Nefertari Documentation

Release

Brandicted

Oct 05, 2017

Contents

Nefertari is a REST API framework for Pyramid that uses Elasticsearch for reads and either MongoDB or Postgres for writes.

Nefertari is fully production ready and actively maintained.

Source code: http://github.com/ramses-tech/nefertari

CHAPTER 1

Table of Content

Getting started

Create your project in a virtualenv directory (see the virtualenv documentation)

```
$ virtualenv my_project
$ source my_project/bin/activate
$ pip install nefertari
$ pcreate -s nefertari_starter my_project
$ cd my_project
$ pserve local.ini
```

Requirements

- Python 2.7, 3.3 or 3.4
- Elasticsearch for Elasticsearch-powered resources (see *models* and *requests*)
- Postgres or Mongodb or Your Data Store™

Tutorials

- For a more complete example of a Pyramid project using Nefertari, you can take a look at the Example Project.

Configuring Views

```
from nefertari.view import BaseView
from example_api.models import Story
```

```python
class StoriesView(BaseView):
    Model = Story

    def index(self):
        return self.get_collection_es()

    def show(self, **kwargs):
        return self.context

    def create(self):
        story = self.Model(**self._json_params)
        return story.save(self.request)

    def update(self, **kwargs):
        story = self.Model.get_item(
            id=kwargs.pop('story_id'), **kwargs)
        return story.update(self._json_params, self.request)

    def replace(self, **kwargs):
        return self.update(**kwargs)

    def delete(self, **kwargs):
        story = self.Model.get_item(
            id=kwargs.pop('story_id'), **kwargs)
        story.delete(self.request)

    def delete_many(self):
        es_stories = self.get_collection_es()
        stories = self.Model.filter_objects(es_stories)

        return self.Model._delete_many(stories, self.request)

    def update_many(self):
        es_stories = self.get_collection_es()
        stories = self.Model.filter_objects(es_stories)

        return self.Model._update_many(
            stories, self._json_params, self.request)
```

- `index()` called upon `GET` request to a collection, e.g. `/collection`
- `show()` called upon `GET` request to a collection-item, e.g. `/collection/<id>`
- `create()` called upon `POST` request to a collection
- `update()` called upon `PATCH` request to a collection-item
- `replace()` called upon `PUT` request to a collection-item
- `delete()` called upon `DELETE` request to a collection-item
- `update_many()` called upon `PATCH` request to a collection or filtered collection
- `delete_many()` called upon `DELETE` request to a collection or filtered collection

Polymorphic Views

Set `elasticsearch.enable_polymorphic_query = true` in your .ini file to enable this feature. Polymorphic views are views that return two or more comma-separated collections, e.g.'/api/<collection_1>,<collection_N>'. They are dynamic which means that they do not need to be defined in your code.

Other Considerations

It is recommended that your views reside in a package: In this case, each module of that package would contain all views of any given root-level route. Alternatively, ou can explicitly provide a view name, or a view class as a `view` keyword argument to `resource.add()` in your project's `main` function.

For singular resources: there is no need to define `index()`

Each view must define the following property: *Model*: model being served by the current view. Must be set at class definition for features to work properly. E.g.:

```python
from nefertari.view import BaseView
from example_api.models import Story

class StoriesView(BaseView):
    Model = Story
```

Optional properties: *_json_encoder*: encoder to encode objects to JSON. Database-specific encoders are available at `nefertari.engine.JSONEncoder`

Configuring Models

```python
from datetime import datetime
from nefertari import engine as eng
from nefertari.engine import BaseDocument


class Story(BaseDocument):
    __tablename__ = 'stories'

    _auth_fields = [
        'id', 'updated_at', 'created_at', 'start_date',
        'due_date', 'name', 'description']
    _public_fields = ['id', 'start_date', 'due_date', 'name']

    id = eng.IdField(primary_key=True)
    updated_at = eng.DateTimeField(onupdate=datetime.utcnow)
    created_at = eng.DateTimeField(default=datetime.utcnow)

    start_date = eng.DateTimeField(default=datetime.utcnow)
    due_date = eng.DateTimeField()

    name = eng.StringField(required=True)
    description = eng.TextField()
```

Database Backends

Nefertari implements database engines on top of two different ORMs: SQLAlchemy and MongoEngine. These two engines wrap the underlying APIs of each ORM and provide a standardized syntax for using either one, making it easy to switch between them with minimal changes. Each Nefertari engine is maintained in its own repository:

- nefertari-sqla github repository
- nefertari-mongodb github repository

Nefertari can either use Elasticsearch (*ESBaseDocument*) or the database engine itself (*BaseDocument*) for reads.

```
from nefertari.engine import ESBaseDocument


class Story(ESBaseDocument):
    (...)
```

or

```
from nefertari.engine import BaseDocument


class Story(BaseDocument):
    (...)
```

You can read more about *ESBaseDocument* and *BaseDocument* in the *Wrapper API* section below.

Wrapper API

Both database engines used by Nefertari implement a "Wrapper API" for developers who use Nefertari in their project. You can read more about either engine's in their respective documentation:

- nefertari-sqla documentation
- nefertari-mongodb documentation

BaseMixin Mixin with most of the API of *BaseDocument*. *BaseDocument* subclasses from this mixin.

BaseDocument Base for regular models defined in your application. Just subclass it to define your model's fields. Relevant attributes:

- *__tablename__*: table name (only required by nefertari-sqla)
- *_auth_fields*: String names of fields meant to be displayed to authenticated users.
- *_public_fields*: String names of fields meant to be displayed to non-authenticated users.
- *_hidden_fields*: String names of fields meant to be hidden but editable.
- *_nested_relationships*: String names of relationship fields that should be included in JSON data of an object as full included documents. If relationship field is not present in this list, this field's value in JSON will be an object's ID or list of IDs.

ESBaseDocument Subclass of *BaseDocument* instances of which are indexed on create/update/delete.

ESMetaclass Document metaclass which is used in *ESBaseDocument* to enable automatic indexation to Elasticsearch of documents.

get_document_cls(name) Helper function used to get the class of document by the name of the class.

JSONEncoder JSON encoder that should be used to encode output of views.

ESJSONSerializer JSON encoder used to encode documents prior indexing them in Elasticsearch.

relationship_fields Tuple of classes that represent relationship fields in specific engine.

is_relationship_field(field, model_cls) Helper function to determine whether *field* is a relationship field at *model_cls* class.

relationship_cls(field, model_cls) Return class which is pointed to by relationship field *field* from model *model_cls*.

Field Types

This is the list of all the available field types:

- BigIntegerField
- BinaryField
- BooleanField
- ChoiceField
- DateField
- DateTimeField
- DecimalField
- DictField
- FloatField
- ForeignKeyField (ignored/not required when using mongodb)
- IdField
- IntegerField
- IntervalField
- ListField
- PickleField
- Relationship
- SmallIntegerField
- StringField
- TextField
- TimeField
- UnicodeField
- UnicodeTextField

Authentication & Security

In order to enable authentication, add the `auth` paramer to your .ini file:

```
auth = true
```

Nefertari currently uses the default Pyramid "auth ticket" cookie mechanism.

Custom User Model

When authentication is enabled, Nefertari uses its own *User* model. This model has 4 fields by default: username, email, password and groups (list field with values 'admin' and 'user'). However, this model can be extanded.

```
from nefertari import engine as eng
from nefertari.authentication.models import AuthUserMixin
from nefertari.engine import BaseDocument


class User(AuthUserMixin, BaseDocument):
    __tablename__ = 'users'

    first_name = eng.StringField(max_length=50, default='')
    last_name = eng.StringField(max_length=50, default='')
```

Visible Fields in Views

You can control which fields to display by defining the following properties on your models:

_auth_fields Lists fields to be displayed to authenticated users.

_public_fields Lists fields to be displayed to all users including unauthenticated users.

_hidden_fields Lists fields to be hidden but remain editable (as long as user has permission), e.g. password.

Permissions

This section describes permissions used by nefertari, their relation to view methods and HTTP methods. These permissions should be used when defining ACLs.

To make things easier to grasp, let's imagine we have an application that defines a view which handles all possible requests under `/products` route. We are going to use this example to make permissions description more obvious.

Following lists nefertari permissions along with HTTP methods and view methods they correspond to:

view

- Collection GET (`GET /products`). View method `index`
- Item GET (`GET /products/1`) View method `show`
- Collection HEAD (`HEAD /products`). View method `index`
- Item HEAD (`HEAD /products/1`). View method `show`

create

- Collection POST (`POST /products`). View method `create`

update

- Collection PATCH (`PATCH /products`). View method `update_many`
- Collection PUT (`PUT /products`). View method `update_many`
- Item PATCH (`PATCH /products/1`). View method `update`
- Item PUT (`PUT /products/1`) View method `replace`

delete

- Collection DELETE (`DELETE /products`). View method `delete_many`
- Item DELETE (`DELETE /products/1`). View method `delete`

options

- Collection OPTIONS (`OPTIONS /products`). View method `collection_options`
- Item OPTIONS (`OPTIONS /products/1`). View method `item_options`

ACL API

For authorizing access to specific resources, Nefertari uses standard Pyramid access control lists. See the documentation on Pyramid ACLs to understand how to extend and customize them.

Considerations:

- An item will inherit its collection's permissions if the item's permissions are not specified in an ACL class
- If you create an ACL class for your document that does something like give the document.owner edit permissions, then you can't rely on this setting to be respected during collection operation. in other words, only if you walk up to the item via a URL will this permission setting be applied.

class `nefertari.acl.`**`AuthenticatedReadACL`**(*request*, *name=''*, *parent=None*)

Authenticated users ACL base class

Gives read access to all Authenticated users. Gives delete, create, update access to admin only.

class `nefertari.acl.`**`AuthenticationACL`**(*request*, *name=''*, *parent=None*)

Special ACL factory to be used with authentication views (login, logout, register, etc.)

Allows create, view and option methods to everyone.

class `nefertari.acl.`**`CollectionACL`**(*request*, *name=''*, *parent=None*)

Collection resource.

You must specify the `item_model`. It should be a nefertari.engine document class. It is the model class for collection items.

Define a `__acl__` attribute on this class to define the container's permissions, and default child permissions. Inherits its acl from the root, if no acl is set.

Override the *item_acl* method if you wish to provide custom acls for collection items.

Override the *item_db_id* method if you wish to transform the collection item db id, e.g. to support a `self` item on a user collection.

class `nefertari.acl.`**`Contained`**(*request*, *name=''*, *parent=None*)

Contained base class resource

Can inherit its acl from its parent.

class `nefertari.acl.`**`GuestACL`**(*request*, *name=''*, *parent=None*)

Guest level ACL base class

Gives read permissions to everyone.

`nefertari.acl.`**`authenticated_userid`**(*request*)

Helper function that can be used in `db_key` to support *self* as a collection key.

Advanced ACLs

For more advanced ACLs, you can look into using nefertari-guards in you project. This package stores ACLs at the object level, making it easier to build multi-tenant applications using a single data store.

CORS

To enable CORS headers, set the following lines in your .ini file:

```
cors.enable = true
cors.allow_origins = http://localhost
cors.allow_credentials = true
```

Event Handlers

Nefertari event handler module includes a set of events, maps of events, event handler predicates and helper function to connect it all together. All the objects are contained in `nefertari.events` module. Nefertari event handlers use Pyramid event system.

Events

`nefertari.events` defines a set of event classes inherited from `nefertari.events.RequestEvent`, `nefertari.events.BeforeEvent` and `nefertari.events.AfterEvent`.

There are two types of nefertari events:

- "Before" events, which are run after view class is instantiated, but before view method is run, and before request is processed. Can be used to edit the request.
- "After" events, which are run after view method has been called. Can be used to edit the response.

Check the API section for a full list of attributes/params events have.

Complete list of events:

- BeforeIndex
- BeforeShow
- BeforeCreate
- BeforeUpdate
- BeforeReplace
- BeforeDelete
- BeforeUpdateMany
- BeforeDeleteMany
- BeforeItemOptions
- BeforeCollectionOptions
- AfterIndex
- AfterShow

- AfterCreate
- AfterUpdate
- AfterReplace
- AfterDelete
- AfterUpdateMany
- AfterDeleteMany
- AfterItemOptions
- AfterCollectionOptions

All events are named after camel-cased name of view method they are called around and prefixed with "Before" or "After" depending on the place event is triggered from (as described above). E.g. event classed for view method `update_many` are called `BeforeUpdateMany` and `AfterUpdateMany`.

Before vs After

It is recommended to use `before` events to:

- Transform input
- Perform validation
- Apply changes to object that is being affected by request using `event.set_field_value` method

And `after` events to:

- Change DB objects which are not affected by request
- Perform notifications/logging
- Edit a response using `event.set_field_value` method

Note: if a field changed via `event.set_field_value` is not affected by the request, it will be added to `event.fields` which will make any field processors attached to this field to be triggered, if they are run after this method call (connected to events after handler that performs method call).

Predicates

nefertari.events.ModelClassIs Available under `model` param when connecting event handlers, it allows to connect event handlers on per-model basis. When event handler is connected using this predicate, it will only be called when `view.Model` is the same class or subclass of this param value.

Utilities

nefertari.events.subscribe_to_events Helper function that allows to connect event handler to multiple events at once. Supports `model` event handler predicate param. Available at `config.subscribe_to_events`. Subscribers are run in order connected.

nefertari.events.BEFORE_EVENTS Map of `{view_method_name: EventClass}` of "Before" events. E.g. one of its elements is `'index': BeforeIndex`.

nefertari.events.AFTER_EVENTS Map of `{view_method_name: EventClass}` of "AFter" events. E.g. one of its elements is `'index': AfterIndex`.

nefertari.events.silent Decorator which marks view class or view method as "silent". Silent view classes and methods don't fire events. In the example below, view `ItemsView` won't fire any event. `UsersView` won't fire `BeforeIndex` and `AfterIndex` events but `BeforeShow` and `AfterShow` events will be fired.

```
from nefertari import view, events


@events.silent
class ItemsView(view.BaseView):
    ...


class UsersView(view.BaseView):

    @events.silent
    def index(self):
        ...

    def show(self):
        ...
```

nefertari.events.trigger_instead Decorator which allows view method to trigger another event instead of the default one. In the example above collection GET requests (`UsersView.index`) will trigger the event which corresponds to item PATCH (`update`).

```
from nefertari import view, events


class UsersView(view.BaseView):

    @events.trigger_instead('update')
    def index(self):
        ...
```

Example

We will use the following example to demonstrate how to connect handlers to events. This handler logs `request` to the console.

```
import logging
log = logging.getLogger(__name__)


def log_request(event):
    log.debug(event.request.body)
```

We can connect this handler to any of Nefertari events of any requests. E.g. lets log all collection POST after requests are made (view `create` method):

```
from nefertari import events


config.subscribe_to_events(
    log_request, [events.AfterCreate])
```

Or, if we wanted to limit the models for which this handler should be called, we can connect it with a `model` predicate:

```
from nefertari import events
from .models import User


config.subscribe_to_events(
    log_request, [events.AfterCreate],
    model=User)
```

This way, `log_request` event handler will only be called when collection POST request comes at an endpoint which handles the `User` model.

Whenever the response has to be edited, `after` events with `event.set_field_value` must be used. If the response contains a single item, calling this method will change this single item's field. Otherwise field will be changed for all the objects in the response. To edit the response meta, you can access `event.response` directly. `event.response` is a view method response serialized into `dict`.

E.g. if we want to hide user passwords from response on collection and item GET:

```
from nefertari import events
from .models import User

def hide_user_passwords(event):
    # Hide 'password' field
    event.set_field_value('password', 'VERY_SECRET')

config.subscribe_to_events(
    hide_user_passwords,
    [events.AfterIndex, events.AfterShow],
    model=User)
```

API

class `nefertari.events.`**`RequestEvent`**(*model*, *view*, *fields=None*, *field=None*, *instance=None*, *response=None*)

Nefertari request event.

Parameters

- **`model`** – Model class affected by the request
- **`view`** – View instance which will process the request. Some useful attributes are: request, _json_params, _query_params. Change _json_params to edit data used to create/update objects and _query_params to edit data used to query database.
- **`fields`** – Dict of all fields from request.json. Keys are fields names and values are nefertari.utils.FieldData instances. If request does not have JSON body, value will be an empty dict.
- **`field`** – Changed field object. This field is set/changed in FieldIsChanged subscriber predicate. Do not use this field to determine what event was triggered when same event handler was registered with and without field predicate.
- **`instance`** – Object instance affected by request. Used in item requests only(item GET, PATCH, PUT, DELETE). Should be used to read data only. Changes to the instance may result in database data inconsistency.
- **`response`** – Return value of view method serialized into dict. E.g. if view method returns "1", value of event.response will be "1". Is None in all "before" events. Note that is not a

Pyramid Response instance but the value returned by view method. May be useful to access newly created object on "create" action if it is returned by view method.

class `nefertari.events.`**`ModelClassIs`**(*model*, *config*)

Subscriber predicate to check event.model is the right model.

Example: config.add_subscriber(func, event, model=ModelCls)

`nefertari.events.`**`subscribe_to_events`**(*config*, *subscriber*, *events*, *model=None*)

Helper function to subscribe to group of events.

Parameters

- **`config`** – Pyramid contig instance.
- **`subscriber`** – Event subscriber function.
- **`events`** – Sequence of events to subscribe to.
- **`model`** – Model predicate value.

`nefertari.events.`**`silent`**(*obj*)

Mark view method or class as "silent" so events won't be fired.

Should be used as decorator on view classes or methods.

Parameters **`obj`** – Any object that allows attributes assignment. Should be either view method or view class.

`nefertari.events.`**`trigger_instead`**(*event_action*)

Specify action name to change event triggered by view method.

In the example above `MyView.index` method will trigger before/after `update` events.

```
class MyView(BaseView):
    @events.trigger_instead('update')
    def index(self):
        (...)
```

Parameters **`event_action`** – Event action name which should be triggered instead of default one.

Field Processors

Nefertari allows to define functions that accept field data and return modified field value, may perform validation or perform other actions related to field.

These functions are called "field processors". They are set up per-field and are called when request comes into application that modifies the field for which processor is set up (when the field is present in the request JSON body).

Setup

`nefertari.events.add_field_processors` is used to connect processors to fields. This function is accessible through Pyramid Configurator instance. Processors are called in the order in which they are defined. Each processor must return the processed value which is used as input for the successive processor (if such processor exists). `nefertari.events.add_field_processors` expects the following parameters:

processors Sequence of processor functions

model Model class for field if which processors are registered

field Field name for which processors are registered

Keyword Arguments

new_value New value of of field

instance Instance affected by request. Is None when set of items is updated in bulk and when item is created. `event.response` may be used to access newly created object, if object is returned by view method.

field Instance of nefertari.utils.data.FieldData instance containing data of changed field.

request Current Pyramid Request instance

model Model class affected by request

event Underlying event object. Should be used to edit other fields of instance using `event.set_field_value(field_name, value)`

Example

We will use the following example to demonstrate how to connect fields to processors. This processor lowercases values that are passed to it.

```
# processors.py
def lowercase(**kwargs):
    return kwargs['new_value'].lower()
```

```
# models.py
from nefertari import engine


class Item(engine.BaseDocument):
    __tablename__ = 'stories'
    id = engine.IdField(primary_key=True)
    name = engine.StringField(required=True)
```

We want to make sure `Item.name` is always lowercase, we can connect `lowercase` to `Item.name` field using `nefertari.events.add_field_processors` like this:

```
# __init__.py
from .models import Item
from .processors import lowercase


# Get access to Pyramid configurator
...

config.add_field_processors([lowercase], model=Item, field='name')
```

`lowercase` processor will be called each time application gets a request that passes `Item.name`

You can use the `event.set_field_value` helper method to edit other fields from within a processor. E.g. assuming we had the fields `due_date` and `days_left` and we connected the processor defined below to the field `due_date`, we can update `days_left` from within that same processor:

```
from .helpers import parse_data
from datetime import datetime


def calculate_days_left(**kwargs):
    parsed_date = parse_data(kwargs['new_value'])
    days_left = (parsed_date-datetime.now()).days
    event = kwargs['event']
    event.set_field_value('days_left', days_left)
    return kwargs['new_value']
```

Note: if a field changed via `event.set_field_value` is not affected by request, it will be added to `event.fields` which will make any field processors which are connected to this field to be triggered, if they are run after this method call (connected to events after handler that performs method call).

E.g. if in addition to the above `calculate_days_left` processor we had another processor for the `days_left` field, `calculate_days_left` will make the `days_left` processor run because `event.set_field_value` is called from within `calculate_days_left` field and therefor `days_left` is considered "updated/changed".

API

class `nefertari.events.`**`FieldIsChanged`**(*field*, *config*)

Subscriber predicate to check particular field is changed.

Used to implement field processors.

`nefertari.events.`**`add_field_processors`**(*config*, *processors*, *model*, *field*)

Add processors for model field.

Under the hood, regular nefertari event subscribed is created which calls field processors in order passed to this function.

Processors are passed following params:

- **new_value**: New value of of field.
- **instance**: Instance affected by request. Is None when set of items is updated in bulk and when item is created.
- **field**: Instance of nefertari.utils.data.FieldData instance containing data of changed field.
- **request**: Current Pyramid Request instance.
- **model**: Model class affected by request.
- **event**: Underlying event object.

Each processor must return processed value which is passed to next processor.

Parameters

- **`config`** – Pyramid Congurator instance.
- **`processors`** – Sequence of processor functions.
- **`model`** – Model class for field if which processors are registered.
- **`field`** – Field name for which processors are registered.

Making requests

Query syntax

Query parameters can be used on either GET, PATCH, PUT or DELETE requests.

url parameter	description
`_m=<method>`	tunnel any http method using GET, e.g. _m=POST[1]
`_limit=<n>`	limit the returned collection to <n> results (default: 20, max limit: 100 for unauthenticated users)
`_sort=<field_name>`	sort collection by <field_name>
`_start=<n>`	start collection from the <n>th resource
`_page=<n>`	start collection at page <n> (n * _limit)
`_fields=<field_lis`	display only specific fields, use `-` before field names to exclude those fields, e.g. `_fields=-descripton`

Query syntax for Elasticsearch

Additional parameters are available when using an Elasticsearch-enabled collection (see **ESBaseDocument** in the *Wrapper API* section of this documentation).

url parameter	description
`<field_name>=<keywords>`	to filter a collection using full-text search on <field_name>, ES operators[2] can be used, e.g. `?title=foo AND bar`
`<field_name>=(!<keywords>)`	to filter a collection using negative search
`q=<keywords>`	to filter a collection using full-text search on all fields
`_search_fields=<field_lis`	use with `?q=<keywords>` to restrict search to specific fields
`_refresh_index=true`	to refresh the ES index after performing the operation[3]
`_aggregations.<dot_notation_object>`	to use ES search aggregations, e.g. `?_aggregations.my_agg.terms.field=tag`[4]

Updating listfields

Items in listfields can be removed using "-" prefix.

```
$ curl -XPATCH 'http://localhost:6543/api/<collection>/<id>' -d '{
    "<list_field_name>": [-<item>]
}
'
```

Items can be both added and removed at the same time.

```
$ curl -XPATCH 'http://localhost:6543/api/<collection>/<id>' -d '{
    "<list_field_name>": [<item_to_add>,-<item_to_remove>]
```

[1] Set `enable_get_tunneling = true` in your .ini file to enable this feature. To update listfields and dictfields, you can use the following syntax: `_m=PATCH&<listfield>.<value>&<dictfield>.<key>=<value>`

[2] The full syntax of Elasticsearch querying is beyond the scope of this documentation. You can read more on the Elasticsearch Query String Query documentation to do things like fuzzy search: `?name=fuzzy~` or date range search: `?date=[2015-01-01 TO *]`

[3] Set `elasticsearch.enable_refresh_query = true` in your .ini file to enable this feature. This parameter only works with POST, PATCH, PUT and DELETE methods. Read more on Elasticsearch Bulk API documentation.

[4] Set `elasticsearch.enable_aggregations = true` in your .ini file to enable this feature. You can also use the short name *_aggs*. Read more on Elasticsearch Aggregations.

```
}
'
```

Listfields can be emptied by setting their value to “” or null.

```
$ curl -XPATCH 'http://localhost:6543/api/<collection>/<id>' -d '{
    "<list_field_name>": ""
}
'
```

Updating collections

If update_many() is defined in your view, you will be able to update a single field across an entire collection or a filtered collection. E.g.

```
$ curl -XPATCH 'http://localhost:6543/api/<collection>?q=<keywords>' -d '{
    "<field_name>": "<new_value>"
}
'
```

Deleting collections

Similarly, if delete_many() is defined, you will be able to delete entire collections or filtered collections. E.g.

```
$ curl -XDELETE 'http://localhost:6543/api/<collection>?_missing_=<field_name>'
```

Development Tools

Indexing in Elasticsearch

`nefertari.index` console script can be used to manually (re-)index models from your database engine to Elasticsearch.

You can run it like so:

```
$ nefertari.index --config local.ini --models Model
```

The available options are:

--config	specify ini file to use (required)
--models	list of models to index. Models must subclass ESBaseDocument.
--params	URL-encoded parameters for each module
--quiet	“quiet mode” (surpress output)
--index	Specify name of index. E.g. the slug at the end of http://localhost:9200/example_api
--chunk	Index chunk size
--force	Force re-indexation of all documents in database engine (defaults to False)

Importing bulk data

`nefertari.post2api` console script can be used to POST data to your api. It may be useful to import data in bulk, e.g. mock data.

You can run it like so:

```
$ nefertari.post2api -f ./users.json -u http://localhost:6543/api/users
```

The available options are:

-f	specify a json file containing an array of json objects
-u	specify the url of the collection you wish to POST to

Why Nefertari?

Nefertari is a tool for making REST APIs using the Pyramid web framework.

Rationale

There are many other libraries that make writing REST APIs easy. Nefertari did not begin as a tool. It was extracted from the API powering Brandicted after almost two years of development. Part of this decision was based on thinking highly of using Elasticsearch-powered views for endpoints, after having had a great experience with it.

We wanted to build powerful REST APIs that are relatively opinionated but still flexible (in order to make easy things easy and hard things possible). We happened to need to use Postgres on a side project, but Brandicted's API only supported MongoDB.

Before extracting Nefertari and turning it into an open source project, we shopped around the Python ecosystem and tested every REST API library/framework to see what would allow us to be as lazy as possible and also allow our APIs to grow bigger over time.

The most convenient option was the beautiful flask-restless by Jeffrey Finkelstein. It depends on SQLAlchemy and does a really good job being super easy to use. We had some subjective reservations about using Flask because of globals and the fact that our closest community members happen to be Pyramid folks. Also, ditching Elasticsearch would have meant needing to write queries in views.

We were also inspired by pyramid-royal from our fellow Montreal Python colleague Hadrien David. He showed how traversal is a-ok for matching routes in a tree of resources, which is what REST should be anyway.

However, we had become quite used to the power of Elasticsearch and wanted to retain the option of using it as a first class citizen to power most GET views. Plus we were already using traversal without really thinking. Therefore we decided to add Postgres support via SQLA to our platform, and thus was born Nefertari.

Vision

To us, "REST API" means something like "HTTP verbs mapped to CRUD operations on resources described as JSON". We are not trying to do full-on HATEOAS to satisfy any academic ideal of the REST style. There are quite a few development tasks that can be abstracted away by using our simple definition. This might change in the future when there's more of an ecosystem around client libraries which would know how to browse/consume hypermedia.

By making assumptions about sane defaults, we can eliminate the need for boilerplate to do things like serialization, URL mapping, validation, authentication/authorization, versioning, testing, database queries in views, etc. The only things that should absolutely need to be defined are the resources themselves, and they should just know how to act in a RESTful way by default. They should be configurable to a degree and extendable in special cases. Contrast this idea with something like the Django Rest Framework where quite a number of things need to be laid out in order to create an endpoint.[1]

Nefertari is the meat and potatoes of our development stack. Her partner project, Ramses, is the seasoning/sugar/cherry on top! Ramses allows whole production-ready Nefertari apps to be generated at runtime from a simple YAML file specifying the endpoints desired. Check it out.

Changelog

- : Added ability to edit responses within 'after' event handlers
- #128: Build ES params when body provided
- #121: Fixed issue with nested resources referencing parents
- : Scaffold defaults to Pyramid 1.6.1
- : Added support for Pyramid 1.6.x
- #130: Added support for Elasticsearch 2.x
- : 'request.user' is now set to None when using 'auth = False'
- : Fixed a bug with GET '/auth/logout'
- : Added 'event.instance' to 'event' object to access newly created object (if object is returned by view method)
- : Added '_hidden_fields' model attribute to hide fields while remaining editable (e.g. password)
- : Nested relationships are now indexed in bulk in Elasticsearch
- : Removed unnecessary extra '__confirmation' parameter from PATCH/PUT/DELETE collection requests
- : Refactored field processors to use the new event system
- : Event system is now crud-based as opposed to db-based
- : Fixed nested relationships not respecting '_auth_fields'
- : Fixed a bug causing polymorchic collections to always return 403
- : Added support for 'nefertari-guards'
- : Simplified ACLs (refactoring)
- : Fixed a bug allowing to update hidden fields
- : Fixed an error preventing RelationshipFields' backrefs to be set as _nested_relationships
- : Fixed a bug when using reserved query params with GET tunneling
- : Fixed '_self' param for `/api/users/self` convience route
- : Routes can now have the same member/collection name. E.g. root.add('staff', 'staff', ...)
- : Model's save()/update()/delete()/_delete_many()/_update_many() methods now require self.request to be passed for '_refresh_index' parameter to work
- : Added the ability to apply processors on 'Relationship' fields and their backrefs
- : Renamed setting *debug* to *enable_get_tunneling*

[1] For the record, DRF is pretty badass and we have great respect for its vision and breadth, and the hard work of its community. Laying out a ton of boilerplate can be considered to fall into "flat is better than nested" or "explicit is better than implicit" and might be best for some teams.

- : Refactored authentication
- : Renamed field 'self' to '_self'
- : Fixed error response when aggregating hidden fields with `auth = true`, it now returns 403
- : Fixed *_count* querying not respecting `public_max_limit` .ini setting
- : Fixed 400 error returned when querying resources with id in another format than the id field used in URL schema, e.g. `/api/<collection>/<string_instead_of_integer>`, it now returns 404
- : Fixed duplicate records when querying ES aggregations by '_type'
- : Fixed formatting error when using *_fields* query parameter
- : Fixed 40x error responses returning html, now all responses are json-formatted
- : Fixed bug with full-text search ('?q=') when used in combination with field search ('&<field>=')
- : Removed unused 'id' field from 'AuthUserMixin'
- : Fixed sorting by 'id' when two ES-based models have two different 'id' field types
- : Added support for Elasticsearch polymorphic collections accessible at `/api/<collection_1>, <collection_N>`
- : Fixed response of http methods POST/PATCH/PUT not returning created/updated objects
- : Fixed errors in http methods HEAD/OPTIONS response
- : Fixed a bug when setting `cors.allow_origins = *`
- : Added support for Elasticsearch polymorphic collections accessible at `/api/<collection_1>, <collection_N>`
- : Added ability to empty listfields by setting them to "" or null
- : Reworked ES bulk queries to use 'elasticsearch.helpers.bulk'
- : Added ES aggregations
- : Added python3 support
- : Fixed bug whereby *_count* would throw exception when authentication was enabled
- : Fixed bug with posting multiple new relations at the same time
- : Fixed race condition in Elasticsearch indexing by adding the optional '_refresh_index' query parameter
- : Fixed bug with Elasticsearch indexing of nested relationships
- : Fixed ES mapping error when values of field were all null
- : Fixed posting to singular resources e.g. `/api/users/<username>/profile`
- : Fixed PUT to replace all fields and PATCH to update some
- : Implemented API output control by field (apply_privacy wrapper)
- : Added ability to PATCH/DELETE collections
- : Fixed several issues related to Elasticsearch indexing
- : Increased test coverave
- : Step-by-step 'Getting started' guide
- : Fixed URL parsing for DictField and ListField values with _m=VERB options
- : Started adding tests

- : Added script to index Elasticsearch models
- : Improved docs
- : Listing on PyPI
- : Initial release after two years of development as 'Presto'. Now with database engines! Originally extracted and generalized from the Brandicted API which only used MongoDB.

Image credit: Wikipedia

Python Module Index

Index