs*, *batch_size=None*)
Not implemented yet and unlikely to be due to inherent limitations of multi-table inserts.

update_or_create(*self*, *defaults=None*, ***kwargs*)
Not implemented yet.

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 `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 will use the language of this queryset from *_language_code*, or the current language as returned by `get_language()` of that is None.

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.

> **Warning:** It is an error to pass *language_code* in a Q object if a *select_related()* clause was enabled on this queryset. Doing so will raise an `AssertionError`.

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.

earliest(*self*, *field_name=None*)
New in version 0.4.

Translates the fieldname (if given) using *field_translator* and calls the superclass.

Only defined if django version is 1.6 or newer.

in_bulk (*self*, *id_list*)
New in version 0.4.

Retrieves the objects, building a dict from `iterator()`.

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

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_queryset()` 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 `_translate_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.

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

Only defined if django version is 1.6 or newer.

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 `_translate_fieldnames()` and calls the superclass.

reverse (*self*)
Calls the superclass.

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*, and *shared_model* 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__(). This is because the former uses the latter, which in turn fetches results from an iterator.

TranslationManager

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

translations_model
The *Translations Model* for this manager.

queryset_class
The QuerySet for this manager, used by the *language()* method. Overwrite to use a custom queryset. Your custom queryset class must inherit *TranslationQueryset*. Defaults to *TranslationQueryset*.

fallback_class
The QuerySet for this manager, used by the *untranslated()* method. Overwrite to use a custom queryset. Defaults to *FallbackQueryset*.

default_class
The QuerySet for this manager, used by the *get_queryset()* method and generally any query that does not invoke either *language()* or *untranslated()*. Overwrite to use a custom queryset. Defaults to QuerySet.

language(*self*, *language_code=None*)
Instanciates a *TranslationQueryset* from *queryset_class* and calls *TranslationQueryset.language()* on that queryset. This type of queryset will filter by language, returning only objects that have a translation in the specified language. Translated fields will be available on the objects, in the specified language.

untranslated(*self*)
Returns an instance of *FallbackQueryset* for this manager, or any custom queryset defined by *fallback_class*. This type of queryset will load translations using fallbacks if current language is not available. It can generate a lot a queries, use with caution.

get_queryset(*self*)
Returns a vanilla, non-translating queryset for this manager. It uses the default QuerySet or any custom queryset defined by *default_class*.

Instances returned will not have translated fields, and attempts to access them will result in an exception being raised. See *language()* and *untranslated()* to access translated fields.

It is possible to override this behavior by setting *default_class* to *TranslationQueryset*, *FallbackQueryset* or any queryset that has a translation-aware implementation.

contribute_to_class(*self*, *model*, *name*)
Contributes this manager onto the class.

FallbackQueryset

class hvad.manager.**FallbackQueryset**

Deprecated since version 1.4.

A queryset that can optionally use fallbacks and by default only fetches the *Shared Model*.

There are actually two underlying implementations, the `LegacyFallbackQueryset` and the `SelfJoinFallbackQueryset`. Implementation is chosen at initialization based on the `HVAD_LEGACY_FALLBACKS` setting. It defaults to `False` (use SelfJoin) on Django 1.6 and newer, and `True` (use Legacy) on older versions.

The `LegacyFallbackQueryset` generates lots of queries as it walks through batches of models, fetches their translations and matches them onto the models.

The `SelfJoinFallbackQueryset` uses a single self outer join to achieve the same result in only one (complex) query. Performance is good as the number of items per model in the cross-product is limited to the number of languages that Django supports. Implementation digs deeper into Django internals, though.

_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*)

Deprecated since version 1.4.

If this method gets called, *iterator()* will use the fallbacks defined here. *None* value will be replaced with current language at query evaluation, as returned by `get_language()`. If not fallbacks are given, *FALLBACK_LANGUAGES* will be used, with current language prepended.

This method has been superseded by *fallbacks()* and will be removed when support for Django 1.4 is dropped.

_clone(*self*, *klass=None*, *setup=False*, ***kwargs*)

Injects *translation_fallbacks* into *kwargs* and calls the superclass.

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 `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 `Q` object to use to filter for the current language.

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

Returns a tuple of *q* and *language_joins* where *q* is the translated 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 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 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 Q object.

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

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

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

> **Warning:** This internal method returns a `super()` proxy object, be sure to understand the implications before using it.

TranslationAwareManager

class hvad.manager.**TranslationAwareManager**

get_queryset(*self*)
Returns an instance of *TranslationAwareQueryset*.

hvad.models

hvad.models.**prepare_translatable_model**(*sender*)
Gets called from Model after Django has completed its setup. It customizes model creation for translations. Most notably, it performs checks, overrides _meta methods and defines translation-aware manager on models that inherit *TranslatableModel*.

TranslatedFields

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

contribute_to_class(*self*, *cls*, *name*)
Invoked by Django while setting up a model that defines translated fields. Django passes is the model being built as `cls` and the field name used for translated fields as `name`.

It triggers translations model creation from the list of field the `TranslatedFields` object was created with, and glues the shared model and the translations model together.

create_translations_model(*self*, *model*, *related_name*)
A model factory used to create the *Translations Model* for the given shared `model`. The translations model will include:

- A foreign key back to the shared model, named `master`, with the given `related_name`.
- A `language_code` field, indexed together with `master`, for looking up a shared model instance's translations.
- All fields passed to `TranslatedFields` object.

Adds the new model to the shared model's module and returns it.

contribute_translations(*self*, *model*, *translations_model*, *related_name*)
Glues the shared `model` and the `translations_model` together. This step includes setting up attribute descriptors for all translatable fields onto the shared `model`.

_scan_model_bases(*self*, *model*)
Recursively walks all `model`'s base classes, looking for translation models and collecting translatable fields. Used to build the inheritance tree of a *Translations Model*.

Returns the list of bases and the list of fields.

_build_meta_class(*self*, *model*, *tfields*)
Creates the `Meta` class for the *Translations Model* passed as `model`. Takes `tfields` as a list of all fields names referring to translatable fields.

Returns the created meta class.

static _split_together(*constraints*, *fields*, *name*)
Helper method that partitions constraint tuples into shared-model constraints and translations model constraints. Argument `constraints` is an iterable of contrain tuples, `fields` is the list of translated field names and `name` is the name of the option being handled (used for raising exceptions).

Returns two list of constraints. First for shared model, second for translations model. Raises an `ImproperlyConfigured` exception if a constraint has both translated and untranslated fields.

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.

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, `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*.

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*.

Returns value of translated field `name`, unless no translation is loaded, or loaded translation doesn't have field `name`. In both cases, it will return `default`, performing no database query.

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.

Note: Use is discouraged on production code paths. It is mostly intended as a helper method for introspection.

---

get_available_languages(*self*)
Returns a list of language codes in which this instance is available. Uses cached values if available (eg if object was loaded with `.prefetch_related('translations')`), otherwise performs a database query.

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.query

This modules containts abstractions for accessing some internal parts of Django ORM that are used in hvad. The intent is that anytime some code in hvad needs to access some Django internals, it should do so through a function in this module.

hvad.query.**query_terms**(*model*, *path*)
This iterator yields all terms in the specified `path`, along with full introspection data. Each term is output as a named tuple with the following members:

- `depth`: how deep in the path is this term. Counted from zero.
- `term`: the term string.
- `model`: `` the model the term is attached to. It will start with passed ``model then walk through relations as terms are enumerated.
- `field`: the actual field, on the model, the term refers to.
- `translated`: whether the field is a translated field (True) or a shared fielf (False).
- `target`: the target model of the relation, or `None` if not a relational field.
- `many`: whether the target can be multiple (that is, it is a M2M or reverse FK).

If a field is not recognized, it is assumed the path is complete and everything that follows is a query expression (such as `__year__in`). Query expression terms will be yielded with `field` set to `None`.

hvad.query.**q_children**(*q*)
Iterator that recursively yields all key-value pairs of a `Q` object. Each pair is yielded as a 3-tuple: the pair itself, its container and its index in the container. This allows modifying it.

hvad.query.**expression_nodes**(*expression*)
Iterator that recursively yields all nodes in an expression tree.

hvad.query.**where_node_children**(*node*)
Iterator that recursively yields all fields of a where node. It is used to determine whether a custom `Q` object included a `language_code` filter.

hvad.utils

hvad.utils.**get_cached_translation**(*instance*)
Returns the cached translation from an instance or `None`. Encapsulates a `getattr()` using the model's **translations_cache**.

hvad.utils.**set_cached_translation**(*instance*, *translation*)
Sets the currently cached translation for the instance, and returns the translation that was loaded before the call. Passing `None` as translation will unload current translation and let the instance untranslated.

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

klass is the *Shared Model* class. This argument is required as there is no way to distinguish a translation of a proxy model from that of a concrete model otherwise.

This function is only intended for loading models from the database. For other uses, *set_cached_translation()* should be used instead.

hvad.utils.**get_translation**(*instance*, *language_code=None*)
Returns the translation for an instance, in the specified language. If given language is None, uses get_language() to get current language.

Encapsulates a getattr() using the model's **translations_accessor** and a call to its get() method using the instance's primary key and given language_code as filters.

hvad.utils.**load_translation**(*instance*, *language*, *enforce=False*)
Returns the translation for an instance.

- If enforce is False, then language is used as a default language, if the instance has no language currently loaded.
- If enforce is True, then language will be enforced upon the translation, ignoring cached translation if it is not in the given language.

A valid translation instance is always returned. It will be loaded from the database as required. If this fails, a new, empty, ready-to-use translation will be returned.

The instance itself is untouched.

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.

This descriptor is pending deprecation as the associated method is being removed from Django.

__init__(*self*, *real*)
Retains a reference to the actual method this descriptor is replacing.

__call__(*self*, *meta*, *name*)
Catches improper use of the get_field_by_name method to access translated fields and raise a WrongManager exception.

class hvad.utils.**SmartGetField**
Smart version of the standard get_field() 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*)
Retains a reference to the actual method this descriptor is replacing.

__call__(*self*, *meta*, *name*)
Catches improper use of the get_field method to access translated fields and raise a WrongManager exception.

class hvad.utils.**_MinimumDjangoVersionDescriptor**
Helper class used by *minimumDjangoVersion()* decorator.

hvad.utils.**minimumDjangoVersion**(**args*)
Decorator that will catch attempts to use methods on a Django version that does not support them and raise a helpful exception.

Arguments must be the minimum allowable Django version, the will be compared against the django.VERSION tuple.

settings_updater(func):
Decorator for setting globals depending on Django settings. It simply invokes the decorated function immediately, then calls it again every time the `setting_changed` signal is sent by Django.

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 *TranslatableModel*.

Translation Manager A subclass of *TranslationManager*, which replaces the default Django manager on Translated Model, allowing access to translated fields. It will use *TranslationQueryset* internally, or a custom subclass if so configured.

Translation-Aware Manager A Django manager that operates on **untranslated** models, yet is aware of translated models it meets when crossing relations. It makes it possible to filter untranslatable models against a translated field of a related model.

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*.

Indices and tables

- genindex
- modindex
- search
- *Glossary*

Python Module Index

h

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W