Nodular Documentation

Release 0.1.0

HasGeek

Jul 14, 2017

Contents

CHAPTER 1

Nodular's documentation

Nodular provides a base class and helpers for hierarchical content objects.

Contents:

Database connection

Nodular provides a Flask-SQLAlchemy database object that all models in your app must use. Typical usage:

```
from nodular import db
from coaster.sqlalchemy import BaseMixin

class MyModel(BaseMixin, db.Model):
    pass
```

To initialize with an app:

```
from flask import Flask
app = Flask(__name__)
db.init_app(app)
```

If you have only one app per Python process (which is typical), add this line to your init sequence:

```
db.app = app
```

This makes your app the default app for this database object and removes the need to use `app.test_request_context()` when querying the database outside a request context.

Node model

Nodular's NodeMixin and Node models are the base classes for all content objects.

class nodular.node.**Node**(***kwargs*)

Base class for all content objects.

access_for(*roles=None*, *user=None*, *token=None*)

Return a proxy object that limits read and write access to attributes based on the user's roles. If the `roles` parameter isn't provided, but a `user` or `token` is provided instead, *roles_for()* is called:

```
# This typical call:
obj.access_for(user=current_user)
# Is shorthand for:
obj.access_for(roles=obj.roles_for(user=current_user))
```

aliases

Dictionary of all aliases for renamed, moved or deleted sub-nodes.

as_dict()

Export the node as a dictionary.

buid

URL-friendly UUID representation, using URL-safe Base64 (BUID)

etype

Effective type of this instance

classmethod get(*buid*)

Get a node by its buid.

getprop(*key*, *default=None*)

Return the inherited value of a property from the closest parent node on which it was set.

id

Database identity for this model, used for foreign key references from other models

import_from(*data*)

Import the node from a dictionary.

make_name(*reserved=[]*)

Autogenerates a *name* from the *title*. If the auto-generated name is already in use in this model, *make_name()* tries again by suffixing numbers starting with 2 until an available name is found.

make_token_for(*user*, *roles=None*, *token=None*)

Generate a token for the specified user that grants access to this object alone, with either all roles available to the user, or just the specified subset. If an existing token is available, add to it.

This method should return `None` if a token cannot be generated. Must be implemented by subclasses.

name

The URL name of this object, unique within a parent container

nodes

Dictionary of all sub-nodes.

parent_id

Container for this node (used mainly to enforce uniqueness of 'name').

path

Path to this node for URL traversal.

permissions(*user*, *inherited=None*)

Permissions for this model, plus permissions inherited from the parent.

query_class

alias of `Query`

roles_for(*user=None*, *token=None*)
Return roles available to the given `user` or `token` on this object. The data type for both parameters are intentionally undefined here. Subclasses are free to define them in any way appropriate. Users and tokens are assumed to be valid.

The role `all` is always granted. If either `user` or `token` is specified, the role `user` is granted. If neither, `anon` is granted.

root
The root node for this node's tree.

short_title()
Generates an abbreviated title by subtracting the parent's title from this instance's title.

suuid
URL-friendly UUID representation, using ShortUUID

title
The title of this object

upsert(*parent*, *name*, ***fields*)
Insert or update an instance

url_for(*action='view'*, ***kwargs*)
Return public URL to this instance for a given action (default 'view')

url_id
URL-friendly UUID representation as a hex string

users_with(*roles*)
Return an iterable of all users who have the specified roles on this object. The iterable may be a list, tuple, set or SQLAlchemy query.

Must be implemented by subclasses.

uuid
UUID column, or synonym to existing *id* column if that is a UUID

class nodular.node.**NodeAlias**(***kwargs*)
When a node is renamed, it gets an alias connecting the old name to the new. NodeAlias makes it possible for users to rename nodes without breaking links.

query_class
alias of `Query`

class nodular.node.**NodeMixin**
NodeMixin provides functionality for content objects to connect to the Node base table. NodeMixin and Node should be used together:

```
class MyContentType(NodeMixin, Node):
    __tablename__ = 'my_content_type'
    my_column = Column(...)
```

NodeMixin will use `__tablename__` as the node `type` identifier and will autogenerate a `__title__` attribute. This title is used in the UI when the user adds a new node.

permissions(*user*, *inherited=None*)
Return permissions available to the given user on this object

class nodular.node.**ProxyDict**(*parent*, *collection_name*, *childclass*, *keyname*, *parentkey*)
Proxies a dictionary on a relationship. This is intended for use with `lazy='dynamic'` relationships, but can also be used with regular InstrumentedList relationships.

ProxyDict is used for *Node.nodes* and *Node.aliases*.

Parameters

- **parent** – The instance in which this dictionary exists.
- **collection_name** – The relationship that is being proxied.
- **childclass** – The model referred to in the relationship.
- **keyname** – Attribute in childclass that will be the dictionary key.
- **parentkey** – Attribute in childclass that refers back to this parent.

clear() → None. Remove all items from D.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

nodular.node.**pathjoin**(*a*, **p*)
Join two or more pathname components, inserting '/' as needed. If any component is an absolute path, all previous path components will be discarded.

Note: This function is the same as `os.path.join()` on POSIX systems but is reproduced here so that Nodular can be used in non-POSIX environments.

Revisioned content

Nodular's RevisionedNodeMixin base class helps make models with revisioned content.

class nodular.revisioned.**RevisionedNodeMixin**
RevisionedNodeMixin replaces NodeMixin for models that need to keep their entire contents under revision control. A revisioned node can have multiple simultaneously active versions (each with a label) or archived versions (label=None).

Revisions are stored as distinct table rows with full content, not as diffs. All columns that need revisioning must be in the `RevisionMixin` model, not in the *RevisionedNodeMixin* model. Usage:

```
class MyDocument(RevisionedNodeMixin, Node):
    __tablename__ = u'my_document'
```

```
class MyDocumentRevision(MyDocument.RevisionMixin, db.Model):
    # __tablename__ is auto-generated
    content = db.Column(db.UnicodeText)
    ...
```

revise(*revision=None*, *user=None*, *workflow_label=None*)
Clone the given revision or make a new blank revision.

Returns New revision object

set_workflow_label(*revision*, *workflow_label*)
Set the workflow label for the given revision.

class nodular.revisioned.RevisionedNodeMixin.**RevisionMixin**
The RevisionMixin base class is available from subclasses of *RevisionedNodeMixin*. It describes the model for a revision of the *RevisionedNodeMixin* node and defines relationships between the two models.

For usage instructions, see *RevisionedNodeMixin*.

RevisionMixin adds the following attributes and methods to your class.

__parent_model__
Refers to the parent *RevisionedNodeMixin* subclass for which this model stores content revisions.

__tablename__
Autogenerated table name (parent table name with `'_revision'` suffixed).

node_id

node
Id and relationship to the parent model instance. The foreign key constraint refers to the parent model and *not* to the *Node* table to ensure that a revision cannot be accidentally attached to the wrong type of node.

status
Unique workflow status code for this node, `None` for archived revisions. A given revision code may be used by only one revision in a document, to indicate a revision currently flagged as Draft, Pending or Published.

user_id

user
The user who created this revision. In a multi-user editing environment, only one user can be tagged as the creator of the revision.

previous_id

previous
Id and relationship to previous revision that revision revises.

copy()
Copies this revision into a new revision attached to the same parent model. *copy()* is not content aware; your subclass will need to override *copy()* to copy contents:

```
class MyDocumentRevision(MyDocument.RevisionMixin, db.Model):
    content = db.Column(db.UnicodeText)

    def copy(self):
        revision = super(MyDocumentRevision, self).copy()
        revision.content = self.content
        return revision
```

Node registry

The node registry is a place to list the relationships between node types and their views.

Nodular does *not* provide a global instance of *NodeRegistry*. Since the registry determines what is available in an app, registries should be constructed as app-level globals.

class nodular.registry.**NodeRegistry**
Registry for node types and node views.

register_node(*model*, *view=None*, *itype=None*, *title=None*, *child_nodetypes=None*, *parent_nodetypes=None*)
Register a node.

Parameters

- **model** (*Node*) – Node model.
- **view** (NodeView) – View for this node type (optional).
- **itype** (*string*) – Register the node model as an instance type (optional).
- **title** (*string*) – Optional title for the instance type.
- **child_nodetypes** (*list*) – Allowed child nodetypes. None or empty implies no children allowed.
- **parent_nodetypes** (*list*) – Nodetypes that this node can be a child of.

The special value `'*'` in `child_nodetypes` implies that this node is a generic container. `'*'` in `parent_nodetypes` implies that this node can appear in any container that has `'*'` in `child_nodetypes`.

register_view(*nodetype*, *view*)
Register a view.

Parameters

- **nodetype** (*string*) – Node type that this view renders for.
- **view** (*NodeView*) – View class.

Node publisher

A node publisher translates between paths and URLs and publishes views for a given path. Typical usage:

```
from nodular import NodeRegistry, NodePublisher
from myapp import app, root, registry

assert isinstance(registry, NodeRegistry)
# Publish everything under /
publisher = NodePublisher(root, registry, '/')

@app.route('/<path:anypath>', methods=['GET', 'POST', 'PUT', 'DELETE'])
def publish_path(anypath):
    return publisher.publish(anypath)
```

class nodular.publisher.**NodePublisher**(*root*, *registry*, *basepath*, *urlpath=None*)
NodePublisher publishes node paths.

Parameters

- **root** – Root node for lookups (Node instance or integer primary key id).
- **registry** (*NodeRegistry*) – Registry for looking up views.
- **basepath** (*string*) – Base path to publish from, typically `'/'`.
- **urlpath** (*string*) – URL path to publish to, typically also `'/'`. Defaults to the `basepath` value.

NodePublisher may be instantiated either globally or per request, but requires a root node to query against. Depending on your setup, this may be available only at request time.

publish(*path*, *user=None*, *permissions=None*)
Publish a path using views from the registry.

Parameters

- **path** – Path to be published (relative to the initialized basepath).
- **user** – User we are rendering for (required for permission checks).
- **permissions** – Externally-granted permissions for this user.

Returns Result of the called view or `NotFound` exception if no view is found.

publish() uses *traverse()* to find a node to publish.

traverse(*path*, *redirect=True*)
Traverse to the node at the given path, returning the closest matching node and status.

Parameters

- **path** – Path to be traversed.
- **redirect** – If True (default), look for redirects when there's a partial match.

Returns Tuple of (status, node, path)

Return value `status` is one of `MATCH`, `REDIRECT`, `PARTIAL`, `NOROOT` or `GONE`. For an exact `MATCH`, `node` is the found node and `path` is `None`. For a `REDIRECT` or `PARTIAL` match, `node` is the closest matching node and `path` is the URL path to redirect to OR the remaining unmatched path. `NOROOT` implies the root node is missing. If redirects are enabled and a *NodeAlias* is found indicating a node is deleted, status is `GONE`.

traverse() does not require a registry since it does not look up views. *NodePublisher* may be initialized with `registry=None` if only used for traversal.

url_for(*node*, *action=u'view'*, *_external=False*, ***kwargs*)
Generates a URL to the given node with the view.

Parameters

- **node** – Node instance
- **endpoint** – the endpoint of the URL (name of the function)

class nodular.publisher.**TRAVERSE_STATUS**
Traversal status codes.

Node views

class nodular.view.**NodeView**(*node*, *user=None*, *permissions=None*)
Base class for node view handlers, to be initialized once per view render. Views are typically constructed like this:

```
class MyNodeView(NodeView):
    @NodeView.route('/')
    def index(self):
        return u'index view'

    @NodeView.route('/edit', methods=['GET', 'POST'])
    @NodeView.route('/', methods=['PUT'])
    @NodeView.requires_permission('edit', 'siteadmin')
    def edit(self):
        return u'edit view'

    @NodeView.route('/delete', methods=['GET', 'POST'])
    @NodeView.route('/', methods=['DELETE'])
    @NodeView.requires_permission('delete', 'siteadmin')
    def delete(self):
        return u'delete view'
```

Parameters

- **node** (*Node*) – Node that we are rendering a view for.
- **user** – User that the view is being rendered for.

static requires_permission(*permission*, **other*)

Decorator to enforce a permission requirement on a view.

Parameters

- **permission** (*string*) – Permission required to access this handler.
- **other** – Other permissions, any of which can be used to access this handler.

Available permissions are posted to `flask.g.permissions` for the lifetime of the request.

static route(*rule*, *endpoint=None*, *methods=None*, *defaults=None*)

Decorator for view handlers.

Parameters

- **rule** (*string*) – URL rule. See Werkzeug routing for syntax.
- **endpoint** (*string*) – Endpoint name, defaulting to method name.
- **methods** (*list*) – List of HTTP methods (default GET only).
- **defaults** (*dict*) – Default values to be passed to handler.

Exceptions

Exceptions raised by Nodular.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

B

C

E

G

I

M

N

P

Q

R

S

T

U

V