django-custard Documentation

Release 0.10

Lucio Asnaghi

June 15, 2015

Contents

Django Custard is a small reusable unobtrusive Django app that implements runtime custom fields that can be attached to any model on the fly: it's possible to create fields and set values for them from the code or manage them through the admin site, with the ability to display them even outside of the admin.

- Home page: https://github.com/kunitoki/django-custard
- Documentation: http://django-custard.readthedocs.org
- Pypi: https://pypi.python.org/pypi/django-custard
- Example app on Github: https://github.com/kunitoki/django-custard/example
- Changelog: Changelog.rst
- License: The MIT License (MIT)
- Supports: Django 1.6, 1.7, 1.8 - Python 2.7, 3.2, 3.3, 3.4

CHAPTER 1

Getting started

1. You can get Django Custard by using pip:

```
pip install django-custard
```

2. You will need to add the 'custard' application to the INSTALLED_APPS setting of your Django project settings.py file.:

```
INSTALLED_APPS = (
    ...

    'custard',
)
```

3. In a models.py file in your app you should just add the 2 models responsible of holding the custom fields type and values.:

```
from custard.builder import CustomFieldsBuilder

builder = CustomFieldsBuilder('myapp.CustomFieldsModel', 'myapp.CustomValuesModel')

...

class CustomFieldsModel(builder.create_fields()):
    pass

class CustomValuesModel(builder.create_values()):
    pass
```

4. Then sync your database with:

```
python manager.py syncdb
```

CHAPTER 2

Customization

2.1 Configuration

Customize Django Custard with some `CUSTOM_*` configuration variables in a Django project `settings.py` file.

2.1.1 Configuration

It's possible to customize Django Custard behaviour by adding `CUSTOM_*` configuration variables to a Django project `settings.py` file.

Default values are the ones specified in examples.

Full example

Configuration sample that can be used as a start:

```
# Django Custard configuration example

CUSTOM_CONTENT_TYPES = (
  'myapp.MyModelName',
  'myotherapp.MyOtherModelName',
  'auth.User',
  'auth.Group',
)

CUSTOM_FIELD_TYPES = {
  'text':     'django.forms.fields.CharField',
  'integer':  'django.forms.fields.IntegerField',
  'float':    'django.forms.fields.FloatField',
  'time':     'django.forms.fields.TimeField',
  'date':     'django.forms.fields.DateField',
  'datetime': 'django.forms.fields.DateTimeField',
  'boolean':  'django.forms.fields.BooleanField',
}

CUSTOM_WIDGET_TYPES = {
  'text':     'django.contrib.admin.widgets.AdminTextInputWidget',
  'integer':  'django.contrib.admin.widgets.AdminIntegerFieldWidget',
  'float':    'django.contrib.admin.widgets.AdminIntegerFieldWidget',
  'time':     'django.contrib.admin.widgets.AdminTimeWidget',
  'date':     'django.contrib.admin.widgets.AdminDateWidget',
```

```
  'datetime': 'django.contrib.admin.widgets.AdminSplitDateTime',
  'boolean':  'django.forms.widgets.CheckboxInput',
}
```

Constants

There are some constants defined in `custard.conf` that hold the field type names. These can't be changed:

```
CUSTOM_TYPE_TEXT     = 'text'
CUSTOM_TYPE_INTEGER  = 'integer'
CUSTOM_TYPE_FLOAT    = 'float'
CUSTOM_TYPE_TIME     = 'time'
CUSTOM_TYPE_DATE     = 'date'
CUSTOM_TYPE_DATETIME = 'datetime'
CUSTOM_TYPE_BOOLEAN  = 'boolean'
```

Parameters

CUSTOM_CONTENT_TYPES

Select which content types will have custom fields, the name is the `ContentType.app_label`.``ContentType.model``:

```
CUSTOM_CONTENT_TYPES = (
  'app_label.ModelName',
)
```

CUSTOM_FIELD_TYPES

It's possible to override which custom form fields are generated for each field type when an instance of `CustomFieldModelBaseForm` is constructed:

```
CUSTOM_FIELD_TYPES = {
  'text':     'app.forms.MySpecialCharField',
  'integer':  'app.forms.AnotherIntegerField',
}
```

CUSTOM_WIDGET_TYPES

It's possible to override which custom form fields widgets are generated for each field type when an instance of `CustomFieldModelBaseForm` is constructed:

```
CUSTOM_WIDGET_TYPES = {
  'time':     'app.forms.widgets.AdminTimeWidget',
  'date':     'app.forms.widgets.AdminDateWidget',
  'datetime': 'app.forms.widgets.AdminSplitDateTime',
}
```

2.2 Models

How to work with Django Custard models and helper classes.

2.2.1 Models

Rationale

Django Custard comes with a series of internally defined models and classes that tries to be as more unobtrusive as possible, so to make it possible any kind of extension and manipulation of its internal models and classes. This is possible through the `custard.builder.CustomFieldsBuilder` class:

```
from django.db import models
from custard.builder import CustomFieldsBuilder

builder = CustomFieldsBuilder('myapp.CustomFieldsModel', 'myapp.CustomValuesModel')

class CustomFieldsModel(builder.create_fields()):
    pass

class CustomValuesModel(builder.create_values()):
    pass
```

The `custard.builder.CustomFieldsBuilder` must know which classes are actually implementing the custom fields definitions and the custom fields values, so to `custard.builder.CustomFieldsBuilder.__init__` must be explicitly specified those models as strings with the full application label, much like when implementing `django.models.fields.ForeignKey` for externally defined models.

The Django Custard models that implement custom fields and values are explicitly declared as abstract, and not defined anywhere statically in the code. So it's possible to implement them in any project, and even have multiple instances of them, for example when it's needed to maintain custom fields separation inside big apps.

When an application makes use of a standard base model for all its models, like when subclassing from `django_extensions.db.models.TimeStampedModel`, Django Custard models can be constructed with a `base_model` class:

```
from django.db import models
from custard.builder import CustomFieldsBuilder
from django_extensions.db.models import TimeStampedModel

builder = CustomFieldsBuilder('myapp.CustomFieldsModel', 'myapp.CustomValuesModel')

class CustomFieldsModel(custom.create_fields(base_model=TimeStampedModel)):
    pass

class CustomValuesModel(custom.create_values(base_model=TimeStampedModel)):
    pass
```

Default for `base_model` is `django.db.models.Model`.

Mixin

Custom fields and values attach to an application *real* models. To ease the interaction with custom fields, it's possible to attach a special model `Mixin` to any model for which it is possible to attach custom fields, and gain a simplified interface to query and set fields and values:

```
from django.db import models
from custard.builder import CustomFieldsBuilder

builder = CustomFieldsBuilder('myapp.CustomFieldsModel', 'myapp.CustomValuesModel')
```

```
CustomMixin = builder.create_mixin()

class Example(models.Model, CustomMixin):
    name = models.CharField(max_length=255)

class CustomFieldsModel(builder.create_fields()):
    pass

class CustomValuesModel(builder.create_values()):
    pass
```

A number of methods are then added to your model:

get_custom_fields(self) Return a list of custom fields for this model

get_custom_field(self, field_name) Get a custom field object for this model

get_custom_value(self, field_name) Get a value for a specified custom field

set_custom_value(self, field_name, value) Set a value for a specified custom field

Also it's possible to access custom fields like any other fields thanks to the `Mixin__getattr__` implementation. Look at this example:

```
# First obtain the content type
example_content_type = ContentType.objects.get_for_model(Example)

# Create a fields for the content type
custom_field = CustomFieldsModel.objects.create(content_type=example_content_type,
                                                data_type=CUSTOM_TYPE_TEXT,
                                                name='a_text_field',
                                                label='My field',
                                                searchable=True)
custom_field.save()

# Create an model instance
obj = Example(name='hello')
obj.save()

# Set a custom field value
obj.set_custom_value('a_text_field', 'world')

# Get fields from the model instance
print(obj.name) # This is a normal field
print(obj.a_text_field) # This is a custom field
```

Manager

In order to be able to search custom fields flagged as `searchable` in models, it's possible to add a special manager for any model needs:

```
from django.db import models
from custard.builder import CustomFieldsBuilder

builder = CustomFieldsBuilder('myapp.CustomFieldsModel', 'myapp.CustomValuesModel')
CustomManager = builder.create_manager()

class Example(models.Model):
    name = models.CharField(max_length=255)
```

```
    objects = CustomManager()

class CustomFieldsModel(builder.create_fields()):
    pass

class CustomValuesModel(builder.create_values()):
    pass
```

Executing the `search` method in the model will then search Example instances that contains the search string in any searchable custom field defined for that model and returns a queryset much like doing a filter call:

```
qs = Example.custom.search('foobar')
```

By passing a specific Manager class as `base_manager` parameter, the custom manager will then inherit from that base class:

```
from django.db import models
from custard.builder import CustomFieldsBuilder

builder = CustomFieldsBuilder('myapp.CustomFieldsModel', 'myapp.CustomValuesModel')

class MyUberManager(models.Manager):
    def super_duper(self):
        return None

CustomManager = builder.create_manager(base_manager=MyUberManager)

class Example(models.Model):
    objects = CustomManager()

Example.objects.super_duper()
```

Warning: Be careful to always define a default_manager named `objects` for any Model. If for some reason you omit to do so, you likely will end up in runtime errors when you use any class in Django Custard.

Using the models

It's possible to create fields on the fly for any model and create:

```
from django.contrib.contenttypes.models import ContentType
from custard.conf import CUSTOM_TYPE_TEXT

from .models import Example, CustomFieldsModel, CustomValuesModel

# First obtain the content type
example_content_type = ContentType.objects.get_for_model(Example)

# Create a text custom field
custom_field = CustomFieldsModel.objects.create(content_type=example_content_type,
                                                data_type=CUSTOM_TYPE_TEXT,
                                                name='my_first_text_field',
                                                label='My field',
                                                searchable=False)
custom_field.save()

# Create a value for an instance of you model
```

```
custom_value = CustomValuesModel.objects.create(custom_field=custom_field,
                                                object_id=Example.objects.get(pk=1).pk,
                                                value="this is a custom value")
custom_value.save()
```

2.3 Admin

Process of integrating Django Custard in admin.

2.3.1 Admin

Editing fields

It's possible to edit fields for custom content types by registering a model admin for it:

```
from django.contrib import admin

from .models import CustomFieldsModel, CustomValuesModel

class CustomFieldsModelAdmin(admin.ModelAdmin):
    pass
admin.site.register(CustomFieldsModel, CustomFieldsModelAdmin)

# Same goes for editing values
class CustomValuesModelAdmin(admin.ModelAdmin):
    pass
admin.site.register(CustomValuesModel, CustomValuesModelAdmin)
```

The form subclass

In order to integrate Django Custard with an app admin site is to create the custom ModelForm using `builder.create_modelform`:

```
from django.contrib import admin

from .models import Example, CustomFieldsModel, CustomValuesModel, builder

class ExampleForm(builder.create_modelform()):
    custom_name = 'My Custom Fields'
    custom_description = 'Edit the Example custom fields here'
    custom_classes = ''

    class Meta:
        model = Example
```

It's possible to subclass the form and override 3 functions to specify even more the search for custom fields and values (for example when filtering with User or Group, so multiple custom fields can be enable for each User or Group independently):

`get_fields_for_content_type(self, content_type)` This function will return all fields defined for a specific content type

`search_value_for_field(self, field, content_type, object_id)` This function will return a custom value instance of the given field of a given content type object

create_value_for_field(self, field, object_id, value) This function will create a value of the given field of a given content type object

Here is an example:

```
class ExampleForm(builder.create_modelform()):
    class Meta:
        model = Example

    # Returns all fields for a content type
    def get_fields_for_content_type(self, content_type):
        return CustomFieldsModel.objects.filter(content_type=content_type)

    # Returns the value for this field of a content type object
    def search_value_for_field(self, field, content_type, object_id):
        return CustomValuesModel.objects.filter(custom_field=field,
                                                content_type=content_type,
                                                object_id=object_id)

    # Create a value for this field of a content type object
    def create_value_for_field(self, field, object_id, value):
        return CustomValuesModel(custom_field=field,
                                 object_id=object_id,
                                 value=value)
```

When using this form with `commit=False` you have to take care of calling `save_custom_fields` after you saved your model instance to save custom field values too, as they need an instance already in the database:

```
class ExampleForm(builder.create_modelform()):
    class Meta:
        model = Example

form = ExampleForm(request.POST, instance=example_object)
if form.is_valid():
    # Do not hit the database
    obj = form.save(commit=False)

    # Do your logic here...

    # Save the object to the database
    obj.save()

    # Save many to many as Django usual
    form.save_m2m()

    # Save the custom fields
    form.save_custom_fields()
```

ModelAdmin

In admin site, add this new form as the default for for a `ModelAdmin` of any model:

```
class ExampleAdmin(builder.create_modeladmin()):
    form = ExampleForm

admin.site.register(Example, ExampleAdmin)
```

Then the last step is to subclass a model `change_form.html` and use the Django Custard modified version:

templates/admin/myapp/example/change_form.html:

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

Then editing Example object custom fields is enabled in the admin site.

Searches in list_view

In order to enable search custom fields in admin in search_fields, only overriding ModelAdmin.get_search_results is needed:

```
class ExampleAdmin(builder.create_modeladmin()):
    form = ExampleForm

    def get_search_results(self, request, queryset, search_term):
        queryset, use_distinct = super(ExampleAdmin, self).get_search_results(request,
                                                                              queryset,
                                                                              search_term)
        queryset |= self.model.objects.search(search_term)
        return queryset, use_distinct

admin.site.register(Example, ExampleAdmin)
```

Note: This implies you have overridden your default objects manager in Example model with the manager that comes with Django Custard.

Then the last step is to subclass a model change_list.html and use the Django Custard modified version for search:

templates/admin/myapp/example/change_list.html:

```
{% extends "admin/change_list.html" %}
{% block search %}
  {% include "custard/admin/search_form.html" %}
{% endblock %}
```

CHAPTER 3

Support

- Github: Use django-custard github issues, if you have any problems using Django Custard.

CHAPTER 4

Changelog

Only important changes are mentioned below.

4.1 v0.10 (2015-06-15)

- Added Mixin.__getattr__ to access model custom fields like any other fields
- Fixed a problem when using commit=False in ModelForms
- Fixed a problem when custom_content_types is passed in CustomFieldsBuilder constructor (thanks to Kamil Wargula for the report and patch)
- Revert problem in mixin "get_custom_value" introduced with https://github.com/kunitoki/django-custard/commit/f3e171e6170d33a1ba6aba170e76549c55021ade

4.2 v0.9 (2015-03-12)

- Added ability to pass "value" in CustomValuesModel.objects.create
- Fixed https://github.com/kunitoki/django-custard/pull/2
- More tests and fixes

4.3 v0.8 (2014-10-08)

- Fixed https://github.com/kunitoki/django-custard/issues/1
- More tests
- More coverage

4.4 v0.7 (2014-07-29)

- Moved away from the static custard.models.custom class in favour of custard.builder
- Enable search of custom fields in admin without hacks
- Simplified a bit how to determine the classes in the builder

- Updated tests and example to the new builder methods

4.5 v0.6 (2014-07-27)

- Added support for python 2.6/2.7/3.2 and django 1.5/1.6
- Added test cases for forms and widgets
- Improved settings class to be cached at runtime while being able to work with override_settings in tests
- Fixed documentation links and sections not showing up

4.6 v0.5 (2014-07-23)

- Removed useless custom Exception classes not used anymore
- Added model mixin creation class for holding additional helper functions
- Fixed broken links in docs
- Added manager subclass to docs
- Added admin to docs

4.7 v0.4 (2014-07-23)

- Moved generation of FormField from model to Form
- Improved documentation

4.8 v0.3 (2014-07-22)

- Improved documentation

4.9 v0.2 (2014-07-20)

- Avoid using a global registry for registered apps in favour of a setting variable
- Improved documentation

4.10 v0.1 (2014-07-18)

- First stable version released