Django Categories Documentation

Release 1.6

CoreyOordt

Feb 26, 2022

Contents

Django Categories grew out of our need to provide a basic hierarchical taxonomy management system that multiple applications could use independently or in concert.

As a news site, our stories, photos, and other content get divided into "sections" and we wanted all the apps to use the same set of sections. As our needs grew, the Django Categories grew in the functionality it gave to category handling within web pages.

CHAPTER 1

New in 1.1

- Fixed a cosmetic bug in the Django 1.4 admin. Action checkboxes now only appear once.
- Template tags are refactored to allow easy use of any model derived from `CategoryBase`.
- Improved test suite.
- Improved some of the documentation.

CHAPTER 2

Contents

2.1 Installation

2.1.1 To use the Category model

1. Install django-categories:

```
pip install django-categories
```

2. Add `"categories"` and `"categories.editor"` to your `INSTALLED_APPS` list in your project's `settings.py` file.

```
INSTALLED_APPS = [
    # ...
    "categories",
    "categories.editor",
]
```

3. Run `./manage.py syncdb` (or `./manage.py migrate categories` if you are using South)

2.1.2 To only subclass CategoryBase

If you are going to create your own models using *`CategoryBase`*, (see *Creating Custom Categories*) you'll need a few different steps.

1. Install django-categories:

```
pip install django-categories
```

2. Add `"categories.editor"` to your `INSTALLED_APPS` list in your project's `settings.py` file.

```
INSTALLED_APPS = [
    # ...
    "categories.editor",
]
```

3. Create your own models.

2.2 Getting Started

You can use Django Categories in two ways:

1. As storage for one tree of categories, using the *Category* model:

```
Top Category 1
  Subcategory 1-1
    Subcategory 1-2
      subcategory 1-2-1
Top Category 2
  Subcategory 2-1
```

2. As the basis for several custom categories (see *Creating Custom Categories*), e.g. a **MusicGenre** model

```
MusicGenre 1
  MusicSubGenre 1-1
  MusicSubGenre 1-2
    MusicSubGenre 1-2-1
MusicGenre 2
  MusicSubGenre 2-1
```

and a **Subject** model

```
Subject 1
  Discipline 1-1
  Discipline 1-2
    SubDiscipline 1-2-1
Subject 2
  Discipline 2-1
```

2.2.1 Connecting your model with Django-Categories

There are two ways to add Category fields to an application. If you are in control of the code (it's your application) or you are willing to take control of the code (fork someone else's app) you can make a *Hard Coded Connection*.

For 3rd-party apps or even your own apps that you don't wish to add Django-Categories as a dependency, you can configure a *Lazy Connection*.

Hard Coded Connection

Hard coded connections are done in the exact same way you handle any other foreign key or many-to-many relations to a model.

```
from django.db import models

class MyModel(models.Model):
    name = models.CharField(max_length=100)
    category = models.ForeignKey('categories.Category')
```

Don't forget to add the field or fields to your `ModelAdmin` class as well.

Lazy Connection

Lazy connections are done through configuring Django Categories in the project's `settings.py` file. When the project starts up, the configured fields are dynamically added to the configured models and admin.

If you do this before you have created the database (before you ran `manage.py syncdb`), the fields will also be in the tables. If you do this after you have already created all the tables, you can run `manage.py add_category_fields` to create the fields (this requires Django South to be installed).

You add a many-to-one or many-to-many relationship with Django Categories using the *FK_REGISTRY* and *M2M_REGISTRY* settings respectively. For more information see *Registering Models*.

2.3 Using categories in templates

2.3.1 Getting all items within a category

The *Category* model automatically gets reverse relationships with all other models related to it.

This allows you access to the related objects from the template without altering any views. For example, if you only had `Entry` models related to *Category*, your `categories/category_detail.html` template could look like

```
{% extends 'categories/base.html' %}
{% block content %}
<h1>{{ category }}</h1>
{% if category.parent %}
    <h2>Go up to
        <a href="{{ category.parent.get_absolute_url }}">
            {{ category.parent }}
        </a></h2>
{% endif %}
{% if category.description %}<p>{{ category.description }}</p>{% endif %}
{% if category.children.count %}
    <h2>Subcategories</h2>
    <ul>
        {% for child in category.children.all %}
        <li><a href="{{ child.get_absolute_url }}">{{ child }}</a></li>
        {% endfor %}
    </ul>
{% endif %}
<h2>Entries</h2>
{% if category.entries_set.all %}
    {% for entry in category.entries_set.all %}
        <p><a href="{{ entry.get_absolute_url }}">{{ entry.headline }}</a></p>
    {% endfor %}
{% else %}
```

(continues on next page)

(continued from previous page)

```
    <p><em>No entries for {{ category }}</em></p>
{% endif %}

{% endblock %}
```

If you have `related_name` parameters to the configuration (see *Registering Models*), then you would use `category.related_name.all` instead of `category.relatedmodel_set.all`.

2.3.2 Template Tags

To use the template tags:

```
{% import category_tags %}
```

tree_info

Given a list of categories, iterates over the list, generating two-tuples of the current tree item and a `dict` containing information about the tree structure around the item, with the following keys:

'new_level' `True` if the current item is the start of a new level in the tree, `False` otherwise.

'closed_levels' A list of levels which end after the current item. This will be an empty list if the next item's level is the same as or greater than the level of the current item.

An optional argument can be provided to specify extra details about the structure which should appear in the `dict`. This should be a comma-separated list of feature names. The valid feature names are:

ancestors Adds a list of unicode representations of the ancestors of the current node, in descending order (root node first, immediate parent last), under the key `'ancestors'`.

For example: given the sample tree below, the contents of the list which would be available under the `'ancestors'` key are given on the right:

```
Books                   ->  []
   Sci-fi               ->  [u'Books']
      Dystopian Futures ->  [u'Books', u'Sci-fi']
```

2.4 Registering Models

2.4.1 Registering models in settings.py

It is nice to not have to modify the code of applications to add a relation to categories. You can therefore do all the registering in `settings.py`. For example:

```
CATEGORIES_SETTINGS = {
    'FK_REGISTRY': {
        'app.AModel': 'category',
        'app.MyModel': (
            'primary_category',
            {'name': 'secondary_category', 'related_name': 'mymodel_sec_cat'}, )
    },
    'M2M_REGISTRY': {
```

(continues on next page)

(continued from previous page)

```
        'app.BModel': 'categories',
        'app.MyModel': ('other_categories', 'more_categories', ),
    }
}
```

The `FK_REGISTRY` is a dictionary whose keys are the model to which to add the new field(s). The value is a string or tuple of strings or dictionaries specifying the necessary information for each field.

The `M2M_REGISTRY` is a dictionary whose keys are the model to which to add the new field(s). The value is a string or tuple of strings specifying the necessary information for each field.

Registering one Category field to model

The simplest way is to specify the name of the field, such as:

```
CATEGORIES_SETTINGS = {
    'FK_REGISTRY': {
        'app.AModel': 'category'
    }
}
```

This is equivalent to adding the following the `AModel` of `app`:

```
category = models.ForeignKey(Category)
```

If you want more control over the new field, you can use a dictionary and pass other `ForeignKey` options. The `name` key is required:

```
CATEGORIES_SETTINGS = {
    'FK_REGISTRY': {
        'app.AModel': {'name': 'category', 'related_name': 'amodel_cats'}
    }
}
```

This is equivalent to adding the following the `AModel` of `app`:

```
category = models.ForeignKey(Category, related_name='amodel_cats')
```

Registering two or more Category fields to a Model

When you want more than one relation to `Category`, all but one of the fields must specify a `related_name` to avoid conflicts, like so:

```
CATEGORIES_SETTINGS = {
    'FK_REGISTRY': {
        'app.MyModel': (
            'primary_category',
            {'name': 'secondary_category', 'related_name': 'mymodel_sec_cat'}, )
    },
```

Registering one or more Many-to-Many Category fields to a Model

```
CATEGORIES_SETTINGS = {
    'M2M_REGISTRY': {
        'app.AModel': 'categories',
        'app.MyModel': (
            {'name': 'other_categories', 'related_name': 'other_cats'},
            {'name': 'more_categories', 'related_name': 'more_cats'},
        ),
    }
}
```

2.4.2 Registering a many-to-one relationship

To create a many-to-one relationship (foreign key) between a model and Django Categories, you register your model with the `register_fk` function.

register_fk(*model, field_name='category', extra_params={}*)

Parameters

- **model** – The Django Model to link to Django Categories
- **field_name** – Optional name for the field **default:** category
- **extra_params** – Optional dictionary of extra parameters passed to the `ForeignKey` class.

Example, in your `models.py`:

```
import categories
categories.register_fk(MyModel)
```

If you want more than one field on a model you have to have some extra arguments:

```
import categories
categories.register_fk(MyModel, 'primary_category')
categories.register_fk(MyModel, 'secondary_category', {'related_name':'mymodel_sec_cat
↪'})
```

The `extra_args` allows you to specify the related_name of one of the fields so it doesn't clash.

2.4.3 Registering a many-to-many relationship

To create a many-to-many relationship between a model and Django Categories, you register your model with the `register_m2m` function.

register_m2m(*model, field_name='categories', extra_params={}*)

Parameters

- **model** – The Django Model to link to Django Categories
- **field_name** – Optional name for the field **default:** categories
- **extra_params** – Optional dictionary of extra parameters passed to the `ManyToManyField` class.

Example, in your `models.py`:

```
import categories
categories.register_m2m(MyModel)
```

2.5 Adding the fields to the database

While Django will create the appropriate columns and tables if you configure Django Categories first, many times that is not possible. You could also reset the table, but you would loose all data in it. There is another way.

2.5.1 Enter South

South is a Django application for managing database schema changes. South's default behavior is for managing permanent changes to a model's structure. In the case of dynamically adding a field or fields to a model, this doesn't work because you are not making the change permanent. And probably don't want to.

Django Categories has a management command to create any missing fields. It requires South because it uses the South's API for making changes to databases. The management command is simple: `python manage.py add_category_fields [app]`. If you do not include an app name, it will attempt to do all applications configured.

Running this command several times will not hurt any data or cause any errors.

2.5.2 Reconfiguring Fields

You can make changes to the field configurations as long as they do not change the underlying database structure. For example, adding a `related_name` (see *Registering a many-to-one relationship*) because it only affects Django code. Changing the name of the field, however, is a different matter.

Django Categories provides a complementary management command to drop a field from the database (the field must still be in the configuration to do so): `python manage.py drop_category_field app_name model_name field_name`

2.6 Adding the fields to the Admin

By default, Django Categories adds the fields you configure to the model's Admin class. If your ModelAdmin class does not use the `fieldsets` attribute, the configured category fields are simply appended to the bottom the fields. If your ModelAdmin uses the `fieldsets` attribute, a new fieldset named `Categories`, containing all the configured fields is appended to the fieldsets. You can override or alter this behavior with the *ADMIN_FIELDSETS* setting.

ADMIN_FIELDSETS allows you to:

- Prevent Django Categories from adding the fields or fieldsets to a model's ModelAdmin class.
- Change the name of the fieldset (from the default: "Categories")
- Change the placement of the fieldset (from the default: bottom)

2.6.1 Preventing fields in the admin class

If you don't want Django Categories to add any fields to the admin class, simply use the following format:

```
CATEGORIES_SETTINGS = {
    'ADMIN_FIELDSETS': [
        'app.model': None,
    ]
}
```

2.6.2 Changing the name of the field set

To rename the field set, use the following format:

```
CATEGORIES_SETTINGS = {
    'ADMIN_FIELDSETS': [
        'app.model': 'Taxonomy',
    ]
}
```

2.6.3 Putting the field set exactly where you want it

For complete control over the field set, use the following format:

```
CATEGORIES_SETTINGS = {
    'ADMIN_FIELDSETS': [
        'app.model': {
            'name': 'Categories',
            'index': 0
        },
    ]
}
```

2.7 Creating Custom Categories

Django Categories isn't just for using a single category model. It allows you to create your own custom category-like models with as little or much customization as you need.

2.7.1 Name only

For many cases, you want a simple user-managed lookup table. You can do this with just a little bit of code. The resulting model will include name, slug and active fields and a hierarchical admin.

1. Create a model that subclasses *CategoryBase*

```
1  from categories.models import CategoryBase
2
3
4  class SimpleCategory(CategoryBase):
5      """
6      A simple of catgorizing example
7      """
8
9      class Meta:
10         verbose_name_plural = 'simple categories'
```

2. Create a subclass of CategoryBaseAdmin.

```
from django.contrib import admin

from categories.admin import CategoryBaseAdmin

from .models import SimpleCategory


class SimpleCategoryAdmin(CategoryBaseAdmin):
    pass


admin.site.register(SimpleCategory, SimpleCategoryAdmin)
```

3. Register your model and custom model admin class.

2.7.2 Name and other data

Sometimes you need more functionality, such as extra metadata and custom functions. The *Category* model in this package does this.

1. Create a model that subclasses *CategoryBase* as above.
2. Add new fields to the model. The *Category* model adds these extra fields.

```
from categories import models, settings
from categories.base import CategoryBase


class Category(CategoryBase):
    thumbnail = models.FileField(
        upload_to=settings.THUMBNAIL_UPLOAD_PATH,
        null=True, blank=True,
        storage=settings.THUMBNAIL_STORAGE,)
    thumbnail_width = models.IntegerField(blank=True, null=True)
    thumbnail_height = models.IntegerField(blank=True, null=True)
    order = models.IntegerField(default=0)
    alternate_title = models.CharField(
        blank=True,
        default="",
        max_length=100,
        help_text="An alternative title to use on pages with this category.")
    alternate_url = models.CharField(
        blank=True,
        max_length=200,
        help_text="An alternative URL to use instead of the one derived from "
                  "the category hierarchy.")
    description = models.TextField(blank=True, null=True)
    meta_keywords = models.CharField(
        blank=True,
        default="",
        max_length=255,
        help_text="Comma-separated keywords for search engines.")
    meta_extra = models.TextField(
        blank=True,
        default="",
```

(continues on next page)

(continued from previous page)

```
        help_text="(Advanced) Any additional HTML to be placed verbatim "
                  "in the &lt;head&gt;")
```

3. Add new methods to the model. For example, the *Category* model adds several new methods, including overriding the `save()` method.

```
from categories.models import Category


def save(self, *args, **kwargs):
    if self.thumbnail:
        from django.core.files.images import get_image_dimensions
        import django
        if django.VERSION[1] < 2:
            width, height = get_image_dimensions(self.thumbnail.file)
        else:
            width, height = get_image_dimensions(self.thumbnail.file, close=True)
    else:
        width, height = None, None

    self.thumbnail_width = width
    self.thumbnail_height = height

    super(Category, self).save(*args, **kwargs)
```

4. Alter `Meta` or `MPTTMeta` class. Either of these inner classes can be overridden, however your `Meta` class should inherit `CategoryBase.Meta`. Options for `Meta` are in the Django-MPTT docs.

```
from categories.base import CategoryBase


class Meta(CategoryBase.Meta):
    verbose_name_plural = 'categories'


class MPTTMeta:
    order_insertion_by = ('order', 'name')
```

5. For the admin, you must create a form that subclasses `CategoryBaseAdminForm` and at least sets the `Meta.model` attribute. You can also alter the form fields and cleaning methods, as *Category* does.

```
from categories.base import CategoryBaseAdminForm
from categories.models import Category


class CategoryAdminForm(CategoryBaseAdminForm):
    class Meta:
        model = Category

    def clean_alternate_title(self):
        if self.instance is None or not self.cleaned_data['alternate_title']:
            return self.cleaned_data['name']
        else:
            return self.cleaned_data['alternate_title']
```

6. Next you must subclass `CategoryBaseAdmin` and assign the `form` attribute the form class created above. You can alter any other attributes as necessary.

```
from categories.admin import CategoryAdminForm
from categories.base import CategoryBaseAdmin


class CategoryAdmin(CategoryBaseAdmin):
    form = CategoryAdminForm
    list_display = ('name', 'alternate_title', 'active')
    fieldsets = (
        (None, {
            'fields': ('parent', 'name', 'thumbnail', 'active')
        }),
        ('Meta Data', {
            'fields': ('alternate_title', 'alternate_url', 'description',
                       'meta_keywords', 'meta_extra'),
            'classes': ('collapse',),
        }),
        ('Advanced', {
            'fields': ('order', 'slug'),
            'classes': ('collapse',),
        }),
    )
```

2.8 Reference

2.8.1 Management Commands

import_categories

Usage: `./manage.py import_categories /path/to/file.txt [/path/to/file2.txt]`

Imports category tree(s) from a file. Sub categories must be indented by the same multiple of spaces or tabs. The first line in the file cannot start with a space or tab and you can't mix spaces and tabs.

add_category_fields

Usage: `./manage.py add_category_fields [app1 app2 ...]`

Add missing registered category fields to the database table of a specified application or all registered applications.

Requires Django South.

drop_category_field

Usage: `./manage.py drop_category_field app_name model_name field_name`

Drop the `field_name` field from the `app_name_model_name` table, if the field is currently registered in `CATEGORIES_SETTINGS`.

Requires Django South.

2.8.2 Models

CategoryBase

class CategoryBase

parent
`TreeForeignKey (self)`

The category's parent category. Leave this blank for an root category.

name
Required `CharField(100)`

The name of the category.

slug
Required `SlugField`

URL-friendly title. It is automatically generated from the title.

active
Required `BooleanField` *default:* `True`

Is this item active. If it is inactive, all children are set to inactive as well.

objects
`CategoryManager`

An object manager that adds an `active` method for only selecting items whose `active` attribute is `True`.

tree
`TreeManager`

A Django-MPTT TreeManager instance.

Category

class Category
Category is a subclass of *CategoryBase* and includes all its attributes.

thumbnail
`FileField`

An optional thumbnail, that is uploaded to *REGISTER_ADMIN* via *THUMBNAIL_STORAGE*.

Note: Why isn't this an `ImageField`?

For `ImageFields`, Django checks the file system for the existance of the files to handle the height and width. In many cases this can lead to problems and impact performance.

For these reasons, a `FileField` that manually manages the width and height was chosen.

---

thumbnail_width
`IntegerField`

The thumbnail width. Automatically set on save if a thumbnail is uploaded.

thumbnail_height
IntegerField

The thumbnail height. Automatically set on save if a thumbnail is uploaded.

order
Required IntegerField *default:* 0

A manually-managed order of this category in the listing. Items with the same order are sorted alphabetically.

alternate_title
CharField(100)

An alternative title to use on pages with this category.

alternate_url
CharField(200)

An alternative URL to use instead of the one derived from the category hierarchy.

Note: Why isn't this a URLField?

For URLFields, Django checks that the URL includes http:// and the site name. This makes it impossible to use relative URLs in that field.

---

description
TextField

An optional longer description of the category. Very useful on category landing pages.

meta_keywords
CharField(255)

Comma-separated keywords for search engines.

meta_extra
TextField

(Advanced) Any additional HTML to be placed verbatim in the <head> of the page.

2.8.3 Settings

The CATEGORIES_SETTINGS dictionary is where you can override the default settings. You don't have to include all the settings; only the ones which you want to override.

- *THUMBNAIL_UPLOAD_PATH*
- *THUMBNAIL_STORAGE*
- *JAVASCRIPT_URL*
- *ADMIN_FIELDSETS*

The default settings are:

```
CATEGORIES_SETTINGS = {
    'ALLOW_SLUG_CHANGE': False,
    'CACHE_VIEW_LENGTH': 0,
    'RELATION_MODELS': [],
    'M2M_REGISTRY': {},
    'FK_REGISTRY': {},
    'THUMBNAIL_UPLOAD_PATH': 'uploads/categories/thumbnails',
    'THUMBNAIL_STORAGE': settings.DEFAULT_FILE_STORAGE,
    'SLUG_TRANSLITERATOR': lambda x: x,
    'ADMIN_FIELDSETS': {}
}
```

ALLOW_SLUG_CHANGE

Default: `False`

Description: Changing the slug for a category can have serious consequences if it is used as part of a URL. Setting this to `True` will allow users to change the slug of a category.

SLUG_TRANSLITERATOR

Default: `lambda x:  x`

Description: Allows the specification of a function to convert non-ASCII characters in the potential slug to ASCII characters. Allows specifying a `callable()` or a string in the form of `'path.to.module.function'`.

A great tool for this is Unidecode. Use it by setting `SLUG_TRANSLITERATOR` to `'unidecode.unidecode`.

CACHE_VIEW_LENGTH

Default: `0`

Description: This setting will be deprecated soon, but in the mean time, it allows you to specify the amount of time each view result is cached.

RELATION_MODELS

Default: `[]`

Description: Relation models is a set of models that a user can associate with this category. You specify models using `'app_name.modelname'` syntax.

M2M_REGISTRY

Default: {}

Description: A dictionary where the keys are in `'app_name.model_name'` syntax, and the values are a string, dict, or tuple of dicts. See *Registering Models*.

FK_REGISTRY

Default: {}

Description: A dictionary where the keys are in `'app_name.model_name'` syntax, and the values are a string, dict, or tuple of dicts. See *Registering Models*.

REGISTER_ADMIN

Default: `True`

Description: If you write your own category class by subclassing `CategoryBase` then you probably have no use for registering the default `Category` class in the admin.

THUMBNAIL_UPLOAD_PATH

Default: `'uploads/categories/thumbnails'`

Description: Where thumbnails for the categories will be saved.

THUMBNAIL_STORAGE

Default: `settings.DEFAULT_FILE_STORAGE`

Description: How to store the thumbnails. Allows for external storage engines like S3.

JAVASCRIPT_URL

Default: `STATIC_URL or MEDIA_URL + 'js/'`

Description: Allows for customization of javascript placement.

ADMIN_FIELDSETS

Default: `{}`

Description: Allows for selective customization of the default behavior of adding the fields to the admin class. See *Adding the fields to the Admin* for more information.

2.8.4 Template tags and filters

Filters

category_path

Optional Parameter: separator string. *Default:* `" :: "`

Creates a path represented by a categories by joining the items with a separator.

Each path item will be coerced to unicode, so you can pass a list of category instances, if required.

Example using a list of categories:

```
{{ some_list|category_path }}
```

If `some_list` is `[ <Category: Country>, <Category: Country pop>, <Category: Urban Cowboy>]` the result will be:

```
Country :: Country pop :: Urban Cowboy
```

Example using a category node and optional separator parameter:

```
{{ some_node.get_ancestors|category_path:" > " }}
```

If `some_node` was category "Urban Cowboy", the result will be:

```
Country > Country pop > Urban Cowboy
```

tree_info

Optional Parameter: `"ancestors"`

Given a list of categories, it iterates over the list, generating a tuple of the current category and a dict containing information about the tree structure around it, with the following keys:

`'new_level'` `True` if the current item is the start of a new level in the tree, `False` otherwise.

`'closed_levels'` A list of levels which end after the current item. This will be an empty list if the next category's level is the same as or greater than the level of the current item.

Provide the optional argument, `"ancestors"`, to add a list of unicode representations of the ancestors of the current category, in descending order (root node first, immediate parent last), under the key 'ancestors'.

For example: given the sample tree below, the contents of the list which would be available under the 'ancestors' key are given on the right:

```
Country             ->  []
   Country pop      ->  [u'Country pop']
      Urban Cowboy  ->  [u'Country', u'Country pop']
```

Using this filter with unpacking in a {% for %} tag, you should have enough information about the tree structure to create a hierarchical representation of the tree.

```
{% for node,structure in objects|tree_info %}
    {% if structure.new_level %}<ul><li>{% else %}</li><li>{% endif %}
    {{ node.name }}
    {% for level in structure.closed_levels %}</li></ul>{% endfor %}
{% endfor %}
```

tree_queryset

Convert a regular category `QuerySet` into a new, ordered `QuerySet` that includes the categories selected and their ancestors.

This is especially helpful when you have a subset of categories and want to show the hierarchy for all the items.

For example, if we add it to the example for *tree_info*:

```
{% for node,structure in objects|tree_queryset|tree_info %}
    {% if structure.new_level %}<ul><li>{% else %}</li><li>{% endif %}
    {{ node.name }}
    {% for level in structure.closed_levels %}</li></ul>{% endfor %}
{% endfor %}
```

A list of unrelated categories such as `[<Category: Urban cowboy>, <Category: Urban comtemporary>]`, the above template example will output the two categories and their ancestors:

```
<ul><li>
Country
<ul><li>
Country pop
<ul><li>
Urban cowboy
</li></ul></li></ul></li></ul>
<ul><li>
Rhythm and blues
<ul><li>
Urban contemporary
</li></ul></li></ul>
```

Note: Categories that have similar ancestors are grouped accordingly. There is no duplication of the ancestor tree.

Inclusion tags

display_path_as_ul

Template Rendered: `categories/ul_tree.html`

Syntax 1: `{% display_path_as_ul <category_obj> %}`

Syntax 2: `{% display_path_as_ul <path_string>[ using="app.Model"] %}`

Render the category with ancestors, but no children.

Pass either an object that subclasses *CategoryBase* or a path string for the category. Add `using="app.Model"` to specify which model when using a path string. The default model used is *Category*.

Example, using Category model:

```
{% display_path_as_ul "/Grandparent/Parent" %}
```

Example, using custom model:

```
{% display_path_as_ul "/Grandparent/Parent" using="coolapp.MusicGenre" %}
```

Example, using an object:

```
{% display_path_as_ul category_obj %}
```

Returns:

```
<ul>
  <li><a href="/categories/">Top</a>
  <ul>
    <li><a href="/categories/grandparent/">Grandparent</a></li>
  </ul>
  </li>
</ul>
```

display_drilldown_as_ul

Template rendered: `categories/ul_tree.html`

Syntax 1: `{% display_drilldown_as_ul category_obj %}`

Syntax 2: `{% display_drilldown_as_ul "/Grandparent/Parent" [using="app.Model"] %}`

Render the category with ancestors and children.

Example, using Category model:

```
{% display_drilldown_as_ul "/Grandparent/Parent" %}
```

Example, using custom model:

```
{% display_drilldown_as_ul "/Grandparent/Parent" using="coolapp.MusicGenre" %}
```

Example, using an object:

```
{% display_drilldown_as_ul category_obj %}
```

Returns:

```
<ul>
  <li><a href="/categories/">Top</a>
  <ul>
    <li><a href="/categories/grandparent/">Grandparent</a>
    <ul>
      <li><a href="/categories/grandparent/parent/">Parent</a>
      <ul>
        <li><a href="/categories/grandparent/parent/child1">Child1</a></li>
        <li><a href="/categories/grandparent/parent/child2">Child2</a></li>
        <li><a href="/categories/grandparent/parent/child3">Child3</a></li>
      </ul>
      </li>
    </ul>
    </li>
  </ul>
  </li>
</ul>
```

breadcrumbs tag

Template rendered: `categories/breadcrumbs.html`

Syntax 1: `{% breadcrumbs category_obj [separator=" :: "] %}`

Syntax 2: `{% breadcrumbs "/Grandparent/Parent" [separator=" :: "] [using="app.Model"] %}`

Render breadcrumbs for the given path using `::` or the given separator.

Example using Category model:

```
{% breadcrumbs "/Grandparent/Parent" %}
```

Example using a custom model:

```
{% breadcrumbs "/Grandparent/Parent" using="coolapp.MusicGenre" %}
```

Example using an object:

```
{% breadcrumbs category_obj %}
```

Returns:

```
<a href="/categories/grandparent/">Grandparent</a> / Parent
```

You can alter the separator used in the template by adding a separator argument:

```
{% breadcrumbs category_obj separator=" &gt; " %}
```

Returns:

```
<a href="/categories/grandparent/">Grandparent</a> &gt; Parent
```

Template Tags

get_top_level_categories

Retrieves an alphabetical list of all the categories that have no parents.

Syntax:

```
{% get_top_level_categories [using "app.Model"] as categories %}
```

Returns an list of categories `[<category>, <category>, <category, ...]`

get_category_drilldown

Syntax 1: `{% get_category_drilldown <path_string> [using "app.Model"] as <varname> %}`

Syntax 2: `{% get_category_drilldown <object> as <varname> %}`

Retrieves the specified category, its ancestors and its immediate children as an iterable. Syntax 1 allows for the retrieval of the category object via a slash-delimited path. The optional `using "app.Model"` allows you to specify from which model to retrieve the object.

Example:

```
{% get_category_drilldown "/Grandparent/Parent" using "family.Member" as family %}
```

The second syntax uses an instance of any object that subclasses *CategoryBase*

```
{% get_category_drilldown category_obj as family %}
```

Both examples sets `family` to:

```
[Grandparent, Parent, Child 1, Child 2, Child n]
```

recursetree

This tag renders a section of your template recursively for each node in your tree.

For example:

```
<ul class="root">
    {% recursetree nodes %}
        <li>
            {{ node.name }}
            {% if not node.is_leaf_node %}
                <ul class="children">
                    {{ children }}
                </ul>
            {% endif %}
        </li>
    {% endrecursetree %}
</ul>
```

Note the special variables `node` and `children`. These are magically inserted into your context while you're inside the `recursetree` tag.

`node` is an instance of your MPTT model.

`children` : This variable holds the rendered HTML for the children of `node`.

Note: If you already have variables called `node` or `children` in your template, and you need to access them inside the `recursetree` block, you'll need to alias them to some other name first:

```
{% with node as friendly_node %}
    {% recursetree nodes %}
        {{ node.name }} is friends with {{ friendly_node.name }}
        {{ children }}
    {% endrecursetree %}
{% endwith %}
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index