django-treebeard Documentation

Release 4.0.1

Gustavo Picón

Apr 16, 2023

Contents

1 Overview **3**
- 1.1 Installation . . . 3
- 1.2 Tutorial . . . 4
- 1.3 Known Caveats . . . 6
- 1.4 Changelog . . . 7

2 Reference **13**
- 2.1 API . . . 13
- 2.2 Materialized Path trees . . . 23
- 2.3 Nested Sets trees . . . 27
- 2.4 Adjacency List trees . . . 29
- 2.5 Exceptions . . . 30

3 Additional features **31**
- 3.1 Admin . . . 31
- 3.2 Forms . . . 33

4 Development **35**
- 4.1 Running the Test Suite . . . 35

5 Indices and tables **37**

Python Module Index **39**

Index **41**

django-treebeard is a library that implements efficient tree implementations for the Django Web Framework 1.8+, written by Gustavo Picón and licensed under the Apache License 2.0.

`django-treebeard` is:

- **Flexible**: Includes 3 different tree implementations with the same API:
  1. *Adjacency List*
  2. *Materialized Path*
  3. *Nested Sets*
- **Fast**: Optimized non-naive tree operations
- **Easy**: Uses Django's Model inheritance with Abstract base classes. to define your own models.
- **Clean**: Testable and well tested code base. Code/branch test coverage is above 96%.

CHAPTER 1

Overview

1.1 Installation

1.1.1 Prerequisites

`django-treebeard` needs at least **Python 3.6** to run, and **Django 2.2 or later**.

1.1.2 Installing

You have several ways to install `django-treebeard`. If you're not sure, just use pip

pip (or easy_install)

You can install the release versions from django-treebeard's PyPI page using `pip`:

```
$ pip install django-treebeard
```

or if for some reason you can't use `pip`, you can try `easy_install`, (at your own risk):

```
$ easy_install --always-unzip django-treebeard
```

setup.py

Download a release from the treebeard download page and unpack it, then run:

```
$ python setup.py install
```

.deb packages

Both Debian and Ubuntu include `django-treebeard` as a package, so you can just use:

```
$ apt-get install python-django-treebeard
```

or:

```
$ aptitude install python-django-treebeard
```

Remember that the packages included in linux distributions are usually not the most recent versions.

1.1.3 Configuration

Add `'treebeard'` to the `INSTALLED_APPS` section in your django settings file.

Note: If you are going to use the *`TreeAdmin`* class, you need to add the path to treebeard's templates in `TEMPLATE_DIRS`. Also you need to add `django.template.context_processors.request` to `TEMPLATES['OPTIONS']['context_processors']` setting in your django settings file (see https://docs.djangoproject.com/en/1.11/ref/templates/upgrading/ for how to define this setting within the TEMPLATES settings). For more recent versions of Django, use `django.core.context_processors.request` instead.

1.2 Tutorial

Create a basic model for your tree. In this example we'll use a Materialized Path tree:

```
from django.db import models

from treebeard.mp_tree import MP_Node

class Category(MP_Node):
    name = models.CharField(max_length=30)

    node_order_by = ['name']

    def __str__(self):
        return 'Category: {}'.format(self.name)
```

Create and apply migrations:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Let's create some nodes:

```
>>> from treebeard_tutorial.models import Category
>>> get = lambda node_id: Category.objects.get(pk=node_id)
>>> root = Category.add_root(name='Computer Hardware')
>>> node = get(root.pk).add_child(name='Memory')
>>> get(node.pk).add_sibling(name='Hard Drives')
<Category: Category: Hard Drives>
>>> get(node.pk).add_sibling(name='SSD')
```

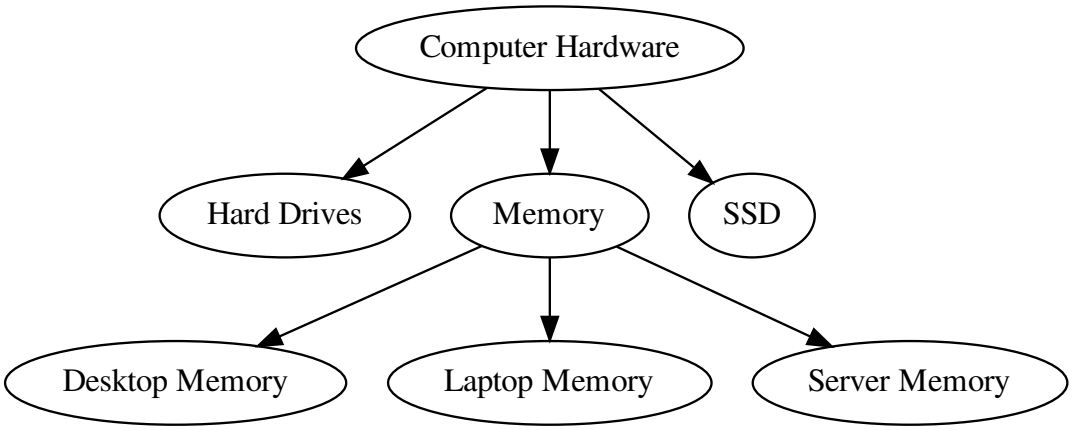
(continues on next page)

(continued from previous page)

```
<Category: Category: SSD>
>>> get(node.pk).add_child(name='Desktop Memory')
<Category: Category: Desktop Memory>
>>> get(node.pk).add_child(name='Laptop Memory')
<Category: Category: Laptop Memory>
>>> get(node.pk).add_child(name='Server Memory')
<Category: Category: Server Memory>
```

Note: Why retrieving every node again after the first operation? Because `django-treebeard` uses raw queries for most write operations, and raw queries don't update the django objects of the db entries they modify. See: *Known Caveats*.

We just created this tree:



You can see the tree structure with code:

```
>>> Category.dump_bulk()
[{'id': 1, 'data': {'name': u'Computer Hardware'},
  'children': [
     {'id': 3, 'data': {'name': u'Hard Drives'}},
     {'id': 2, 'data': {'name': u'Memory'},
      'children': [
         {'id': 5, 'data': {'name': u'Desktop Memory'}},
         {'id': 6, 'data': {'name': u'Laptop Memory'}},
         {'id': 7, 'data': {'name': u'Server Memory'}}]},
     {'id': 4, 'data': {'name': u'SSD'}}]}]
>>> Category.get_annotated_list()
[(<Category: Category: Computer Hardware>,
  {'close': [], 'level': 0, 'open': True}),
 (<Category: Category: Hard Drives>,
  {'close': [], 'level': 1, 'open': True}),
 (<Category: Category: Memory>,
  {'close': [], 'level': 1, 'open': False}),
```

(continues on next page)

(continued from previous page)

```
 (<Category: Category: Desktop Memory>,
  {'close': [], 'level': 2, 'open': True}),
 (<Category: Category: Laptop Memory>,
  {'close': [], 'level': 2, 'open': False}),
 (<Category: Category: Server Memory>,
  {'close': [0], 'level': 2, 'open': False}),
 (<Category: Category: SSD>,
  {'close': [0, 1], 'level': 1, 'open': False})]
>>> Category.get_annotated_list_qs(Category.objects.filter(name__icontains='Hardware
↪'))
[(<Category: Category: Computer Hardware>, {'open': True, 'close': [], 'level': 0})]
```

Read the *treebeard.models.Node* API reference for detailed info.

1.3 Known Caveats

1.3.1 Raw Queries

`django-treebeard` uses Django raw SQL queries for some write operations, and raw queries don't update the objects in the ORM since it's being bypassed.

Because of this, if you have a node in memory and plan to use it after a tree modification (adding/removing/moving nodes), you need to reload it.

1.3.2 Overriding the default manager

One of the most common source of bug reports in `django-treebeard` is the overriding of the default managers in the subclasses.

`django-treebeard` relies on the default manager for correctness and internal maintenance. If you override the default manager, by overriding the `objects` member in your subclass, you *WILL* have errors and inconsistencies in your tree.

To avoid this problem, if you need to override the default manager, you'll *NEED* to subclass the manager from the base manager class for the tree you are using.

Read the documentation in each tree type for details.

1.3.3 Custom Managers

Related to the previous caveat, if you need to create custom managers, you *NEED* to subclass the manager from the base manager class for the tree you are using.

Read the documentation in each tree type for details.

1.3.4 Copying model instances

Starting in version 4.5, we made a change to support custom names in primary fields that exposed a bug in Django's documentation. This has been fixed in the dev version of Django (3.2 as of writing this), but even when using older versions, the new instructions apply.

1.4 Changelog

1.4.1 Release 4.7 (Apr 7, 2023)

- Drop support for Django 4.0.
- Add support for Django 4.2.

1.4.2 Release 4.6.1 (Feb 5, 2023)

- Fix unescaped string representation of *AL_Node* models in the Django admin. Thanks to goodguyandy for reporting the issue.
- Optimise *MP_Node.get_descendants* to avoid database queries when called on a leaf node.

1.4.3 Release 4.6 (Jan 2, 2023)

- Drop support for Django 3.1 and lower.
- Add support for Django 4.0 and 4.1.
- Drop support for Python 3.7 and lower.
- Add support for Python 3.10 and Python 3.11.
- Change the return value of *delete()* for all node classes to be consistent with Django, and return a tuple of the number of objects deleted and a dictionary with the number of deletions per object type.
- Change the *delete()* methods for all node classes to accept arbitrary positional and keyword arguments which are passed to the parent method.
- Set *alters_data* and *queryset_only* attributes on the *delete()* methods for all node classes to prevent them being used in an unwanted context (e.g., in Django templates).
- Drop dependency on jQuery UI in the admin.

1.4.4 Release 4.5.1 (Feb 22, 2021)

- Removed unnecessary default in MP's depth field.

1.4.5 Release 4.5 (Feb 17, 2021)

- Add support for custom primary key fields with custom names.
- Add support for Python 3.9.
- Add support for MSSQL 2019.
- Add Code of conduct
- Removed outdated Sqlite workaround code
- Remove last remains of Python 2.7 code
- Use Pytest-django and fixtures for testing

1.4.6 Release 4.4 (Jan 13, 2021)

- Implement a non-destructive path-fixing algorithm for *MP_Node.fix_tree*.
- Ensure *post_save* is triggered *after* the parent node is updated in *MP_AddChildHandler*.
- Fix static URL generation to use *static* template tag instead of constructing the URL manually.
- Declare support for Django 2.2, 3.0 and 3.1.
- Drop support for Django 2.1 and lower.
- Drop support for Python 2.7 and Python 3.5.
- Increase performance for *MoveNodeForm* when using large trees.

1.4.7 Release 4.3.1 (Dec 25, 2019)

- Added check to avoid unnecessary database query for *MP_Node.get_ancestors()* if the node is a root node.
- Drop support for Python-3.4.
- Play more nicely with other form classes, that implement *__init__(self, *args, **kwargs)*, e.g. django-parler's *TranslatableModelForm*, where *kwargs.get('instance')* is *None* when called from here.
- Sorting on path on necessary queries, fixes some issues and stabilizes the whole MP section.
- Add German translation strings.

1.4.8 Release 4.3 (Apr 16, 2018)

- Support for Django-2.0

1.4.9 Release 4.2.2 (Mar 11, 2018)

- Bugfix issues #97: UnboundLocalError raised on treebeard admin

1.4.10 Release 4.2.1 (Mar 9, 2018)

- Bugfix issues #90: admin change list view and jsi18n load for Django-1.11

1.4.11 Release 4.2.0 (Dec 8, 2017)

- Support for Django-2.0

1.4.12 Release 4.1.2 (Jun 22, 2017)

- Fixed MANIFEST.in for Debian packaging.

1.4.13 Release 4.1.1 (May 24, 2017)

- Removed deprecated templatetag inclusion
- Added support for Python-3.6
- Added support for MS-SQL

1.4.14 Release 4.1.0 (Nov 24, 2016)

- Add support for Django-1.10
- Drop support for Django-1.7
- Moved Repository from Bitbucket to GitHub
- Moved documentation to https://django-treebeard.readthedocs.io/
- Moved continuous integration to https://travis-ci.org/django-treebeard/django-treebeard

1.4.15 Release 4.0.1 (May 1, 2016)

- Escape input in forms (Martin Koistinen / Divio)
- Clarification on model detail pages (Michael Huang)

1.4.16 Release 4.0 (Dec 28, 2015)

- Added support for 3.5 and Django 1.7, 1.8 and 1.9
- Django 1.6 is no longer supported.
- Remove deprecated backports needed for now unsupported Django versions
- Fixed a bug with queryset deletion not handling inheritance correctly.
- Assorted documentation fixes

1.4.17 Release 3.0 (Jan 18, 2015)

- Limited tests (and hence support) to Python 2.7+/3.4+ and Django 1.6+
- Removed usage of deprecated Django functions.
- Fixed documentation issues.
- Fixed issues in MoveNodeForm
- Added get_annotated_list_qs and max_depth for get_annotated_list

1.4.18 Release 2.0 (April 2, 2014)

- Stable release.

1.4.19 Release 2.0rc2 (March, 2014)

- Support models that use multi-table inheritance (Matt Wescott)
- Tree methods called on proxy models should consistently return instances of that proxy model (Matt Wescott)

1.4.20 Release 2.0rc1 (February, 2014)

- Fixed unicode related issue in the template tags.
- Major documentation cleanup.
- More warnings on the use of managers.
- Faster MP's is_root() method.

1.4.21 Release 2.0b2 (December, 2013)

- Dropped support for Python 2.5

1.4.22 Release 2.0b1 (May 29, 2013)

This is a beta release.

- Added support for Django 1.5 and Python 3.X
- Updated docs: the library supports python 2.5+ and Django 1.4+. Dropped support for older versions
- Revamped admin interface for MP and NS trees, supporting drag&drop to reorder nodes. Work on this patch was sponsored by the Oregon Center for Applied Science, inspired by FeinCMS developed by Jesús del Carpio with tests from Fernando Gutierrez. Thanks ORCAS!
- Updated setup.py to use distribute/setuptools instead of distutils
- Now using pytest for testing
- Small optimization to ns_tree.is_root
- Moved treebeard.tests to it's own directory (instead of tests.py)
- Added the runtests.py test runner
- Added tox support
- Fixed drag&drop bug in the admin
- Fixed a bug when moving MP_Nodes
- Using .pk instead of .id when accessing nodes.
- Removed the Benchmark (tbbench) and example (tbexample) apps.
- Fixed url parts join issues in the admin.
- Fixed: Now installing the static resources
- Fixed ManyToMany form field save handling
- In the admin, the node is now saved when moving so it can trigger handlers and/or signals.
- Improved translation files, including javascript.

- Renamed Node.get_database_engine() to Node.get_database_vendor(). As the name implies, it returns the database vendor instead of the engine used. Treebeard will get the value from Django, but you can subclass the method if needed.

1.4.23 Release 1.61 (Jul 24, 2010)

- Added admin i18n. Included translations: es, ru
- Fixed a bug when trying to introspect the database engine used in Django 1.2+ while using new style db settings (DATABASES). Added Node.get_database_engine to deal with this.

1.4.24 Release 1.60 (Apr 18, 2010)

- Added get_annotated_list
- Complete revamp of the documentation. It's now divided in sections for easier reading, and the package includes .rst files instead of the html build.
- Added raw id fields support in the admin
- Fixed setup.py to make it work in 2.4 again
- The correct ordering in NS/MP trees is now enforced in the queryset.
- Cleaned up code, removed some unnecessary statements.
- Tests refactoring, to make it easier to spot the model being tested.
- Fixed support of trees using proxied models. It was broken due to a bug in Django.
- Fixed a bug in add_child when adding nodes to a non-leaf in sorted MP.
- There are now 648 unit tests. Test coverage is 96%
- This will be the last version compatible with Django 1.0. There will be a a 1.6.X branch maintained for urgent bug fixes, but the main development will focus on recent Django versions.

1.4.25 Release 1.52 (Dec 18, 2009)

- Really fixed the installation of templates.

1.4.26 Release 1.51 (Dec 16, 2009)

- Forgot to include treebeard/tempates/*.html in MANIFEST.in

1.4.27 Release 1.5 (Dec 15, 2009)

New features added

- Forms
  - Added MoveNodeForm
- Django Admin
  - Added TreeAdmin

- MP_Node
  - Added 2 new checks in MP_Node.find_problems():
    4. a list of ids of nodes with the wrong depth value for their path
    5. a list of ids nodes that report a wrong number of children
  - Added a new (safer and faster but less comprehensive) MP_Node.fix_tree() approach.
- Documentation
  - Added warnings in the documentation when subclassing MP_Node or NS_Node and adding a new Meta.
  - HTML documentation is now included in the package.
  - CHANGES file and section in the docs.
- Other changes:
  - script to build documentation
  - updated numconv.py

Bugs fixed

- Added table quoting to all the sql queries that bypass the ORM. Solves bug in postgres when the table isn't created by syncdb.
- Removing unused method NS_Node._find_next_node
- Fixed MP_Node.get_tree to include the given parent when given a leaf node

1.4.28 Release 1.1 (Nov 20, 2008)

Bugs fixed

- Added exceptions.py

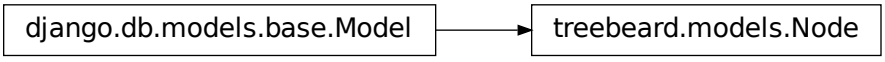
1.4.29 Release 1.0 (Nov 19, 2008)

- First public release.

CHAPTER 2

Reference

2.1 API



class treebeard.models.**Node**(**args*, ***kwargs*)

Bases: django.db.models.base.Model

Node class

This is the base class that defines the API of all tree models in this library:

- *treebeard.mp_tree.MP_Node* (materialized path)
- *treebeard.ns_tree.NS_Node* (nested sets)
- *treebeard.al_tree.AL_Node* (adjacency list)

> **Warning:** Please be aware of the *Known Caveats* when using this library.

classmethod add_root(***kwargs*)

Adds a root node to the tree. The new root node will be the new rightmost root node. If you want to insert a root node at a specific position, use *add_sibling()* in an already existing root node instead.

Parameters

- ****kwargs** – object creation data that will be passed to the inherited Node model

- **instance** – Instead of passing object creation data, you can pass an already-constructed (but not yet saved) model instance to be inserted into the tree.

Returns the created node object. It will be save()d by this method.

Raises ***NodeAlreadySaved*** – when the passed `instance` already exists in the database

Example:

```
MyNode.add_root(numval=1, strval='abcd')
```

Or, to pass in an existing instance:

```
new_node = MyNode(numval=1, strval='abcd')
MyNode.add_root(instance=new_node)
```

add_child(***kwargs*)

Adds a child to the node. The new node will be the new rightmost child. If you want to insert a node at a specific position, use the *add_sibling()* method of an already existing child node instead.

Parameters

- ****kwargs** – Object creation data that will be passed to the inherited Node model
- **instance** – Instead of passing object creation data, you can pass an already-constructed (but not yet saved) model instance to be inserted into the tree.

Returns The created node object. It will be save()d by this method.

Raises ***NodeAlreadySaved*** – when the passed `instance` already exists in the database

Example:

```
node.add_child(numval=1, strval='abcd')
```

Or, to pass in an existing instance:

```
new_node = MyNode(numval=1, strval='abcd')
node.add_child(instance=new_node)
```

add_sibling(*pos=None*, ***kwargs*)

Adds a new node as a sibling to the current node object.

Parameters

- **pos** – The position, relative to the current node object, where the new node will be inserted, can be one of:
  - `first-sibling`: the new node will be the new leftmost sibling
  - `left`: the new node will take the node's place, which will be moved to the right 1 position
  - `right`: the new node will be inserted at the right of the node
  - `last-sibling`: the new node will be the new rightmost sibling
  - `sorted-sibling`: the new node will be at the right position according to the value of node_order_by
- ****kwargs** – Object creation data that will be passed to the inherited Node model
- **instance** – Instead of passing object creation data, you can pass an already-constructed (but not yet saved) model instance to be inserted into the tree.

Returns The created node object. It will be saved by this method.

Raises

- ***InvalidPosition*** – when passing an invalid `pos` parm
- ***InvalidPosition*** – when `node_order_by` is enabled and the `pos` parm wasn't `sorted-sibling`
- ***MissingNodeOrderBy*** – when passing `sorted-sibling` as `pos` and the `node_order_by` attribute is missing
- ***NodeAlreadySaved*** – when the passed `instance` already exists in the database

Examples:

```
node.add_sibling('sorted-sibling', numval=1, strval='abc')
```

Or, to pass in an existing instance:

```
new_node = MyNode(numval=1, strval='abc')
node.add_sibling('sorted-sibling', instance=new_node)
```

delete(**args*, ***kwargs*)

Removes a node and all it's descendants.

Note: Call our queryset's delete to handle children removal. Subclasses will handle extra maintenance.

classmethod get_tree(*parent=None*)

Returns A list of nodes ordered as DFS, including the parent. If no parent is given, the entire tree is returned.

get_depth()

Returns the depth (level) of the node

Example:

```
node.get_depth()
```

get_ancestors()

Returns A queryset containing the current node object's ancestors, starting by the root node and descending to the parent. (some subclasses may return a list)

Example:

```
node.get_ancestors()
```

get_children()

Returns A queryset of all the node's children

Example:

```
node.get_children()
```

get_children_count()

Returns The number of the node's children

Example:

```
node.get_children_count()
```

get_descendants()

Returns A queryset of all the node's descendants, doesn't include the node itself (some subclasses may return a list).

Example:

```
node.get_descendants()
```

get_descendant_count()

Returns the number of descendants of a node.

Example:

```
node.get_descendant_count()
```

get_first_child()

Returns The leftmost node's child, or None if it has no children.

Example:

```
node.get_first_child()
```

get_last_child()

Returns The rightmost node's child, or None if it has no children.

Example:

```
node.get_last_child()
```

get_first_sibling()

Returns The leftmost node's sibling, can return the node itself if it was the leftmost sibling.

Example:

```
node.get_first_sibling()
```

get_last_sibling()

Returns The rightmost node's sibling, can return the node itself if it was the rightmost sibling.

Example:

```
node.get_last_sibling()
```

get_prev_sibling()

Returns The previous node's sibling, or None if it was the leftmost sibling.

Example:

```
node.get_prev_sibling()
```

get_next_sibling()

Returns The next node's sibling, or None if it was the rightmost sibling.

Example:

```
node.get_next_sibling()
```

get_parent(*update=False*)

Returns the parent node of the current node object. Caches the result in the object itself to help in loops.

Parameters **update** – Updates the cached value.

Example:

```
node.get_parent()
```

get_root()

Returns the root node for the current node object.

Example:

```
node.get_root()
```

get_siblings()

Returns A queryset of all the node's siblings, including the node itself.

Example:

```
node.get_siblings()
```

is_child_of(*node*)

Returns `True` if the node is a child of another node given as an argument, else, returns `False`

Parameters **node** – The node that will be checked as a parent

Example:

```
node.is_child_of(node2)
```

is_descendant_of(*node*)

Returns `True` if the node is a descendant of another node given as an argument, else, returns `False`

Parameters **node** – The node that will be checked as an ancestor

Example:

```
node.is_descendant_of(node2)
```

is_sibling_of(*node*)

Returns `True` if the node is a sibling of another node given as an argument, else, returns `False`

Parameters **node** – The node that will be checked as a sibling

Example:

```
node.is_sibling_of(node2)
```

is_root()

Returns True if the node is a root node (else, returns False)

Example:

```
node.is_root()
```

is_leaf()

Returns True if the node is a leaf node (else, returns False)

Example:

```
node.is_leaf()
```

move(*target*, *pos=None*)

Moves the current node and all it's descendants to a new position relative to another node.

Parameters

- **target** – The node that will be used as a relative child/sibling when moving
- **pos** – The position, relative to the target node, where the current node object will be moved to, can be one of:
  - `first-child`: the node will be the new leftmost child of the `target` node
  - `last-child`: the node will be the new rightmost child of the `target` node
  - `sorted-child`: the new node will be moved as a child of the `target` node according to the value of `node_order_by`
  - `first-sibling`: the node will be the new leftmost sibling of the `target` node
  - `left`: the node will take the `target` node's place, which will be moved to the right 1 position
  - `right`: the node will be moved to the right of the `target` node
  - `last-sibling`: the node will be the new rightmost sibling of the `target` node
  - `sorted-sibling`: the new node will be moved as a sibling of the `target` node according to the value of `node_order_by`

Note: If no `pos` is given the library will use `last-sibling`, or `sorted-sibling` if `node_order_by` is enabled.

Returns None

Raises

- ***InvalidPosition*** – when passing an invalid `pos` parm
- ***InvalidPosition*** – when `node_order_by` is enabled and the `pos` parm wasn't `sorted-sibling` or `sorted-child`
- ***InvalidMoveToDescendant*** – when trying to move a node to one of it's own descendants
- ***PathOverflow*** – when the library can't make room for the node's new position
- ***MissingNodeOrderBy*** – when passing `sorted-sibling` or `sorted-child` as `pos` and the `node_order_by` attribute is missing

Note: The node can be moved under another root node.

Examples:

```
node.move(node2, 'sorted-child')
node.move(node2, 'prev-sibling')
```

save (*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

classmethod get_first_root_node()

Returns The first root node in the tree or `None` if it is empty.

Example:

```
MyNodeModel.get_first_root_node()
```

classmethod get_last_root_node()

Returns The last root node in the tree or `None` if it is empty.

Example:

```
MyNodeModel.get_last_root_node()
```

classmethod get_root_nodes()

Returns A queryset containing the root nodes in the tree.

Example:

```
MyNodeModel.get_root_nodes()
```

classmethod load_bulk (*bulk_data*, *parent=None*, *keep_ids=False*)
Loads a list/dictionary structure to the tree.

Parameters

- **bulk_data** – The data that will be loaded, the structure is a list of dictionaries with 2 keys:
  - `data`: will store arguments that will be passed for object creation, and
  - `children`: a list of dictionaries, each one has it's own `data` and `children` keys (a recursive structure)
- **parent** – The node that will receive the structure as children, if not specified the first level of the structure will be loaded as root nodes
- **keep_ids** – If enabled, loads the nodes with the same primary keys that are given in the structure. Will error if there are nodes without primary key info or if the primary keys are already used.

Returns A list of the added node ids.

Note: Any internal data that you may have stored in your nodes' data (`path`, `depth`) will be ignored.

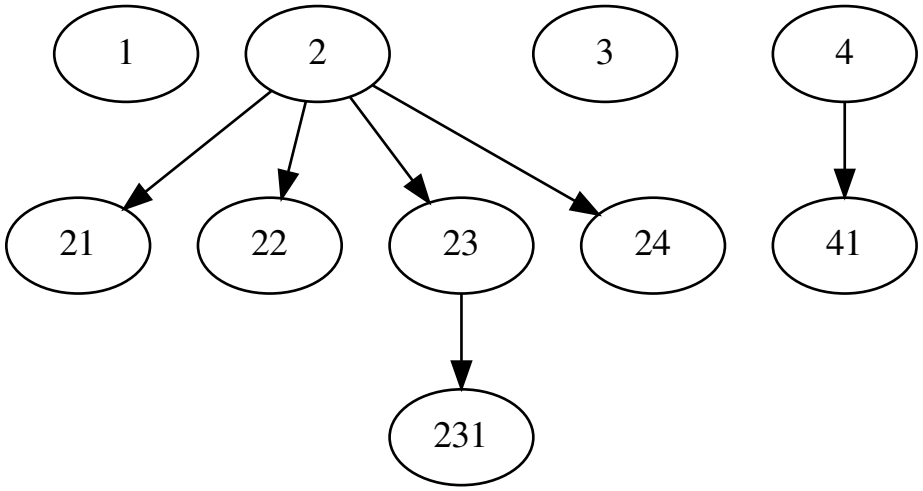
Note: If your node model has a ForeignKey this method will try to load the related object before loading the data. If the related object doesn't exist it won't load anything and will raise a DoesNotExist exception. This is done because the dump_data method uses integers to dump related objects.

Note: If your node model has `node_order_by` enabled, it will take precedence over the order in the structure.

Example:

```
data = [{'data':{'desc':'1'}},
        {'data':{'desc':'2'}, 'children':[
          {'data':{'desc':'21'}},
          {'data':{'desc':'22'}},
          {'data':{'desc':'23'}, 'children':[
            {'data':{'desc':'231'}},
          ]},
          {'data':{'desc':'24'}},
        ]},
        {'data':{'desc':'3'}},
        {'data':{'desc':'4'}, 'children':[
          {'data':{'desc':'41'}},
        ]},
]
# parent = None
MyNodeModel.load_bulk(data, None)
```

Will create:



classmethod dump_bulk(*parent=None*, *keep_ids=True*)
Dumps a tree branch to a python data structure.

Parameters

- **parent** – The node whose descendants will be dumped. The node itself will be included in the dump. If not given, the entire tree will be dumped.
- **keep_ids** – Stores the pk value (primary key) of every node. Enabled by default.

Returns A python data structure, described with detail in *load_bulk()*

Example:

```
tree = MyNodeModel.dump_bulk()
branch = MyNodeModel.dump_bulk(node_obj)
```

classmethod find_problems()
Checks for problems in the tree structure.

classmethod fix_tree()
Solves problems that can appear when transactions are not used and a piece of code breaks, leaving the tree in an inconsistent state.

classmethod get_descendants_group_count(*parent=None*)
Helper for a very common case: get a group of siblings and the number of *descendants* (not only children) in every sibling.

Parameters **parent** – The parent of the siblings to return. If no parent is given, the root nodes will be returned.

Returns A *list* (**NOT** a Queryset) of node objects with an extra attribute: *descendants_count*.

Example:

```
# get a list of the root nodes
root_nodes = MyModel.get_descendants_group_count()

for node in root_nodes:
    print '%s by %s (%d replies)' % (node.comment, node.author,
                                     node.descendants_count)
```

classmethod get_annotated_list(*parent=None*, *max_depth=None*)
Gets an annotated list from a tree branch.

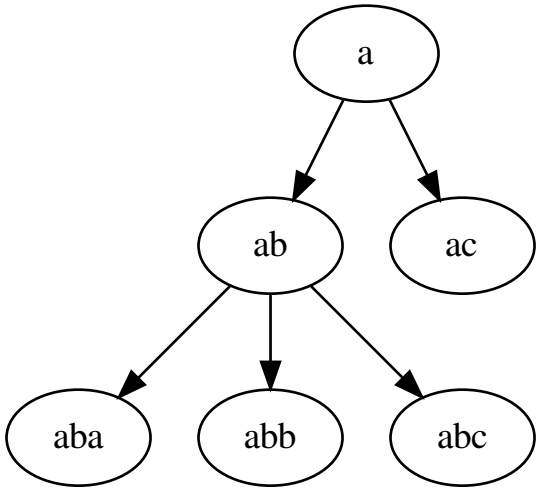
Parameters

- **parent** – The node whose descendants will be annotated. The node itself will be included in the list. If not given, the entire tree will be annotated.
- **max_depth** – Optionally limit to specified depth

Example:

```
annotated_list = MyModel.get_annotated_list()
```

With data:



Will return:

```
[
    (a,     {'open':True,  'close':[],    'level': 0})
    (ab,    {'open':True,  'close':[],    'level': 1})
    (aba,   {'open':True,  'close':[],    'level': 2})
    (abb,   {'open':False, 'close':[],    'level': 2})
    (abc,   {'open':False, 'close':[0,1], 'level': 2})
    (ac,    {'open':False, 'close':[0],   'level': 1})
]
```

This can be used with a template like:

```
{% for item, info in annotated_list %}
    {% if info.open %}
        <ul><li>
    {% else %}
        </li><li>
    {% endif %}

    {{ item }}

    {% for close in info.close %}
        </li></ul>
    {% endfor %}
{% endfor %}
```

Note: This method was contributed originally by Alexey Kinyov, using an idea borrowed from django-mptt.

New in version 1.55.

classmethod get_annotated_list_qs(*qs*)

Gets an annotated list from a queryset.

classmethod get_database_vendor(*action*)
returns the supported database vendor used by a treebeard model when performing read (select) or write (update, insert, delete) operations.

Parameters **action** – *read* or *write*

Returns postgresql, mysql or sqlite

Example:

```
MyNodeModel.get_database_vendor("write")
```

New in version 1.61.

2.2 Materialized Path trees

This is an efficient implementation of Materialized Path trees for Django, as described by Vadim Tropashko in SQL Design Patterns. Materialized Path is probably the fastest way of working with trees in SQL without the need of extra work in the database, like Oracle's `CONNECT BY` or sprocs and triggers for nested intervals.

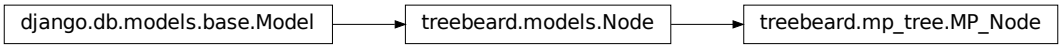
In a materialized path approach, every node in the tree will have a *path* attribute, where the full path from the root to the node will be stored. This has the advantage of needing very simple and fast queries, at the risk of inconsistency because of the denormalization of `parent`/`child` foreign keys. This can be prevented with transactions.

`django-treebeard` uses a particular approach: every step in the path has a fixed width and has no separators. This makes queries predictable and faster at the cost of using more characters to store a step. To address this problem, every step number is encoded.

Also, two extra fields are stored in every node: *depth* and *numchild*. This makes the read operations faster, at the cost of a little more maintenance on tree updates/inserts/deletes. Don't worry, even with these extra steps, materialized path is more efficient than other approaches.

Warning: As with all tree implementations, please be aware of the *Known Caveats*.

Note: The materialized path approach makes heavy use of `LIKE` in your database, with clauses like `WHERE path LIKE '002003%'`. If you think that `LIKE` is too slow, you're right, but in this case the *path* field is indexed in the database, and all `LIKE` clauses that don't **start** with a `%` character will use the index. This is what makes the materialized path approach so fast.



class treebeard.mp_tree.**MP_Node**(**args*, ***kwargs*)
Bases: *treebeard.models.Node*

Abstract model to create your own Materialized Path Trees.

Warning: Do not change the values of *path*, *depth* or *numchild* directly: use one of the included methods instead. Consider these values *read-only*.

Warning: Do not change the values of the *steplen*, *alphabet* or *node_order_by* after saving your first object. Doing so will corrupt the tree.

Warning: If you need to define your own Manager class, you'll need to subclass *MP_NodeManager*.

Also, if in your manager you need to change the default queryset handler, you'll need to subclass *MP_NodeQuerySet*.

Example:

```
class SortedNode(MP_Node):
    node_order_by = ['numval', 'strval']

    numval = models.IntegerField()
    strval = models.CharField(max_length=255)
```

Read the API reference of *treebeard.models.Node* for info on methods available in this class, or read the following section for methods with particular arguments or exceptions.

steplen

Attribute that defines the length of each step in the *path* of a node. The default value of *4* allows a maximum of *1679615* children per node. Increase this value if you plan to store large trees (a steplen of *5* allows more than *60M* children per node). Note that increasing this value, while increasing the number of children per node, will decrease the max *depth* of the tree (by default: *63*). To increase the max *depth*, increase the max_length attribute of the *path* field in your model.

alphabet

Attribute: the alphabet that will be used in base conversions when encoding the path steps into strings. The default value, 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ is the most optimal possible value that is portable between the supported databases (which means: their default collation will order the *path* field correctly).

Note: In case you know what you are doing, there is a test that is disabled by default that can tell you the optimal default alphabet in your enviroment. To run the test you must enable the TREEBEARD_TEST_ALPHABET enviroment variable:

```
$ TREEBEARD_TEST_ALPHABET=1 py.test -k test_alphabet
```

In OS X Mavericks, good readable values for the three supported databases in their *default* configuration:

Database	Optimal Alphabet	Base
MySQL 5.6.17	0-9A-Z	36
PostgreSQL 9.3.4	0-9A-Za-z	62
Sqlite3	0-9A-Za-z	62

The default value is MySQL's since it will work in all DBs, but when working with a better database, changing the *alphabet* value is recommended in order to increase the density of the paths.

For an even better approach, change the collation of the *path* column in the database to handle raw ASCII, and use the printable ASCII characters (0x20 to 0x7E) as the *alphabet*.

node_order_by
Attribute: a list of model fields that will be used for node ordering. When enabled, all tree operations will assume this ordering.

Example:

```
node_order_by = ['field1', 'field2', 'field3']
```

path
`CharField`, stores the full materialized path for each node. The default value of it's max_length, *255*, is the max efficient and portable value for a `varchar`. Increase it to allow deeper trees (max depth by default: *63*)

Note: *django-treebeard* uses Django's abstract model inheritance, so to change the `max_length` value of the path in your model, you have to redeclare the path field in your model:

```
class MyNodeModel(MP_Node):
    path = models.CharField(max_length=1024, unique=True)
```

Note: For performance, and if your database allows it, you can safely define the path column as ASCII (not utf-8/unicode/iso8859-1/etc) to keep the index smaller (and faster). Also note that some databases (mysql) have a small index size limit. InnoDB for instance has a limit of 765 bytes per index, so that would be the limit if your path is ASCII encoded. If your path column in InnoDB is using unicode, the index limit will be 255 characters since in MySQL's indexes, unicode means 3 bytes per character.

Note: `django-treebeard` uses numconv for path encoding.

depth
`PositiveIntegerField`, depth of a node in the tree. A root node has a depth of *1*.

numchild
`PositiveIntegerField`, the number of children of the node.

classmethod add_root(***kwargs*)
Adds a root node to the tree.

This method saves the node in database. The object is populated as if via:

` obj = cls(**kwargs) `

Raises ***PathOverflow*** – when no more root objects can be added

See: *treebeard.models.Node.add_root()*

add_child(***kwargs*)
Adds a child to the node.

This method saves the node in database. The object is populated as if via:

` obj = self.__class__(**kwargs) `

Raises ***PathOverflow*** – when no more child nodes can be added

See: *treebeard.models.Node.add_child()*

add_sibling(*pos=None*, ***kwargs*)

Adds a new node as a sibling to the current node object.

This method saves the node in database. The object is populated as if via:

```
` obj = self.__class__(**kwargs) `
```

Raises ***PathOverflow*** – when the library can't make room for the node's new position

See: *treebeard.models.Node.add_sibling()*

move(*target*, *pos=None*)

Moves the current node and all it's descendants to a new position relative to another node.

Raises ***PathOverflow*** – when the library can't make room for the node's new position

See: *treebeard.models.Node.move()*

classmethod get_tree(*parent=None*)

Returns A *queryset* of nodes ordered as DFS, including the parent. If no parent is given, the entire tree is returned.

See: *treebeard.models.Node.get_tree()*

Note: This metod returns a queryset.

---

classmethod find_problems()

Checks for problems in the tree structure, problems can occur when:

1. your code breaks and you get incomplete transactions (always use transactions!)
2. changing the `steplen` value in a model (you must `dump_bulk()` first, change `steplen` and then `load_bulk()`

Returns

A tuple of five lists:

1. a list of ids of nodes with characters not found in the `alphabet`
2. a list of ids of nodes when a wrong `path` length according to `steplen`
3. a list of ids of orphaned nodes
4. a list of ids of nodes with the wrong depth value for their path
5. a list of ids nodes that report a wrong number of children

---

Note: A node won't appear in more than one list, even when it exhibits more than one problem. This method stops checking a node when it finds a problem and continues to the next node.

Note: Problems 1, 2 and 3 can't be solved automatically.

Example:

```
MyNodeModel.find_problems()
```

classmethod fix_tree(*destructive=False*, *fix_paths=False*)

Solves some problems that can appear when transactions are not used and a piece of code breaks, leaving the tree in an inconsistent state.

The problems this method solves are:

1. Nodes with an incorrect `depth` or `numchild` values due to incorrect code and lack of database transactions.
2. "Holes" in the tree. This is normal if you move/delete nodes a lot. Holes in a tree don't affect performance,
3. Incorrect ordering of nodes when `node_order_by` is enabled. Ordering is enforced on *node insertion*, so if an attribute in `node_order_by` is modified after the node is inserted, the tree ordering will be inconsistent.

Parameters

- **fix_paths** – A boolean value. If True, a slower, more complex fix_tree method will be attempted. If False (the default), it will use a safe (and fast!) fix approach, but it will only solve the `depth` and `numchild` nodes, it won't fix the tree holes or broken path ordering.
- **destructive** – Deprecated; alias for `fix_paths`.

Example:

```
MyNodeModel.fix_tree()
```

class treebeard.mp_tree.**MP_NodeManager**

Bases: `django.db.models.manager.Manager`

Custom manager for nodes in a Materialized Path tree.

class treebeard.mp_tree.**MP_NodeQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Bases: `django.db.models.query.QuerySet`

Custom queryset for the tree node manager.

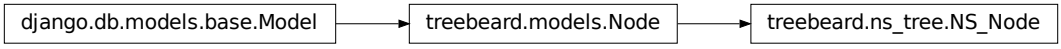
Needed only for the custom delete method.

2.3 Nested Sets trees

An implementation of Nested Sets trees for Django, as described by Joe Celko in Trees and Hierarchies in SQL for Smarties.

Nested sets have very efficient reads at the cost of high maintenance on write/delete operations.

Warning: As with all tree implementations, please be aware of the *Known Caveats*.



class treebeard.ns_tree.**NS_Node**(**args*, ***kwargs*)

Bases: *treebeard.models.Node*

Abstract model to create your own Nested Sets Trees.

> **Warning:** If you need to define your own `Manager` class, you'll need to subclass *NS_NodeManager*.
>
> Also, if in your manager you need to change the default queryset handler, you'll need to subclass *NS_NodeQuerySet*.

node_order_by

Attribute: a list of model fields that will be used for node ordering. When enabled, all tree operations will assume this ordering.

Example:

```
node_order_by = ['field1', 'field2', 'field3']
```

depth

`PositiveIntegerField`, depth of a node in the tree. A root node has a depth of *1*.

lft

`PositiveIntegerField`

rgt

`PositiveIntegerField`

tree_id

`PositiveIntegerField`

classmethod get_tree(*parent=None*)

Returns A *queryset* of nodes ordered as DFS, including the parent. If no parent is given, all trees are returned.

See: *treebeard.models.Node.get_tree()*

> **Note:** This method returns a queryset.

class treebeard.ns_tree.**NS_NodeManager**

Bases: `django.db.models.manager.Manager`

Custom manager for nodes in a Nested Sets tree.

class treebeard.ns_tree.**NS_NodeQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Bases: `django.db.models.query.QuerySet`

Custom queryset for the tree node manager.

Needed only for the customized delete method.

2.4 Adjacency List trees

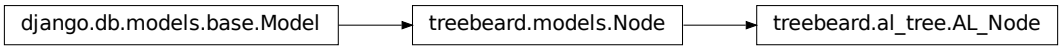
This is a simple implementation of the traditional Adjacency List Model for storing trees in relational databases.

In the adjacency list model, every node will have a "*parent*" key, that will be NULL for root nodes.

Since `django-treebeard` must return trees ordered in a predictable way, the ordering for models without the *node_order_by* attribute will have an extra attribute that will store the relative position of a node between it's siblings: *sib_order*.

The adjacency list model has the advantage of fast writes at the cost of slow reads. If you read more than you write, use *MP_Node* instead.

Warning: As with all tree implementations, please be aware of the *Known Caveats*.



class treebeard.al_tree.**AL_Node**(**args*, ***kwargs*)

Bases: *treebeard.models.Node*

Abstract model to create your own Adjacency List Trees.

Warning: If you need to define your own `Manager` class, you'll need to subclass *AL_NodeManager*.

node_order_by

Attribute: a list of model fields that will be used for node ordering. When enabled, all tree operations will assume this ordering.

Example:

```
node_order_by = ['field1', 'field2', 'field3']
```

parent

`ForeignKey` to itself. This attribute **MUST** be defined in the subclass (sadly, this isn't inherited correctly from the ABC in *Django 1.0*). Just copy&paste these lines to your model:

```
parent = models.ForeignKey('self',
                           related_name='children_set',
                           null=True,
                           db_index=True)
```

sib_order

`PositiveIntegerField` used to store the relative position of a node between it's siblings. This

attribute is mandatory *ONLY* if you don't set a *node_order_by* field. You can define it copy&pasting this line in your model:

```
sib_order = models.PositiveIntegerField()
```

Examples:

```
class AL_TestNode(AL_Node):
    parent = models.ForeignKey('self',
                               related_name='children_set',
                               null=True,
                               db_index=True)
    sib_order = models.PositiveIntegerField()
    desc = models.CharField(max_length=255)

class AL_TestNodeSorted(AL_Node):
    parent = models.ForeignKey('self',
                               related_name='children_set',
                               null=True,
                               db_index=True)
    node_order_by = ['val1', 'val2', 'desc']
    val1 = models.IntegerField()
    val2 = models.IntegerField()
    desc = models.CharField(max_length=255)
```

Read the API reference of *treebeard.models.Node* for info on methods available in this class, or read the following section for methods with particular arguments or exceptions.

get_depth(*update=False*)

Returns the depth (level) of the node Caches the result in the object itself to help in loops.

Parameters **update** – Updates the cached value.

See: *treebeard.models.Node.get_depth()*

class treebeard.al_tree.**AL_NodeManager**

Bases: django.db.models.manager.Manager

Custom manager for nodes in an Adjacency List tree.

2.5 Exceptions

exception treebeard.exceptions.**InvalidPosition**

Raised when passing an invalid pos value

exception treebeard.exceptions.**InvalidMoveToDescendant**

Raised when attempting to move a node to one of it's descendants.

exception treebeard.exceptions.**NodeAlreadySaved**

Raised when attempting to add a node which is already saved to the database.

exception treebeard.exceptions.**PathOverflow**

Raised when trying to add or move a node to a position where no more nodes can be added (see path and alphabet for more info)

exception treebeard.exceptions.**MissingNodeOrderBy**

Raised when an operation needs a missing node_order_by attribute

CHAPTER 3

Additional features

3.1 Admin

3.1.1 API

class treebeard.admin.**TreeAdmin**(*model*, *admin_site*)

Bases: django.contrib.admin.options.ModelAdmin

Django Admin class for treebeard.

Example:

```
from django.contrib import admin
from treebeard.admin import TreeAdmin
from treebeard.forms import movenodeform_factory
from myproject.models import MyNode

class MyAdmin(TreeAdmin):
    form = movenodeform_factory(MyNode)

admin.site.register(MyNode, MyAdmin)
```

treebeard.admin.**admin_factory**(*form_class*)

Dynamically build a TreeAdmin subclass for the given form class.

Parameters **form_class** –

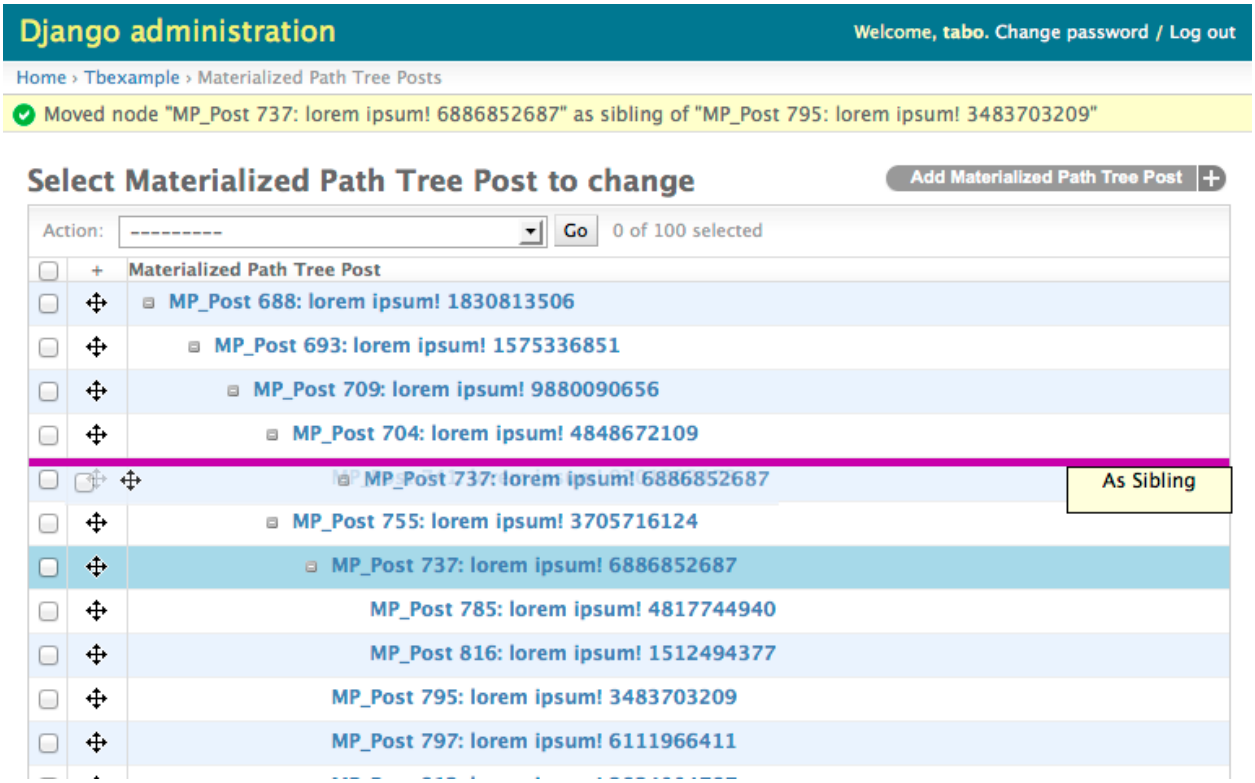
Returns A TreeAdmin subclass.

3.1.2 Interface

The features of the admin interface will depend on the tree type.

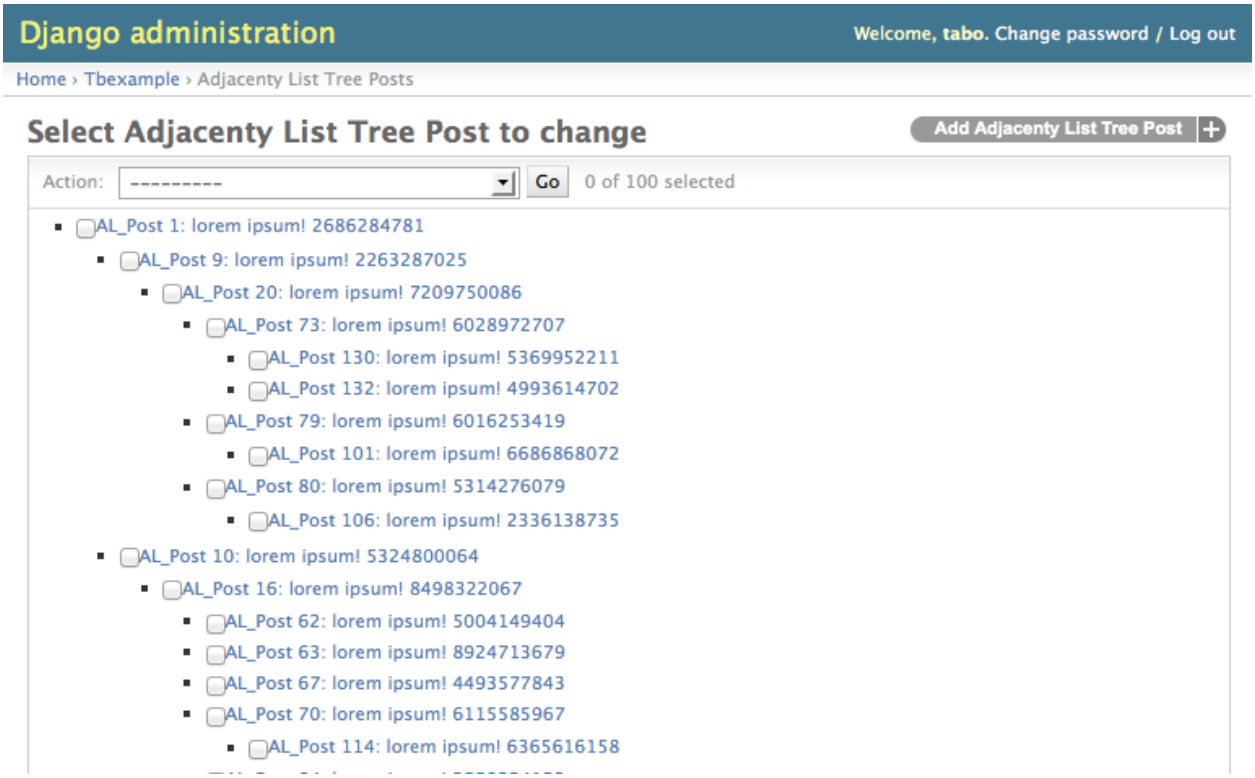
Advanced Interface

Materialized Path and *Nested Sets* trees have an AJAX interface based on FeinCMS, that includes features like drag&drop and an attractive interface.



Basic Interface

Adjacency List trees have a basic admin interface.



Model Detail Pages

If a model's field values are modified, then it is necessary to add the fields '_position' and '_ref_node_id'. Otherwise, it is not possible to create instances of the model.

Example:

```
class MyAdmin(TreeAdmin):
    list_display = ('title', 'body', 'is_edited', 'timestamp', '_position',
↪'_ref_node_id',)
    form = movenodeform_factory(MyNode)

admin.site.register(MyNode, MyAdmin)
```

3.2 Forms

class treebeard.forms.**MoveNodeForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*, ***kwargs*)

Bases: `django.forms.models.ModelForm`

Form to handle moving a node in a tree.

Handles sorted/unsorted trees.

It adds two fields to the form:

- **Relative to: The target node where the current node will** be moved to.

- **Position: The position relative to the target node that** will be used to move the node. These can be:
  - For sorted trees: `Child of` and `Sibling of`
  - For unsorted trees: `First child of`, `Before` and `After`

> **Warning:** Subclassing *MoveNodeForm* directly is discouraged, since special care is needed to handle excluded fields, and these change depending on the tree type.
>
> It is recommended that the *movenodeform_factory()* function is used instead.

`treebeard.forms.`**`movenodeform_factory`**(*model*, *form=<class 'treebeard.forms.MoveNodeForm'>*, *fields=None*, *exclude=None*, *formfield_callback=None*, *widgets=None*)

Dynamically build a MoveNodeForm subclass with the proper Meta.

Parameters

- **`model`** (`Node`) – The subclass of `Node` that will be handled by the form.
- **`form`** – The form class that will be used as a base. By default, *MoveNodeForm* will be used.

Returns A *MoveNodeForm* subclass

For a full reference of this function, please read `modelform_factory()`

Example, `MyNode` is a subclass of *treebeard.al_tree.AL_Node*:

```
MyNodeForm = movenodeform_factory(MyNode)
```

is equivalent to:

```
class MyNodeForm(MoveNodeForm):
    class Meta:
        model = models.MyNode
        exclude = ('sib_order', 'parent')
```

CHAPTER 4

Development

4.1 Running the Test Suite

`django-treebeard` includes a comprehensive test suite. It is highly recommended that you run and update the test suite when you send patches.

4.1.1 pytest

You will need pytest to run the test suite:

```
$ pip install pytest
```

Then just run the test suite:

```
$ pytest
```

You can use all the features and plugins of pytest this way.

By default the test suite will run using a sqlite3 database in RAM, but you can change this setting environment variables:

DATABASE_USER

DATABASE_PASSWORD

DATABASE_HOST

DATABASE_USER_POSTGRES

DATABASE_PORT_POSTGRES

DATABASE_USER_MYSQL

DATABASE_PORT_MYSQL

Sets the database settings to be used by the test suite. Useful if you want to test the same database engine/version you use in production.

4.1.2 tox

`django-treebeard` uses tox to run the test suite in all the supported environments - permutations of:

- Python 3.8 - 3.11
- Django 3.2, 4.1 and 4.2
- Sqlite, MySQL, PostgreSQL and MSSQL

This means that there are a lot of permutations, which takes a long time. If you want to test only one or a few environments, use the *-e* option in tox, like:

```
$ tox -e py39-dj32-postgres
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

treebeard.admin, 31
treebeard.al_tree, 29
treebeard.exceptions, 30
treebeard.forms, 33
treebeard.models, 13
treebeard.mp_tree, 23
treebeard.ns_tree, 27

Index

A

add_child() (*treebeard.models.Node method*), 14
add_child() (*treebeard.mp_tree.MP_Node method*), 25
add_root() (*treebeard.models.Node class method*), 13
add_root() (*treebeard.mp_tree.MP_Node class method*), 25
add_sibling() (*treebeard.models.Node method*), 14
add_sibling() (*treebeard.mp_tree.MP_Node method*), 26
admin_factory() (*in module treebeard.admin*), 31
AL_Node (*class in treebeard.al_tree*), 29
AL_NodeManager (*class in treebeard.al_tree*), 30
alphabet (*treebeard.mp_tree.MP_Node attribute*), 24

C

command line option
    DATABASE_HOST, 35
    DATABASE_PASSWORD, 35
    DATABASE_PORT_MYSQL, 35
    DATABASE_PORT_POSTGRES, 35
    DATABASE_USER, 35
    DATABASE_USER_MYSQL, 35
    DATABASE_USER_POSTGRES, 35

D

DATABASE_HOST
    command line option, 35
DATABASE_PASSWORD
    command line option, 35
DATABASE_PORT_MYSQL
    command line option, 35
DATABASE_PORT_POSTGRES
    command line option, 35
DATABASE_USER
    command line option, 35
DATABASE_USER_MYSQL
    command line option, 35
DATABASE_USER_POSTGRES
    command line option, 35
delete() (*treebeard.models.Node method*), 15
depth (*treebeard.mp_tree.MP_Node attribute*), 25
depth (*treebeard.ns_tree.NS_Node attribute*), 28
dump_bulk() (*treebeard.models.Node class method*), 20

E

environment variable
    TREEBEARD_TEST_ALPHABET, 24

F

find_problems() (*treebeard.models.Node class method*), 21
find_problems() (*treebeard.mp_tree.MP_Node class method*), 26
fix_tree() (*treebeard.models.Node class method*), 21
fix_tree() (*treebeard.mp_tree.MP_Node class method*), 27

G

get_ancestors() (*treebeard.models.Node method*), 15
get_annotated_list() (*treebeard.models.Node class method*), 21
get_annotated_list_qs() (*treebeard.models.Node class method*), 22
get_children() (*treebeard.models.Node method*), 15
get_children_count() (*treebeard.models.Node method*), 15
get_database_vendor() (*treebeard.models.Node class method*), 23
get_depth() (*treebeard.al_tree.AL_Node method*), 30
get_depth() (*treebeard.models.Node method*), 15
get_descendant_count() (*treebeard.models.Node method*), 16
get_descendants() (*treebeard.models.Node method*), 16

get_descendants_group_count() (*treebeard.models.Node class method*), 21
get_first_child() (*treebeard.models.Node method*), 16
get_first_root_node() (*treebeard.models.Node class method*), 19
get_first_sibling() (*treebeard.models.Node method*), 16
get_last_child() (*treebeard.models.Node method*), 16
get_last_root_node() (*treebeard.models.Node class method*), 19
get_last_sibling() (*treebeard.models.Node method*), 16
get_next_sibling() (*treebeard.models.Node method*), 16
get_parent() (*treebeard.models.Node method*), 17
get_prev_sibling() (*treebeard.models.Node method*), 16
get_root() (*treebeard.models.Node method*), 17
get_root_nodes() (*treebeard.models.Node class method*), 19
get_siblings() (*treebeard.models.Node method*), 17
get_tree() (*treebeard.models.Node class method*), 15
get_tree() (*treebeard.mp_tree.MP_Node class method*), 26
get_tree() (*treebeard.ns_tree.NS_Node class method*), 28

I

InvalidMoveToDescendant, 30
InvalidPosition, 30
is_child_of() (*treebeard.models.Node method*), 17
is_descendant_of() (*treebeard.models.Node method*), 17
is_leaf() (*treebeard.models.Node method*), 18
is_root() (*treebeard.models.Node method*), 17
is_sibling_of() (*treebeard.models.Node method*), 17

L

lft (*treebeard.ns_tree.NS_Node attribute*), 28
load_bulk() (*treebeard.models.Node class method*), 19

M

MissingNodeOrderBy, 30
move() (*treebeard.models.Node method*), 18
move() (*treebeard.mp_tree.MP_Node method*), 26
MoveNodeForm (*class in treebeard.forms*), 33
movenodeform_factory() (*in module treebeard.forms*), 34
MP_Node (*class in treebeard.mp_tree*), 23
MP_NodeManager (*class in treebeard.mp_tree*), 27
MP_NodeQuerySet (*class in treebeard.mp_tree*), 27

N

Node (*class in treebeard.models*), 13
node_order_by (*treebeard.al_tree.AL_Node attribute*), 29
node_order_by (*treebeard.mp_tree.MP_Node attribute*), 25
node_order_by (*treebeard.ns_tree.NS_Node attribute*), 28
NodeAlreadySaved, 30
NS_Node (*class in treebeard.ns_tree*), 28
NS_NodeManager (*class in treebeard.ns_tree*), 28
NS_NodeQuerySet (*class in treebeard.ns_tree*), 28
numchild (*treebeard.mp_tree.MP_Node attribute*), 25

P

parent (*treebeard.al_tree.AL_Node attribute*), 29
path (*treebeard.mp_tree.MP_Node attribute*), 25
PathOverflow, 30

R

rgt (*treebeard.ns_tree.NS_Node attribute*), 28

S

save() (*treebeard.models.Node method*), 19
sib_order (*treebeard.al_tree.AL_Node attribute*), 29
steplen (*treebeard.mp_tree.MP_Node attribute*), 24

T

tree_id (*treebeard.ns_tree.NS_Node attribute*), 28
TreeAdmin (*class in treebeard.admin*), 31
treebeard.admin (*module*), 31
treebeard.al_tree (*module*), 29
treebeard.exceptions (*module*), 30
treebeard.forms (*module*), 33
treebeard.models (*module*), 13
treebeard.mp_tree (*module*), 23
treebeard.ns_tree (*module*), 27
TREEBEARD_TEST_ALPHABET, 24