django-easymode Documentation

Release 1.4b5

Lars van de Kerkhof

July 30, 2015

Contents

With easymode you can create backends for dynamic flash/flex websites. Easymode makes internationalization simple and outputs xml by default. To tailor the xml to your application, you can transform it using xslt templates, which easymode integrates.

For more info, look at solipsism

CHAPTER 1

Manual

1.1 Release Notes

1.1.1 v1.4b5

Fixes compatibity with django 1.7. Use setuptools instead of distutils.

1.1.2 v1.4b4

Fixes Exception in `standin_for()`: a new-style class can't have only classic bases.

1.1.3 v1.4b3

Fixed Django 1.5 template requirements for admin media.

1.1.4 v1.4b2

Fixes I18n for models with tuple permissions. Allow custom admin class for register_all. fixes wrong import in easymode.tree.admin.forms.

1.1.5 v1.4b1

This release fixes most incompatibilities of easymode with django 1.4. Most importantly, it fixes i18n. This release is therefor meant to be used with django 1.4.

1.1.6 v1.0b1

Easymode is now moving towards the 1.0 release. I took the liberty of removing code whose usefulness was dubious and also refactor mercilessly. You will also find some very nice new features in this release!

That being said, you can not simply update and run an existing application on this release!

Backward incompatible changes:

- The names of localized table columns are constructed differently now. Upgrading means migrating all your column names!

- Easypublisher was removed, because it was too hard to maintain.
- All xml related code moved to the package easmode.tree.xml.
- `DiocoreCharField`, `DiocoreHTMLField`, `DiocoreTextField`, `CSSField`, `IncludeFileField`, `RemoteIncludeField` where removed.

New features:

- New tree module, which uses *real* inlines See *Admin support for model trees with more than 2 levels of related items*.
- You can now hook into xml serialization and have custom serialization for both your models as your custom model fields See *When the standard serializer is not enough*.

Bugs fixed:

- Fields marked for translation with `I18n` can now be sorted on in the admin when they are included in `list_display`.
- Order_by now works on translated fields with 5 letter locales:

```
from django.utils.languagecode import get_real_fieldname

# this now works:
MyModel.objects.order_by(get_real_fieldname('somefield', 'en-us'))
```

1.1.7 v0.14.5

Models decorated with `I18n` nolonger have problems deleting related models in cascade mode.

1.1.8 v0.14.4

Easymode nolonger installs any packages automatically during installation, these should now be installed by hand.

1.1.9 v0.14.3

Using `super` in an admin class decorated with L10n, will nolonger result in infinite recursion.

1.1.10 v0.14.2

Fixes `SafeHTMLField`'s `buttons` property, which can be used to override the tinymce buttons per field.

1.1.11 v0.14.1

ForeignKeyAwareModelAdmin now properly handles parent_link that points to a model in a different app.

1.1.12 v0.14.0

Added nofollow option to mark foreign keys that shouldn't be followed by the serializer. Nofollow can be used to optimize easymodes queries when generating xml, see *Exclude certain relations from being followed by the serializer*.

1.1.13 v0.13.7

- Fallbacks for translatable fields now also work when the first fallback is not the MSGID_LANGUAGE.

1.1.14 v0.13.6

- You can now override the model form of an admin class decorated with L10n, just like normal admin classes.

1.1.15 v0.13.5

- Easymode nolonger patches SubFieldBase. Fields that throw Exceptions when their descriptor is accessed can now also be internationalized using I18n. This includes ImageField and FileField.

1.1.16 v0.13.4

- `standin_for()` now returns a standin that can be pickled and unpickled.

1.1.17 v0.13.3

- register_all will nolonger try to register abstract models
- search_fields is now supported for ModelAdmin classes that use L10n, however it will not let you access related items.
- You can now use fieldsets with the *Can edit untranslated fields* permission.
- Added support for creating new objects to easypublisher.
- Added tools to build preview functionality for drafts.
- Added filter that removes unpublished items from the xml.
- fixed error 'cannot import name introspection' caused by a circular import.

1.1.18 v0.10.5

- Added option to exclude models from register_all
- Backwards incompatible change: easymode nolonger has any bindings for django-cms.
- Easymode will now show you the origin of a value, by displaying symbols next to the input field in the admin:
  1. If a value is from the gettext catalog or fallback, easymode will display °
  2. If a value is from the database, but the catalog has a different value, easymode will display . You can hover over this symbol to see the catalog value.
  3. If a value is from the database and there is no conflict with the catalog, easymode will display only
- Fixed bug where a value that evaluated to *None* was set with the `string` **None** instead of `types.NoneType` `None`
- fixes bug where get_localized_property would crash if settings did not have FALLBACK_LANGUAGES defined.
- `django.db.models.ManyToManyField` and `django.db.models.ForeignKey` are now handled by the default xslt ('xslt/model-to-xml.xsl').

1.1.19 v0.9.3

- fixes `easymode.admin.abstract.AbstractOrderedModel`
- register_all will ignore models that are `django.contrib.admin.sites.AlreadyRegistered`, but still register other models in the module.

1.1.20 v0.9.2

- Fixed bug in recursion_depth context manager and added tests.

1.1.21 v0.9.1

- Modified the xslt parser to use the file path instead of a string, so you can use xsl:include now.
- libxsltmod is nolonger a supported xslt engine
- Added util to add register all models in some module in one go.

1.1.22 v0.8.6

- Easymode will nolonger complain about rosetta, polib and tinymce when none of the features that require these packages are used.
- Moved polib util to `easymode.utils.polibext` to avoid name clashes
- `DiocoreTextField` now accepts cols and rows as parameters.
- The mechanism to add extra attributes to the xml produced by the serializer is now more generic. If a field has the 'extra_attrs' property, these attributes will be added as attributes to the field xml.
- Updated the serializer to support natural keys: http://docs.djangoproject.com/en/dev/topics/serialization/#natural-keys
- Now easymode can automatically serialize many to many fields. The recursion is guarded, and will let you know when you made a cyclic relation in you model tree. (see *RECURSION_LIMIT*).
- *`mutex`* now raises *`SemaphoreException`* instead of doing sys.exit().
- When to_python returns a weird object on a field instead of a string, it is now converted to unicode before it is used as a msgid.

1.1.23 v0.6.1

- `DiocoreHTMLField` will now also show a tinymce editor when it is not internationalized.
- When there is a problem with monkey patching `django.db.models.SubfieldBase` easymode will throw an exception. (Monkey patch fixes http://code.djangoproject.com/ticket/12568).
- New field aded, `CSSField`, which allows specification of css classes for a rich text field, the css classes will appear in the xml as:

```
    style="class1,class2"
```

1.1.24 v0.6.0

- Django 1.2 is required for easymode as of v0.6.0.
- `get_real_fieldname()` now returns a string instead of `unicode`. This way a `dict` can be constructed using it's results as keys, and the dict can be turned into keyword arguments of `filter` when doing a query in a specific language.
- Small improvements in error handling when *AUTO_CATALOG* is `True`

1.1.25 v0.5.7

- Added `easymode.admin.models.fields.SafeTextField`, a textfield which strips all cariage returns before saving, which is required when using *Automatic catalog management*.
- Updated django requirement to v1.1.2 because python 2.6.5 will otherwise make the unit tests fail.

1.1.26 v0.5.6

- The example app now has a working fixture.

1.1.27 v0.5.5

- Special admin widgets are nolonger discarded by easymode (issue #3)

1.1.28 v0.5.4

- Some data files where not installed correctly by setup.py

1.1.29 v0.5.3

- Added *AUTO_CATALOG* setting, see *Automatic catalog management*.
- Fixed error in *easy_locale* when two properties in the same model have the same value (eg. title and subtitle are the same).

1.2 Internationalization and localization of django models, with admin support

At times it becomes a requirement to translate models. Django supports internationalization of static text in templates and code by means of gettext. For translation of models - dynamic data - easymode offers simple decorators to enable internationalized model fields and localized admin classes.

1.2.1 Internationalization of models

Note: There is one requirement models fields have to satisfy to be able to be internationalised by easymode. Their `to_python()` method may not access `self`.

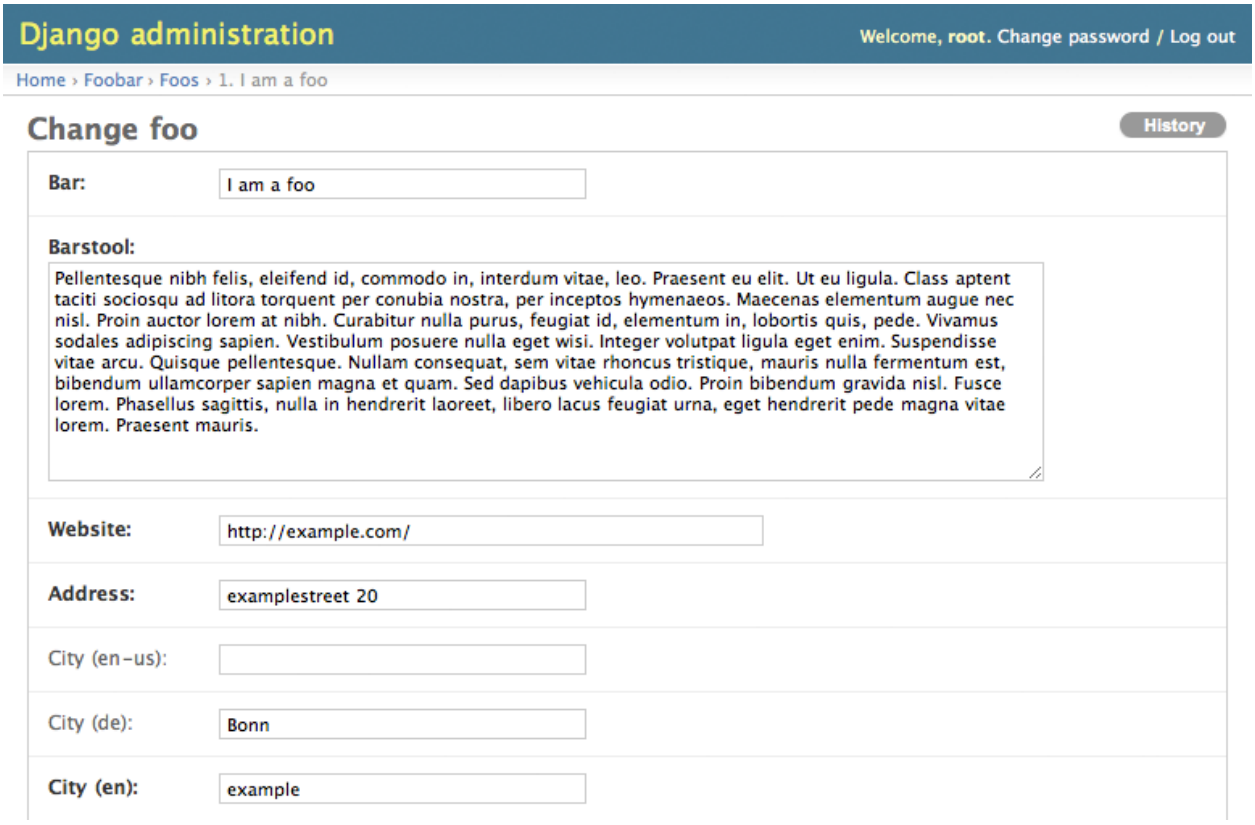
suppose we have the following model.

```
from django.db import models

class Foo(models.Model):
    bar = models.CharField(max_length=255, unique=True)
    barstool = models.TextField(max_length=4)
    website = models.URLField()
    address = models.CharField(max_length=32)
    city = models.CharField(max_length=40)
```

In different languages the city could have a different name, so we would like to make it translatable (eg. internationalize the city field). This can be done using the `I18n` decorator. Decorating the model as follows makes the city field translatable:

```
from django.db import models
from easymode.i18n.decorators import I18n

@I18n('city')
class Foo(models.Model):
    bar = models.CharField(max_length=255, unique=True)
    barstool = models.TextField(max_length=4)
    website = models.URLField()
    address = models.CharField(max_length=32)
    city = models.CharField(max_length=40)
```

Now the `city` field is made translatable. As soon as you register this model with the admin, you will notice this fact. Depending on how many languages you got in `LANGUAGES` this is how your change view will look:



While useful, the interface can become very cluttered when more fields need to be internationalized. To make the interface less cluttered the admin class that belongs to the model, can be *Localized* making it show only the fields in the current language.

1.2.2 Localization of models in django admin

As there are several options to register a model for inclusion in django's admin, there are also several options to localize the admin classes.

The simplest way to make a model editable in the admin is:

```
from django.contrib import admin
from foobar.models import Foo

admin.site.register(Foo)
```

Since the admin class is implicit here, there is no way we can localize the admin class this way. The next simplest way is:

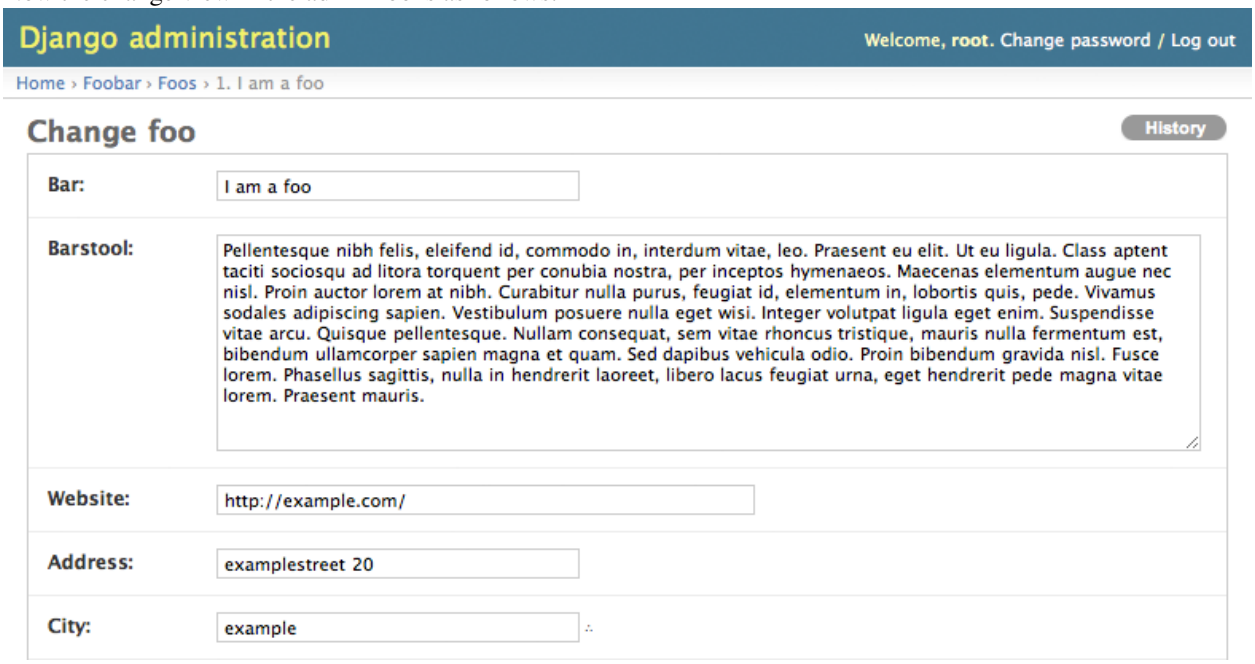
```
from django.contrib import admin
from foobar.models import Foo

admin.site.register(Foo, models.ModelAdmin)
```

Here the admin class is explicit, so we can modify it. The way this is done is by using the `L10n` class decorator:

```
from django.contrib import admin
from easymode.i18n.admin.decorators import L10n
from foobar.models import Foo
```

```
admin.site.register(Foo, L10n(Foo, models.ModelAdmin))
```

Note that the decorator needs the model to determine which fields are localized, so it must be passed as a parameter. Now the change view in the admin looks as follows:



All the 'city' fields are hidden, except for the field in the current language. Note That all fields which can be translated are marked with . To edit the content for the other languages, the current language must be switched. Please refer to *Translation of database content using gettext* for more details.

There is one more way a models can be registered for the admin and that is by creating a new descendant of `ModelAdmin` for a specific model. You can now also use the `L10n` decorator with the new class decorator syntax:

```
from django.contrib import admin
from easymode.i18n.admin.decorators import L10n

from foobar.models import Foo

@L10n(Foo)
class FooAdmin(admin.ModelAdmin):
    """Generic Admin class not specific to any model"""
    pass

admin.site.register(Foo, FooAdmin)
```

Note that you still have to pass the model class as a parameter to the decorator.

For admin classes that specify the `model` attribute you can leave that out:

```
from django.contrib import admin
from easymode.i18n.admin.decorators import L10n

from foobar.models import Foo

@L10n
class FooAdmin(admin.ModelAdmin):
```

```
    """Admin class for the Foo model"""
    model = Foo

admin.site.register(Foo, FooAdmin)
```

As you can see there isn't much to making models translatable this way.

1.2.3 Inline and GenericInline ModelAdmin

All easymode's localization mechanisms fully support django's flavors of `InlineModelAdmin`, both normal and generic. While there is no need to register these types of ModelAdmin classes, you still need to decorate them with `L10n` if you need them to be localized.

1.2.4 Fieldsets are also supported

`fieldsets` are supported for admin classes decorated with `L10n`. However `fields` is not supported, because easymode uses it to hide fields. Since you can do the exact same thing with fieldsets, this should not be a problem.

1.2.5 Don't internationalize relations

```
@I18n('available', 'text')
class SomeModel(models.Model):
    parent = models.ForeignKey('myapp.ParentModel', related_name='children')
    available = models.BooleanField(_('Available in this language'), default=True)
    text = models.TextField(_('The main issue'))
```

In the above example it is tempting to internationalize the parent relation, so you can exclude the content for some language, or maybe even give it an entirely different parent.

Most likely using `I18n` on `ForeignKey`, `ManyToManyField`, or `OneToOneField` is not going to work.

When you are internationalizing a relation, most of the time you want to make content available in one language, but maybe not the other. It is better to have an internationalized `BooleanField` and exclude content for other languages in that way.

When you've got different urls or domains for different languages, you should use the `django.contrib.sites` framework instead.

1.2.6 Use lazy foreign keys

You should always use lazy foreign keys in combination with the the `I18n` decorator. Lazy foreign keys helps to avoid cyclic imports, to which class decorators are extra sensitive.

If for example you've got your models in a package instead of a module, you need to import them all in the `__init__.py` module:

```
from bar.models.foo import *
from bar.models.baz import *
```

This way django will find them when it is collecting and verifying all models at boot time.

BUT!

Now you've got 2 ways to import the model Foo:

```
from bar.models import Foo
```

or:

```
from bar.models.foo import Foo
```

Django imports all models using the first syntax. If you would use the second to import the model somewhere else, in rare cases, the module get's initialized **twice**. This means the class decorator will get applied **twice**. And that gives you a very very strange error.

To avoid all this, just use lazy foreign keys everywhere. That way you never have to import models in other `models` module avoiding the problem entirely.

It is safe to import models in your `views` and `admin` modules ofcourse, but use only the canonical import, directly from models and not some sub package:

```
from bar.models import Foo
```

1.2.7 Haystack

As a general rule, never import models into modules that are collected by django's `importlib`. This includes other `models` modules but also some third party extensions like *django-haystack* use it (or something like it). Haystack automatically collects all `search_indexes` modules.

When you absolutely have to import a model in an automatically collected file, do it like this:

```
from django.db import get_model
Foo = get_model('bar', 'Foo')
```

Yes, that uses django's lazy model loading mechasism as well. It is much easier though to register you models for haystack inside the `models.py` module and not in the `search_indexes` module.

1.3 Translation of database content using gettext

When using the *i18n* and *l10n* features of easymode, you can use gettext's standard translation features to translate all the database content.

1.3.1 Automatic catalog management

If the *MASTER_SITE* and *AUTO_CATALOG* directive are set to True, every time a model decorated with `I18n` is saved, easymode wil add an entry to the corresponding gettext catalog [1]. (for all the options related to the location of the catalogs please refer to Easymode settings). The default for *AUTO_CATALOG* is `False`, the default for *MASTER_SITE* is also `False`.

For each language in your `LANGUAGES` directive, a catalog will be created. This way you can translate all the content using something like poedit or rosetta. This is especially convenient when a new site is created, for the first *big batch* of translations.

For modifications afterward, you can just use the admin interface, which will show the translations from the gettext catalog if they exist.

[1] It is possible to have more finegrained control over which models should be automatically added to the catalog by settings *AUTO_CATALOG* to `False` and using `easymode.i18n.register()` to register individual models. More info in the *AUTO_CATALOG* docs.

1.3.2 TAKE CARE

The translation mechanism using gettext is best used when a site is initially going to be translated to other languages. After this fase, content will most likely be edited directly in the admin interface, and you can have issues with translations not showing up. This can happen when content was allready stored in the database, as described in *Database is bigger than gettext*. In effect any changes made to the gettext catalog after editors are changing content in the admin interface has a very low probability of being shown on the website [2]. However, easymode will help you, showing you the origin of a value in the admin, by displaying symbols next to the input fields:

- If a value is from the gettext catalog or fallback, easymode will display °
- If a value is from the database, but the catalog has a different value, easymode will display . You can hover over this symbol to see the catalog value.
- If a value is from the database and there is no conflict with the catalog, easymode will display only

It takes proper planning to make full use of the gettext capabilities of easymode. The proper workflow is:

1. edit and add base content of the website, **ALL OF IT** and make sure you don't want to modify it anymore.
2. translate content using gettext, and **completely stop all editing, just lock up the site during translation!** [3] [4]
3. edit and modify all you like in the admin, all translations will be there. [5]

If you choose to deviate from this workflow be sure to understand all the next topics and learn how to use *easy_reset_language*.

1.3.3 Translation mechanism explained

It is important to realise, that allthough you can make translations using gettext, the catalog is not the only place where translations are stored. The *I18n decorator* not only registers a model for catalog management, it also modifies the model.

suppose we have a model as follows:

```
@I18n('bar')
class Foo(models.Model):
    bar = models.CharField(max_length=255)
    foobar = models.TextField()
```

Normally the database would look like this:

```
CREATE TABLE "foobar_foo" (
    "id" integer NOT NULL PRIMARY KEY,
    "bar" varchar(255) NOT NULL,
    "foobar" text NOT NULL
)
```

The *I18n decorator* modiefies the model, given we've got both 'en' and 'yx' in out `LANGUAGES` directive this is what the model would look like on the database end:

[2] Obviously, other gettext catalogs, generated from static content, that are not managed by easymode are unaffected.

[3] You can make sure nobody goes into the admin to edit things, by commenting out the admin routes in urls.py the new message id. Unless the content is allready saved in the database (*Database is bigger than gettext*).

[4] If you don't lock up the admin you might have issues with translations not showing up. Someone could for some reason save an item in the wrong language. This means the value in the database will be used instead of the translation in the catalig. You can detect when this has happened by looking for a sign in the admin next to the untranslated field. Hover over the field to see the value in the catalog.

[5] Watch out when you completely replace existing content in the *MSGID_LANGUAGE*. The *MSGID_LANGUAGE* is used for the message id's in the catalogs. When you completely replace the existing message id with something different, gettext will see that as adding a new message instead of changing an existing message. When this happens, translations can nolonger be associated with the new message and all languages will fall back to

```
CREATE TABLE "foobar_foo" (
    "id" integer NOT NULL PRIMARY KEY,
    "bar_en" varchar(255) NULL,
    "bar_yx" varchar(255) NULL,
    "foobar" text NOT NULL
)
```

On the model end you would not see this, because you will still access `bar` like this:

```
>>> m = Foo.objects.get(pk=1)
>>> m.bar = 'hello'
>>> print m.bar
hello
```

Any field that is internationalized using the *I18n decorator* will always return the field in the current languge, both on read and on write.

1.3.4 Database is bigger than gettext

Only when a field is empty (`None`) **in the database for the current language, the gettext catalog will be consulted for a translation**

This way, a model has exactly the same semantics as before, in that we can read and write to the property, the way we defined it in it's declaration. We still get the gettext goodies, which is nice when large ammounts of text must be translated.

If the gettext catalog would be the only place where the translations would be stored, having proper write semantics would become very difficult.

Example:

```
>>> from django.utils.translation import activate

>>> m = Foo()
>>> m.bar = 'hello'
>>> m.bar
'hello'
>>> activate('yx')
>>> m.bar
'hello'
>>> m.bar = 'xy says hello'
>>> m.bar
'xy says hello'
>>> activate('en')
>>> m.bar
'hello'
```

What you'll notice is that `m.bar` is allready available in the language 'yx' even though we did't specify it's value yet. This is because the normal behaviour of gettext is to return the `msgid` if the `msgstr` is not yet available. This is because the value for `m.bar` in language 'yx' was resolved as follows:

- see if the database value bar_yx is not null, if so return bar_yx
- see if the `msgstr` for 'hello' (The value of `m.bar` in the *MSGID_LANGUAGE*) exists if so return uget-text('hello')
- otherwise return the value in the *fallback language*

1.3.5 Importing translations is implicit

One thing that follows from the mechanics as described above, is that there is no need to explicitly import translations from gettext catalogs into the database.

Importing does take place however, each time a model is saved in the admin, the translations are written to the database.

This is because the translations from the gettext catalog *ARE* displayed in the admin, which means they *ARE* present in the form, but since the database column itself is *EMPTY* it will be marked as a change and written to the appropriate field.

This implicit import could pose a problem. If for example a model was edited in the admin, *BEFORE* the gettext catalog was properly translated and imported, it could be that the wrong value, from some *fallback language* got written to the database. Because the database get's precedence over the gettext catalog, the new translation would never show up.

This inconvenience can be resolved using the *easy_reset_language* command

1.4 Automatic generation of xml from models using xslt

Most of the data being transferred to a flash frontend is in xml. This is both because xml is very well supported by Flash (e4x) and because hierarchical data is easily mapped to xml. Most data used for flash sited is hierarchical in nature, because the display list -flash it's version of html's DOM- is hierarchical as well.

What easymode tries to do is give you a basic hierarchical xml document that mirrors your database model, which you can then transform using xslt [6].

1.4.1 Why Xslt?

Xslt is a functional programming language, specifically designed to transform one type of xml into another. So if we can reduce django's template rendering process to transforming one type of xml to another, xslt would be a dead on match for the job.

In fact we can. Easymode comes with a special xml serializer. This serializer differs from the normal django serializers, in that it treats a foreign key relation as a child parent relation. So while django's standard serializers output is flat xml, easymode's serializer outputs hierarchical xml.

1.4.2 Relations must be organized as a DAG

In order for easymode to be able to do it's work, the model tree should be organised as a DAG. if you accidently created a cycle (using `ManyToManyField` relations), easymode will let you know and throw an exception. Any `ManyToManyField` that is related to "self" will be ignored by the serializer.

Most of the time you don't really need the cyclic relation at all. You just need to do some preprocessing of the data. You can render a piece of xml yourself, without using easymode's serializers and pass it to the xslt, see Injecting extra data into the XSLT.

1.4.3 Getting xml from a model

There are several ways to obtain such a hierarchical xml tree from a django model. The first is by decorating a model with the `toxml()` decorator:

[6] Xslt requires a python xslt package to be installed. Easymode can work with lxml , libxslt

```
from easymode.tree.xml.decorators import toxml

@toxml
class Foo(models.ModelAdmin):
    title = models.CharField(max_length=255)
    content = TextField()

class Bar(models.ModelAdmin):
    # use lazy foreign keys!
    # Even in the same models module!
    foo = models.ForeignKey('Foo', related_name=bars)

    label = models.CharField(233)
```

The `Foo` model has now gained a `__xml__` method on both itself as on the queryset it produces. Calling it will produce hierarchical xml, where all inverse *ForeignKey* [7] relations are followed (easymode's serializer follows the managers on a related model).

The preferred method for calling the `__xml__` method is by it's function:

```
from easymode.tree.xml import xml

foos = Foo.objects.all()
rawxml = xml(foos)
```

1.4.4 Getting xml from several queries

The next option, which can also be used with multiple queries, is use the `XmlQuerySetChain`

```
from easymode.tree.xml import xml
from easymode.tree.xml.query import XmlQuerySetChain

foos = Foo.objects.all()
qsc = XmlQuerySetChain(foos)
rawxml = xml(qsc)
```

Normally you would use the `XmlQuerySetChain` to group some `QuerySet` objects together into a single xml:

```
from easymode.tree.xml import xml
from easymode.tree.xml.query import XmlQuerySetChain

foos = Foo.objects.all()
hads = Had.objects.all()

qsc = XmlQuerySetChain(foos, hads)
rawxml = xml(qsc)
```

1.4.5 Using xslt to transform the xml tree

Now you know how to get the xml as a tree from the models, it is time to show how xslt can be used to transform this tree into something a flash developer can use for his application.

[7] While `ForeignKey` relations are followed 'inverse' by the managers on the related model, this is not the case for `ManyToManyField`. Instead they are followed 'straight'.

Easymode does not follow 'straight' foreigkey relations because that would cause a cycle, instead it only takes the value of the foreignkey, which is an integer. If you do need some data from the related object in your xml, you can define the natural_key method on the related model. The output of that method will become the value of the foreignkey, instead of an integer. This way you can include data from a 'straight' related model, without introducing cyclic relations.

Easymode comes with one xslt template [8] that can give good results, depending on your needs:

```
from easymode.xslt.response import render_to_response

foos = foobar_models.Foo.objects.all()
return render_to_response('xslt/model-to-xml.xsl', foos)
```

The `render_to_response()` helper function will take an xslt as a template and a `XmlQuerySetChain` or a model/queryset decorated with `toxml()` to produce it's output. Additionally you can pass it a `dict` containing xslt parameters. You have to make sure to use `prepare_string_param()` on any xslt parameter that should be passed to the xslt processor as a string.

Other helpers can be found in the `easymode.xslt.response` module.

1.4.6 When the standard serializer is not enough

It could be that the raw values are not what you want from your model. For example, it could be you wanted a `datetime.datetime` nicely formatted or maybe you want to have the output of `get_absolute_url` in your xml.

Fortunately you can do that by implementing your own serialization function in your model:

```
@I18n('value')
class TagModel(models.Model):

    value = models.CharField(max_length=23)

    def __serialize__(self, stream):
        stream.startElement('taggy', {})
        stream.characters(self.value)
        stream.endElement('taggy')
```

The `__serialize__` method will be called by the serializer instead of the regular handler. You get the xml stream as an argument and you've got to write to it by calling the methods on it. see `xml.sax.saxutils.XMLGenerator` for the correct api.

The same is `True` when you've got custom fields that should do something with their values before you can use it (like storing data pickled). Implementing a `__serialize__` method in you custom field will make the serializer use your implementation (for example `SafeHTMLField` implements it's own `__serialize__` method to validate the xml before writing it to the serializer stream).

1.4.7 Exclude certain relations from being followed by the serializer

Sometimes you want to have a simple foreign key relation, but you don't need the data of the related object in your xml. As a matter of fact, it would degrade the performance of your view. You could mark the `ForeignKey` as `serialize=False` but that would make the field simply disappear from the xml. You might only want to exclude it from being rendered as a child of the parent object, but on the object itself you want the foreign key rendered as usual.

You can mark any foreign key with a `nofollow` attribute and it will not be followed, when the parent is serialized:

```
class Bar(models.ModelAdmin):
    foo = models.ForeignKey('Foo', related_name=bars)
    foo.nofollow = True

    label = models.CharField(233)
```

[8] The default xslt, with template path: *'xslt/model-to-xml.xsl'* comes with easymode and looks like this:

The foreign key will still be serialized as it always would, but the related object will not be expanded on the parent. This means if I query `Foo`:

```
xml(Foo.objects.all())
```

I will not see any `Bar` objects as children of foo. But when querying `Bar`:

```
xml(Bar.objects.all())
```

You will still see that the foreign key is included in the xml.

```
<?xml version="1.0"?>
<!--
  model-to-xml.xsl

  Created by Lars van de kerkhof on 2009-07-30.
-->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output encoding="UTF-8" indent="yes" method="xml" />

  <!-- Render an object node -->
  <xsl:template match="object">
    <xsl:element name="{substring-after(@model,'.')}">
      <xsl:apply-templates select="field"/>
      <xsl:if test="object">
        <children>
          <xsl:apply-templates select="object"/>
        </children>
      </xsl:if>

    </xsl:element>
  </xsl:template>

  <!-- render a field node -->
  <xsl:template match="field">
    <xsl:if test="@type">
      <xsl:element name="{@name}">
        <xsl:copy-of select="@type"/>
        <xsl:copy-of select="@font"/>
        <xsl:value-of select="."/>
      </xsl:element>
    </xsl:if>
  </xsl:template>

  <!--
    ManyToManyField is only shown if not empty.
    ForeignKey is only shown if natural_key is defined on
    the related object.

    see http://packages.python.org/django-easymode/xslt/index.html#id2
  -->
  <xsl:template match="field[@to]">
    <xsl:if test="natural or object">
      <xsl:element name="{@name}">
        <xsl:copy-of select="@rel"/>
        <xsl:choose>
          <xsl:when test="count(natural) > 1">
```

```
            <xsl:for-each select="natural">
              <property><xsl:value-of select="."/></property>
            </xsl:for-each>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="natural"/>
          </xsl:otherwise>
        </xsl:choose>
        <xsl:if test="object">
          <xsl:apply-templates select="object"/>
        </xsl:if>
      </xsl:element>
    </xsl:if>
  </xsl:template>

  <!-- just copy unmatched nodes -->
  <!-- so we know something is wrong -->
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

  <!-- Parse the root node of the serialized xml -->
  <xsl:template match="django-objects">
    <root>
      <xsl:apply-templates/>
    </root>
  </xsl:template>

</xsl:stylesheet>
```

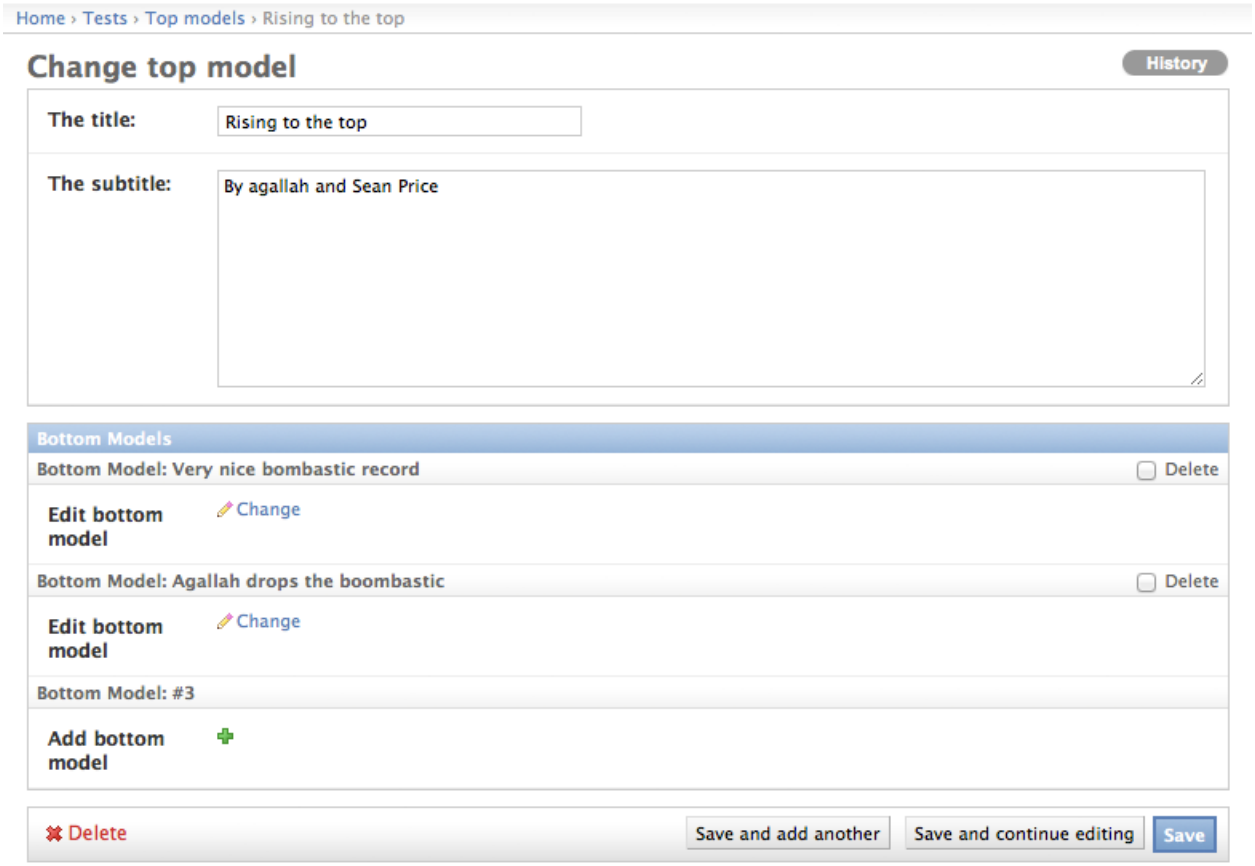
1.5 Admin support for model trees with more than 2 levels of related items

Note: This is the documentation for the **new** tree, which is much more flexible and powerful as the old one. for the old docs, see oldtree_explanation

Easymode has full admin support. Since content easymode was designed to handle is heavy hierarchic, easymode can also support this in the admin.

The single most annoying problem you will encounter when building django apps, is that after you discovered the niceties of `inlines`, you find out that only 1 level of `inlines` is supported. It does not support any form of recursion.

Easymode gives you the `easymode.tree.admin` package which gives you two baseclasses you can use to make recursive inlines possible. Django let's you define both regular `ModelAdmin` as well `StackedInline` or `TabularInline` for your models. But it's less known that you can use them *at the same time*, for the same model. That is exactly what we're going to do to implement a nice admin tree. The result will look like this:



Clicking the *Add bottom model* button will open a popup, just like with foreign keys, that can be used to add a new item to the list. Clicking the *Change* link will move to the edit view of the related item. This way you can nest as deep as you want.

1.5.1 Implementing the tree

As said to implement a tree you've got to define both an inline model admin as a regular modeladmin for the model you want to inline:

```
class BottomAdmin(InvisibleModelAdmin):
    """
    I am using InvisibleModelAdmin as a base class here so I can get the
    parent_link functionality and also that BottomAdmin is not visible in
    the admin listing (see old tree for more info). I could've used
    LinkedItemAdmin as well, if I would've been interested in parent_link
    only. This is the change view for the inlined item.
    """
    parent_link = 'top'


class BottomLinkInline(LinkInline):
    """
    This is the inline view of the inlined item. It will de rendered as a
    link to the change view or add view of the inlined item.

    NOTE that you MUST define fields, which must only include the foreign key.
    Ofcourse you might want to include some more fields and put them as
```

```
    read_only_fields, to give a bit more info.
    """
    fields = ('top',)
    model = BottomModel
```

Also the model that holds the inlines, needs to extend a special admin class:

```
class TopAdmin(LinkedItemAdmin):
    """
    This is the top view that has the inlines. It has a special form and
    template to make it display the foreign key fields as links.
    """
    inlines = [BottomLinkInline]
```

That is all the code to implement one level of the tree. Note that because the inlined items are **real** inlines, you can do all the funky stuff you used to do with them like drag and drop reorder and such.

See `easymode.tree.admin.abstract.LinkInline`, `easymode.tree.admin.abstract.LinkedItemAdmin` and `InvisibleModelAdmin` for more info.

1.6 Easymode settings

1.6.1 AUTO_CATALOG

Easymode can manage a gettext catalog with your database content for you. If `AUTO_CATALOG` is `True`, easymode will add every new object of a model decorated with `I18n` to the gettext catalog. The default is `False`.

How does gettext work

When existing content is updated in the *MSGID_LANGUAGE* on the *MASTER_SITE*, gettext will try to updated the msgid's in all the languages. Therefor keeping the mapping between original and translation. There is a limit on the ammount of change, before gettext can nolonger identify a string as a change in an existing msgid. For example:

```
# in the english django.po:
#: main.GalleryItem.title_text:32
msgid "I've got a car"
msgstr ""
```

```
# in the french django.po:
#: main.GalleryItem.title_text:32
msgid "I've got a car"
msgstr "J'ai une voiture"
```

Now we update the main.GalleryItem.title_text in the db in english, which will also change the english gettext catalog's message id:

```
# in the english django.po:
#: main.GalleryItem.title_text:32
msgid "I've had a car"
msgstr ""
```

gettext will now also update the message id in french so the link between original and translation is kept.

```
# in the french django.po:
#: main.GalleryItem.title_text:32
```

```
msgid "I've had a car"
msgstr "J'ai une voiture"
```

The location of the catalog can be controlled using *LOCALE_DIR* and *LOCALE_POSTFIX*,

What does AUTO_CATALOG do?

example:

```
AUTO_CATALOG = False
```

With the above settings, no catalogs are managed automatically by easymode. You have to manually generate them using *easy_locale*.

AUTO_CATALOG can also be used when you only need *some* (but not all) of the internationalised models to auto update the catalog. For this to work you need to set *AUTO_CATALOG* to `False` in settings.py:

```
AUTO_CATALOG = False
```

Then somewhere else, for example in your admin.py or models.py you can turn on automatic catalog updates for specific models:

```
from models import News
import easymode.i18n

easymode.i18n.register(News)
```

Now only the `News` model will automatically update the catalog, but other models will leave it alone. See `easymode.i18n.register()` for more info.

Ofcourse, for this to work you must have *MASTER_SITE* set to `True`.

In a nutshell, `MASTER_SITE=False` will disable all gettext updating, while `AUTO_CATALOG=False`, still allows you to turn it on for selected models.

1.6.2 MASTER_SITE

The `MASTER_SITE` directive must be set to `True` if a gettext catalog should be automatically populated when new contents are created. This way all contents can be translated using gettext. You can also populate the catalogs manually using the *easy_locale* command.

In a multiple site context, you might not want to have all sites updating the catalog. Because the content created on some of these sites might not need to be translated because it is not used on any other sites. Content can flow from 'master site' to 'slave site' but not from 'slave site' to 'slave site'.

for more fine grained control over which models should be automatically added to a gettext catalog, see *AUTO_CATALOG*.

example:

```
MASTER_SITE = True
```

1.6.3 MSGID_LANGUAGE

The `MSGID_LANGUAGE` is the language used for the message id's in the gettext catalogs. Only when a content was created in this language, it will be added to the gettext catalog. If `MSGID_LANGUAGE` is not defined, the `LANGUAGE_CODE` will be used instead. The msgid's in the gettext catalogs should be the same for all languages.

This setting should be used when there are different sites, each with a different LANGUAGE_CODE set. These sites can all share the same catalogs.

example:

```
MSGID_LANGUAGE = 'en'
```

1.6.4 FALLBACK_LANGUAGES

The FALLBACK_LANGUAGES is a dictionary of values that looks like this:

```
FALLBACK_LANGUAGES = {
    'en': [],
    'hu': ['en'],
    'be': ['en'],
    'ff': ['hu','en']
}
```

Any string that is not translated in 'ff' will be taken from the 'hu' language. If the 'hu' also has no translation, finally it will be taken from 'en'.

1.6.5 LOCALE_DIR

Use the LOCALE_DIR setting if you want all contents to be collected in a single gettext catalog. If LOCALE_DIR is not specified, the contents will be grouped by app. When a model belongs to the 'foo' app, new contents will be added to the catalog located in foo/locale.

You might not want to have the dynamic contents written to your app's locale, if you also have static translations. You can separate the dynamic and static content by specifying the *LOCALE_POSTFIX*.

example:

```
PROJECT_DIR = os.dirname(__file__)
LOCALE_DIR = os.path.join(PROJECT_DIR, 'db_content')
LOCALE_PATHS = (join(LOCALE_DIR, 'locale'), )
```

(Note that by using LOCALE_PATHS the extra catalogs are loaded by django).

1.6.6 LOCALE_POSTFIX

The LOCALE_POSTFIX must be used like this:

```
LOCALE_POSTFIX = '_content'
```

Contents that belong to models defined in the 'foo' app, will be added to the catalog located at foo_content/locale instead of foo/locale.

1.6.7 USE_SHORT_LANGUAGE_CODES

Easymode has some utilities that help in having sites with multiple languages. LocaliseUrlsMiddleware and LocaleFromUrlMiddleWare help with adding and extracting the current language in the url eg:

http://example.com/**en**/page/1

When having many similar languages in a multi site context, you will have to use 5 letter language codes:

en-us en-gb

These language codes do not look pretty in an url:

http://example.com/en-us/page/1

and they might even be redundant because the country code is allready in the domain extension:

http://example.co.uk/en-gb/page/1

When `USE_SHORT_LANGUAGE_CODES` is set to `True`, the country codes are removed in urls, leaving only the language code. This means the url would say:

http://example.com/en/page/1

even when the current language would be 'en-us'.

THIS DIRECTIVE ONLY WORKS WHEN THERE IS NO AMBIGUITY IN YOUR `LANGUAGES` DIRECTIVE.

This means i can not have the same language defined twice in my `LANGUAGES`:

```
LANGUAGES = (
    ('en-us', _('American English')),
    ('en-gb', _('British')),
)
```

This will **NOT** work because both languages will be displayed in the url as 'en' which is ambiguous.

1.6.8 SKIPPED_TESTS

It might be that some tests fail because you've got some modules disabled or you can not comply to the test requirements. This is very annoying in a continuous integration environment. If you are sure that the failing tests cause no harm to your application, they can be disabled.

`SKIPPED_TESTS` is a sequence of test case names eg:

```
SKIPPED_TESTS = ('test_this_method_will_fail', 'test_this_boy_has_green_hair')
```

will make sure these 2 tests will not be executed when running the test suite.

1.6.9 RECURSION_LIMIT

When a model tree is not a dag, easymode can get into an infinite recursion when producing xml, resulting in a stack overflow. Because xml is produced using `xml.sax`, which is a c-extension, your app will simply crash and not raise any exceptions. Easymode will try to help you, by never allowing recursion to go deeper then `RECURSION_LIMIT`. The default is set to:

```
RECURSION_LIMIT = sys.getrecursionlimit() / 10
```

which usually means 100. Take care when increasing this value, because most of the time when the limit is reached it actually *IS* caused by cycles in your data model and not because of how many objects you've got in your database.

1.7 Management Commands

1.7.1 easy_locale

Note: Easy locale will update the gettext catalogs with content from the database. This can be specific to a single app or model.

1.7.2 easy_reset_language

Note: This command will clear the database fields in one language for a specific app or model, so the translation will once again come from the catalog, instead of the database.

1.8 Easyfilters

Easymode comes with 3 templatetags that can be used to modify existing templates so they can be used in a multilingual environment.

1.8.1 strip_locale()

`strip_locale()` will have an url as an argument and if there is a locale in the url, it will be stripped:

```
{% load 'easyfilters' %}

{{ 'http://example.com/en/greetings'|strip_locale }}
```

this will render as: `http://example.com/greetings` so the 'en' part will be removed from the url.

You can use this filter in combination with `LocaliseUrlsMiddleware`. The middleware will add the current language to any urls that do not have the language code in the url yet.

1.8.2 fix_locale_from_request()

Fixes the language code as follows:

If there is only one language used in the site, it strips the language code. If there are more languages used, it will make sure that the url has the current language as a prefix.

usage:

```
{% load 'easyfilters' %}

{{ 'http://example.com/en/greetings'|fix_locale_from_request:request.LANGUAGE_CODE }}
```

Suppose `request.LANGUAGE_CODE` was 'ru' then the output would become:

```
http://example.com/ru/greetings
```

Suppose `settings.LANGUAGES` contained only one language, the output would become:

```
http://example.com/greetings
```

You probably do not need this templatetag if you are using `LocaliseUrlsMiddleware`.

1.8.3 fix_shorthand()

Use this if you want to use *USE_SHORT_LANGUAGE_CODES*.

`fix_shorthand()` will always return the correct locale to use in an url, depending on your settings of *USE_SHORT_LANGUAGE_CODES*.

usage:

```
{% load 'easyfilters' %}

{{ request.lANGUAGE_CODE|fix_shorthand }}
```

Suppose `request.LANGUAGE_CODE` is 'fr-be' and *USE_SHORT_LANGUAGE_CODES* is set to `True`, the output would become:

```
fr
```

If *USE_SHORT_LANGUAGE_CODES* is set to `False` the output would be:

```
fr-be
```

If `request.LANGUAGE_CODE` is not a five letter language code, nothing happens.

1.9 easymode.middleware

1.9.1 Google Analytics

Easymode has middleware to support caching in combination with google analytics. Google analytics updates a session cookie on each request. Because django's `SessionMiddleware` places cookie in it's vary header, *you will save every single request to the cache* if you use it.

1.9.2 Internationalization related middleware

When using the internationalization middlewares, you should include *easymode.urls* in your url conf:

```
(r'^', include('easymode.urls')),
```

This will make sure that when you have defined *get_abolute_url* on your model, the *view on site* button will lead you to the page in the language you have currently selected.

1.10 Injecting extra data into the XSLT

If you want to have some extra data passed to the xslt, which can not be obtained by the serializer you can make some view helpers that create xml and pass it as a stringparam to the xslt.

Reasons why you would need this:

- You've got a model that has a foreign key to itself. You need this if you want some kind of hierarchical page tree or something. You might want to put the self referencing `ForeignKey` to `serialize=False`. This way it can not mess up the serializer, but you don't have a hierarchic structure in your xml.
- You pull data from an external source.
- You have to do some processing on the models before they get turned into xml.

- You have some data not coming from models that needs to be passed to the xslt.

In all these cases you can use *XmlPrinter* to make some well formed unicode safe xml you can feed to the xslt.

Here is an example where some static strings get passed to the xslt. These strings are translatable using django's regular i18n mechanism, but they are not in the database:

```
from django.utils.translation import ugettext as _

stringlib = dict(
    close_button = _('Close'),
    next_button = _('Next'),
    the_end = _("That's all folks")
)

def render_stringlib_xml():
    """Renders the stringlib xml"""
    stream = StringIO()
    xml = XmlPrinter(stream, settings.DEFAULT_CHARSET)
    xml.startElement('stringlib', {'id':'stringlib'})
    for (key, value) in stringlib.iteritems():
        xml.startElement(key, {})
        xml.characters(value)
        xml.endElement(key)
    xml.endElement('stringlib')

    byte_string = stream.getvalue()
    return byte_string.decode('utf-8')
```

Before you pass the rendered xml string, you should prepare it using `prepare_string_param()`:

```
from easymode.xslt import prepare_string_param as q
from easymode.xslt.response import render_to_response

params = {
    'stringlib' : q(render_stringlib_xml()),
}

qs = Foo.objects.all()

return render_to_response('xslt/model-to-xml.xsl', qs, params)
```

The best way to learn how easymode works, is to read the above topics in sequence and then look at the example_app. If you have questions please send them to the mailing list at easymode@librelist.com.

CHAPTER 2

Installation

You can download easymode from:

http://github.com/specialunderwear/django-easymode/downloads/

Or you can do:

- `pip install django-easymode`

Or:

- `pip install -e git://github.com/specialunderwear/django-easymode.git#egg=easymode`

Note the version number in the top left corner and use:

- `easy_install http://github.com/specialunderwear/django-easymode/tarball/[VERSION]`

Which, if the version was v0.1.0 would become http://github.com/specialunderwear/django-easymode/tarball/v0.1.0.
If you want to use easymode's xslt fascilities, make sure to install either lxml or libxslt.

CHAPTER 3

Requirements

Easymode requires python 2.6, furthermore the following packages must be installed:

- Django

The following packages might also be required, depending on what features you are using.

- lxml
- polib
- django-reversion

CHAPTER 4

Example

Easymode comes with an example app which is available from github:

http://github.com/specialunderwear/django-easymode/

To run the example app, you must clone the repository, install the dependencies and initialize the database:

```
git clone http://github.com/specialunderwear/django-easymode.git
cd django-easymode
pip install -r requirements.txt
cd example
python manage.py syncdb
python manage.py loaddata example_data.xml
python manage.py runserver
open http://127.0.0.1:8000/
```

CHAPTER 5

Unsupported django features

The following features, which django supports, are not supported by easymode:

- `unique_together`
- `unique_for_date`, `unique_for_month`, `unique_for_year`
- `django.contrib.admin.ModelAdmin.fields`, use `django.contrib.admin.ModelAdmin.fieldsets` instead.
- Inheritance for models is restricted to `abstract` base classes. This is a direct result of the fact that `OneToOneField` are *not* supported by the serializer.
- `django.contrib.admin.ModelAdmin.prepopulated_fields` is not supported.
- You can not use fields marked for translation with `I18n` in the `ordering` attribute of a model's Meta options.
- Unfortunately, the new template loaders are not supported with xslt templates. Please use the old, deprecated ones like
  - `django.template.loaders.app_directories.load_template_source()`
  - `django.template.loaders.filesystem.load_template_source()`

Most these features are not supported because the ammount of work to have them was greater than the benefit of having them. However, it could also be that I just didn't need it yet.

CHAPTER 6

Actionscript bindings

If you are developing flex or flash sites with easymode, you are invited to try out the new actionscript bindings at http://github.com/specialunderwear/robotlegs-dungdungdung

These integrate object creation and databinding for easymode's xml output.

CHAPTER 7

Api docs

7.1 easymode.urls

7.2 easymode.i18n

7.3 easymode.i18n.decorators

7.4 easymode.i18n.admin.decorators

7.5 easymode.tree

7.6 easymode.tree.xml.decorators

7.7 easymode.tree.xml.query

7.8 easymode.tree.admin.relation

7.9 easymode.tree.admin.abstract

7.10 easymode.tree.introspection

7.11 easymode.xslt

7.12 easymode.xslt.response

7.13 easymode.utils

Contains utils for:

Matching first item in a list that matches some predicate. Mutex XML parser/generator that handles unknown entities. Controlling recursion depth

easymode.utils.**recursion_depth**(**args*, ***kwds*)

A context manager used to guard recursion depth for some function. Multiple functions can be kept separately because it will be counted per key.

Any exceptions raise in the recursive function will reset the counter, because the stack will be unwinded.

usage:

```
with recursion_depth('some_function_name') as recursion_level:
    if recursion_level > getattr(settings, 'RECURSION_LIMIT', sys.getrecursionlimit() / 10):
        raise Exception("Too deep")

    # do some recursive dangerous things.
```

Parameters **key** – The key under which the recursion depth is kept.

easymode.utils.**first_match**(*predicate*, *lst*)

returns the first value of predicate applied to list, which does not return None

```
>>>
>>> def return_if_even(x):
...     if x % 2 is 0:
...         return x
...     return None
>>>
>>> first_match(return_if_even, [1, 3, 4, 7])
4
>>> first_match(return_if_even, [1, 3, 5, 7])
>>>
```

Parameters

- **predicate** – a function that returns None or a value.
- **list** – A list of items that can serve as input to `predicate`.

Return type whatever `predicate` returns instead of None. (or None).

class easymode.utils.**mutex**(*max_wait=None*, *lockfile=None*)

A semaphore Context Manager that uses a temporary file for locking. Only one thread or process can get a lock on the file at once.

it can be used to mark a block of code as being executed exclusively by some thread. see mutex.

usage:

```
from __future__ import with_statement
from easymode.utils import mutex

with mutex:
    print "hi only one thread will be executing this block of code at a time."
```

Mutex raises an *easymode.utils.SemaphoreException* when it has to wait to long to obtain a lock or when it can not determine how long it was waiting.

Parameters

- **max_wait** – The maximum amount of seconds the process should wait to obtain the semaphore.
- **lockfile** – The path and name of the pid file used to create the semaphore.

exception easymode.utils.**SemaphoreException**
An exception that get thrown when an error occurs in the mutex semphore context

easymode.utils.**bases_walker**(*cls*)
Loop through all bases of cls

```
>>> str = u'hai'
>>> for base in bases_walker(unicode):
...     isinstance(str, base)
True
True
```

Parameters **cls** – The class in which we want to loop through the base classes.

easymode.utils.**url_add_params**(*url*, ***kwargs*)
Add parameters to an url

```
>>> url_add_params('http://example.com/', a=1, b=3)
'http://example.com/?a=1&b=3'
>>> url_add_params('http://example.com/?c=8', a=1, b=3)
'http://example.com/?c=8&a=1&b=3'
>>> url_add_params('http://example.com/#/irock', a=1, b=3)
'http://example.com/?a=1&b=3#/irock'
>>> url_add_params('http://example.com/?id=10#/irock', a=1, b=3)
'http://example.com/?id=10&a=1&b=3#/irock'
```

7.14 easymode.utils.xmlutils

Contains classes and functions for dealing with html entities.

You need this as soon as you are doing anything with rich text fields and want to use easymode.tree.xml.decorators.toxml().

easymode.utils.xmlutils.**unescape_all**(*string*)
Resolve all html entities to their corresponding unicode character

class easymode.utils.xmlutils.**XmlScanner**(**args*, ***kwargs*)
This class is an xml parser that will not throw errors when unknown entities are found.

It can be used as follows:

```
parser = sax.make_parser(["easymode.utils.xmlutils"])
```

All entities that can not be resolved will be skipped. This will trigger skippedEntity on the contentHandler.

A good contenthandler for this scanner is *XmlPrinter* because it will turn these entities into their unicode characters.

class easymode.utils.xmlutils.**XmlPrinter**(*out=None*, *encoding='iso-8859-1'*)
XmlPrinter can be used as a contenthandler for the *XmlScanner*.

It will convert all skippedEntities to their unicode characters and copy these to the output stream.

You can use it as a simple xml generator, to create xml:

```
    stream = StringIO.StringIO()
    xml = XmlPrinter(stream, settings.DEFAULT_CHARSET)

    def _render_page(page):
        xml.startElement('title', {})
        xml.characters(page.title)
        xml.endElement('title')
        xml.startElement('template', {})
        xml.characters(page.template)
        xml.endElement('template')

    for child in Page.children.all():
        xml.startElement('page', {})
        _render_page(child)
        xml.endElement('page')
```

easymode.utils.xmlutils.**create_parser**(**args*, ***kwargs*)

Because this function is defined, you can create an *XmlScanner* like this:

```
    from xml import sax

    parser = sax.make_parser(["easymode.utils.xmlutils"])
```

7.15 easymode.utils.languagecode

7.16 easymode.utils.polibext

7.17 easymode.utils.standin

easymode.utils.standin.**standin_for**(*obj*, ***attrs*)

Returns an object that can be used as a standin for the original object.

The standin object will have extra attrs, which you can define passed as keyword arguments.

Use standin like this:

```
    >>> a = u'I am E.T.'
    >>> b = standin_for(a, origin='outerspace', package='easymode.utils.standin')
    >>> b
    u'I am E.T.'
    >>> b == a
    True
    >>> b.origin
    'outerspace'
    >>> b.package
    'easymode.utils.standin'
    >>> isinstance(b, unicode)
    True
    >>> type(b)
    <class 'easymode.utils.standin.unicodeStandInWithOriginAndPackageAttributes'>
    >>> import pickle
    >>> b_pickle = pickle.dumps(b, pickle.HIGHEST_PROTOCOL)
    >>> c = pickle.loads(b_pickle)
    >>> b == c
    True
```

```
>>> c
u'I am E.T.'
>>> c.origin
'outerspace'
>>> c.package
'easymode.utils.standin'
```

Some types are not supported by standin_for. This is because these types are often used in statements like:

```
a = True
if a is True:
    print 'hi'
```

The *is* keyword checks for equal memory adress, and in case of a standin that can never be *True*. Also, since it is impossible to extend `bool` and `NoneType` you can never get:

```
isinstance(standin, bool)
isinstance(standin, NoneType)
```

To work with anything else but the real thing. This is why `bool` and `NoneType` instances are returned unmodified:

```
>>> a = False
>>> b = standin_for(a, crazy=True)
>>> b
False
>>> b.crazy
Traceback (most recent call last):
    ...
AttributeError: 'bool' object has no attribute 'crazy'
```

Parameters

- **obj** – An instance of some class
- ****attrs** – Attributes that will be added to the standin for *obj*

Return type A new object that can be used where the original was used. However it has extra attributes.

7.18 easymode.utils.template

7.19 easymode.admin.utils

7.20 easymode.admin.models.fields

7.21 easymode.admin.forms.fields

7.22 easymode.debug

Tools for debugging python applications

easymode.debug.**stack_trace**(*depth=None*)
returns a print friendly stack trace at the current frame, without aborting the application.

Parameters **depth** – The depth of the stack trace. if omitted, the entire stack will be printed.

usage:

```
print stack_trace(10)
```

7.23 easymode.debug.middleware

7.24 Version naming convention

- Each update to the development status will increase the first digit. (eg beta or alpha or production ready)
- Each new feature will increase the second digit.
- Each bugfix or refactor will increase the last digit
- An update to a 'big' digit, resets the 'smaller' digits.

e

B

C

E

F

M

R

S

U

X