
Ramses Documentation

Release

Brandicted

Oct 05, 2017

Contents

1	Table of Contents	3
1.1	Getting started	3
1.2	RAML Configuration	4
1.3	Defining Schemas	5
1.4	Fields	6
1.5	Event Handlers	9
1.6	Field processors	12
1.7	Relationships	13
1.8	Changelog	18

Ramses is a framework that generates a RESTful API using [RAML](#). It uses Pyramid and [Nefertari](#) which provides Elasticsearch / Posgres / MongoDB / Your Data Store™ -powered views. Using Elasticsearch enables [Elasticsearch-powered requests](#) which provides near real-time search.

Website: <http://ramses.tech>

Source code: <http://github.com/ramses-tech/ramses>

Getting started

1. Create your project in a virtualenv directory (see the [virtualenv documentation](#))

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

2. Tada! Start editing `api.raml` to modify the API and `items.json` for the schema.

Requirements

- Python 2.7, 3.3 or 3.4
- Elasticsearch (data is automatically indexed for near real-time search)
- Postgres or MongoDB or Your Data Store™

Examples

- For a more complete example of a Pyramid project using Ramses, you can take a look at the [Example Project](#).
- RAML can be used to generate an end-to-end application, check out [this example](#) using Ramses on the backend and RAML-javascript-client + BackboneJS on the front-end.

Tutorials

- [Create a REST API in Minutes With Pyramid and Ramses](#)


```
/items:
  securedBy: [read_only_users]
```

Enabling HTTP Methods

Listing an HTTP method in your resource definition is all it takes to enable such method.

```
/items:
  (...)
  post:
    description: Create an item
  get:
    description: Get multiple items
  patch:
    description: Update multiple items
  delete:
    description: delete multiple items

/{id}:
  displayName: One item
  get:
    description: Get a particular item
  delete:
    description: Delete a particular item
  patch:
    description: Update a particular item
```

You can link your schema definition for each resource by adding it to the post section.

```
/items:
  (...)
  post:
    (...)
    body:
      application/json:
        schema: !include schemas/items.json
```

Defining Schemas

JSON Schema

Ramses supports JSON Schema Draft 3 and Draft 4. You can read the official [JSON Schema documentation here](#).

```
{
  "type": "object",
  "title": "Item schema",
  "$schema": "http://json-schema.org/draft-04/schema",
  (...)
}
```

All Ramses-specific properties are prefixed with an underscore.

Showing Fields

If you've enabled authentication, you can list which fields to return to authenticated users in `_auth_fields` and to non-authenticated users in `_public_fields`. Additionally, you can list fields to be hidden but remain hidden (with proper permissions) in `_hidden_fields`.

```
{
  (...)
  "_auth_fields": ["id", "name", "description"],
  "_public_fields": ["name"],
  "_hidden_fields": ["token"],
  (...)
}
```

Nested Documents

If you use `Relationship` fields in your schemas, you can list those fields in `_nested_relationships`. Your fields will then become nested documents instead of just showing the `id`. You can control the level of nesting by specifying the `_nesting_depth` property, default is 1.

```
{
  (...)
  "_nested_relationships": ["relationship_field_name"],
  "_nesting_depth": 2
  (...)
}
```

Custom “user” Model

When authentication is enabled, a default “user” model will be created automatically with 4 fields: “username”, “email”, “groups” and “password”. You can extend this default model by defining your own “user” schema and by setting `_auth_model` to `true` on that schema. You can add any additional fields in addition to those 4 default fields.

```
{
  (...)
  "_auth_model": true,
  (...)
}
```

Fields

Types

You can set a field's type by setting the `type` property under `_db_settings`.

```
"created_at": {
  (...)
  "_db_settings": {
    "type": "datetime"
  }
}
```

This is a list of all available types:

- biginteger
- binary
- boolean
- choice
- date
- datetime
- decimal
- dict
- float
- foreign_key
- id_field
- integer
- interval
- list
- pickle
- relationship
- smallinteger
- string
- text
- time
- unicode
- unicodetext

Required Fields

You can set a field as required by setting the `required` property under `_db_settings`.

```
"password": {  
    (...)  
    "_db_settings": {  
        (...)  
        "required": true  
    }  
}
```

Primary Key

You can use an `id_field` in lieu of primary key.

```
"id": {
  (...)
  "_db_settings": {
    (...)
    "primary_key": true
  }
}
```

You can alternatively elect a field to be the primary key of your model by setting its `primary_key` property under `_db_settings`. For example, if you decide to use `username` as the primary key of your `User` model. This will enable resources to refer to that field in their url, e.g. `/api/users/john`

```
"username": {
  (...)
  "_db_settings": {
    (...)
    "primary_key": true
  }
}
```

Constraints

You can set a minimum and/or maximum length of your field by setting the `min_length`/`max_length` properties under `_db_settings`. You can also add a unique constraint on a field by setting the `unique` property.

```
"field": {
  (...)
  "_db_settings": {
    (...)
    "unique": true,
    "min_length": 5,
    "max_length": 50
  }
}
```

Default Value

You can set a default value for you field by setting the `default` property under `_db_settings`.

```
"field": {
  (...)
  "_db_settings": {
    (...)
    "default": "default value"
  }
},
```

The default value can also be set to a Python callable, e.g.

```
"datetime_field": {
  (...)
  "_db_settings": {
    (...)
    "default": "{{datetime.datetime.utcnow}}"
  }
}
```

```

    }
  },

```

Update Default Value

You can set an update default value for your field by setting the `onupdate` property under `_db_settings`. This is particularly useful to update 'datetime' fields on every updates, e.g.

```

"datetime_field": {
  (...)
  "_db_settings": {
    (...)
    "onupdate": "{{datetime.datetime.utcnow}}"
  }
},

```

List Fields

You can list the accepted values of any list or choice fields by setting the `choices` property under `_db_settings`.

```

"field": {
  (...)
  "_db_settings": {
    "type": "choice",
    "choices": ["choice1", "choice2", "choice3"],
    "default": "choice1"
  }
}

```

You can also provide the list/choice items' `item_type`.

```

"field": {
  (...)
  "_db_settings": {
    "type": "list",
    "item_type": "string"
  }
}

```

Other `_db_settings`

Note that you can pass any engine-specific arguments to your fields by defining such arguments in `_db_settings`.

Event Handlers

Ramses supports [Nefertari event handlers](#). Ramses event handlers also have access to [Nefertari's wrapper API](#) which provides additional helpers.

Setup

Writing Event Handlers

You can write custom functions inside your `__init__.py` file, then add the `@registry.add` decorator before the functions that you'd like to turn into CRUD event handlers. Ramses CRUD event handlers has the same API as Nefertari CRUD event handlers. Check Nefertari CRUD Events doc for more details on events API.

Example:

```
import logging
from ramses import registry

log = logging.getLogger('foo')

@registry.add
def log_changed_fields(event):
    changed = ['{}: {}'.format(name, field.new_value)
              for name, field in event.fields.items()]
    logger.debug('Changed fields: ' + ', '.join(changed))
```

Connecting Event Handlers

When you define event handlers in your `__init__.py` as described above, you can apply them on per-model basis. If multiple handlers are listed, they are executed in the order in which they are listed. Handlers should be defined in the root of JSON schema using `_event_handlers` property. This property is an object, keys of which are called “event tags” and values are lists of handler names. Event tags are composed of two parts: `<type>_<action>` whereby:

type Is either *before* or *after*, depending on when handler should run - before view method call or after respectively. You can read more about when to use [before vs after event handlers](#).

action Exact name of Nefertari view method that processes the request (action) and special names for authentication actions.

Complete list of actions:

- **index** - Collection GET
- **create** - Collection POST
- **update_many** - Collection PATCH/PUT
- **delete_many** - Collection DELETE
- **collection_options** - Collection OPTIONS
- **show** - Item GET
- **update** - Item PATCH
- **replace** - Item PUT
- **delete** - Item DELETE
- **item_options** - Item OPTIONS
- **login** - User login (POST /auth/login)
- **logout** - User logout (POST /auth/logout)
- **register** - User register (POST /auth/register)

- **set** - triggers on all the following actions: **create**, **update**, **replace**, **update_many** and **register**.

Example

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

```
import logging
from ramses import registry

log = logging.getLogger('foo')

@registry.add
def log_request(event):
    log.debug(event.view.request)
```

Assuming we had a JSON schema representing the model `User` and we want to log all collection GET requests on the `User` model after they are processed using the `log_request` handler, we would register the handler in the JSON schema like this:

```
{
  "type": "object",
  "title": "User schema",
  "$schema": "http://json-schema.org/draft-04/schema",
  "_event_handlers": {
    "after_index": ["log_request"]
  },
  ...
}
```

Other Things You Can Do

You can update another field's value, for example, increment a counter:

```
from ramses import registry

@registry.add
def increment_count(event):
    instance = event.instance or event.response
    counter = instance.counter
    incremented = counter + 1
    event.set_field_value('counter', incremented)
```

You can update other collections (or filtered collections), for example, mark sub-tasks as completed whenever a task is completed:

```
from ramses import registry
from nefertari import engine

@registry.add
def mark_subtasks_completed(event):
    if 'task' not in event.fields:
        return
```

```
completed = event.fields['task'].new_value
instance = event.instance or event.response

if completed:
    subtask_model = engine.get_document_cls('Subtask')
    subtasks = subtask_model.get_collection(task_id=instance.id)
    subtask_model._update_many(subtasks, {'completed': True})
```

You can perform more complex queries using Elasticsearch:

```
from ramses import registry
from nefertari import engine
from nefertari.elasticsearch import ES

@registry.add
def mark_subtasks_after_2015_completed(event):
    if 'task' not in event.fields:
        return

    completed = event.fields['task'].new_value
    instance = event.instance or event.response

    if completed:
        subtask_model = engine.get_document_cls('Subtask')
        es_query = 'task_id:{} AND created_at:[2015 TO *]'.format(instance.id)
        subtasks_es = ES(subtask_model.__name__).get_collection(_raw_terms=es_query)
        subtasks_db = subtask_model.filter_objects(subtasks_es)
        subtask_model._update_many(subtasks_db, {'completed': True})
```

Field processors

Ramses supports [Nefertari field processors](#). Ramses field processors also have access to [Nefertari's wrapper API](#) which provides additional helpers.

Setup

To setup a field processor, you can define the `_processors` property in your field definition (same level as `_db_settings`). It should be an array of processor names to apply. You can also use the `_backref_processors` property to specify processors for backref field. For backref processors to work, `_db_settings` must contain the following properties: `document`, `type=relationship` and `backref_name`.

```
"username": {
    ...
    "_processors": ["lowercase"]
},
...
```

You can read more about processors in [Nefertari's field processors documentation](#) including the list of keyword arguments passed to processors.

Example

If we had following processors defined:

```
from .my_helpers import get_stories_by_ids

@registry.add
def lowercase(**kwargs):
    """ Make :new_value: lowercase """
    return (kwargs['new_value'] or '').lower()

@registry.add
def validate_stories_exist(**kwargs):
    """ Make sure added stories exist. """
    story_ids = kwargs['new_value']
    if story_ids:
        # Get stories by ids
        stories = get_stories_by_ids(story_ids)
        if not stories or len(stories) < len(story_ids):
            raise Exception("Some of provided stories do not exist")
    return story_ids
```

```
# User model json
{
  "type": "object",
  "title": "User schema",
  "$schema": "http://json-schema.org/draft-04/schema",
  "properties": {
    "stories": {
      "_db_settings": {
        "type": "relationship",
        "document": "Story",
        "backref_name": "owner"
      },
      "_processors": ["validate_stories_exist"],
      "_backref_processors": ["lowercase"]
    },
    ...
  }
}
```

Notes:

- `validate_stories_exist` processor will be run when request changes `User.stories` value. The processor will make sure all of story IDs from request exist.
- `lowercase` processor will be run when request changes `Story.owner` field. The processor will lowercase new value of the `Story.owner` field.

Relationships

Basics

Relationships in Ramses are used to represent One-To-Many(o2m) and One-To-One(o2o) relationships between objects in database.

To set up relationships fields of types `foreign_key` and `relationship` are used. `foreign_key` field is not required when using `nefertari_mongodb` engine and is ignored.

For this tutorial we are going to use the example of users and stories. In this example we have a `OneToMany` relationship between `User` and `Story`. One user may have many stories but each story has only one owner. Check the end of the tutorial for the complete example RAML file and schemas.

Example code is the very minimum needed to explain the subject. We will be referring to the examples along all the tutorial.

Field “type”: “relationship”

Must be defined on the *One* side of `OneToOne` or `OneToMany` relationship (`User` in our example). Relationships are created as `OneToMany` by default.

Example of using `relationship` field (defined on `User` model in our example):

```
"stories": {
  "_db_settings": {
    "type": "relationship",
    "document": "Story",
    "backref_name": "owner"
  }
}
```

Required params:

type String. Just `relationship`.

document String. Exact name of model class to which relationship is set up. To find out the name of model use singularized uppercased version of route name. E.g. if we want to set up relationship to objects of `/stories` then the `document` arg will be `Story`.

backref_name String. Name of *back reference* field. This field will be auto-generated on model we set up relationship to and will hold the instance of model we are defining. In our example, field `Story.owner` will be generated and it will hold instance of `User` model to which story instance belongs. **Use this field to change relationships between objects.**

Field “type”: “foreign_key”

This represents a Foreign Key constraint in SQL and is only required when using `nefertari_sqla` engine. It is used in conjunction with the `relationship` field, but is used on the model that `relationship` refers to. For example, if the `User` model contained the `relationship` field, than the `Story` model would need a `foreign_key` field.

Notes:

- This field is not required and is ignored when using `nefertari_mongodb` engine.
- Name of the `foreign_key` field does not depend on relationship params in any way.
- This field **MUST NOT** be used to change relationships. This field only exists because it is required by SQLAlchemy.

Example of using `foreign_key` field (defined on `Story` model in our example):

```
"owner_id": {
  "_db_settings": {
    "type": "foreign_key",
    "ref_document": "User",
  }
}
```

```

    "ref_column": "user.username",
    "ref_column_type": "string"
  }
}

```

Required params:

type String. Just `foreign_key`.

ref_document String. Exact name of model class to which foreign key is set up. To find out the name of model use singularized uppercased version of route name. E.g. if we want to set up foreign key to objects of `/user` then the `ref_document` arg will be `User`.

ref_column String. Dotted name/path to `ref_document` model's primary key column. `ref_column` is the lowercased name of model we refer to in `ref_document` joined by a dot with the exact name of its primary key column. In our example this is `"user.username"`.

ref_column_type String. Ramses field type of `ref_document` model's primary key column specified in `ref_column` parameter. In our example this is `"string"` because `User.username` is `"type": "string"`.

One to One relationship

To create `OneToOne` relationships, specify `"uselist": false` in `_db_settings` of relationship field. When setting up One-to-One relationship, it doesn't matter which side defines the relationship field.

E.g. if we had `Profile` model and we wanted to set up One-to-One relationship between `Profile` and `User`, we would have to define a regular `foreign_key` field on `Profile`:

```

"user_id": {
  "_db_settings": {
    "type": "foreign_key",
    "ref_document": "User",
    "ref_column": "user.username",
    "ref_column_type": "string"
  }
}

```

and relationship field with `"uselist": false` on `User`:

```

"profile": {
  "_db_settings": {
    "type": "relationship",
    "document": "Profile",
    "backref_name": "user",
    "uselist": false
  }
}

```

This relationship could also be defined the other way but with the same result: `foreign_key` field on `User` and relationship field on `Profile` pointing to `User`.

Multiple relationships

Note: This part is only valid(required) for `nefertari_sqla` engine, as `nefertari_mongodb` engine does not use `foreign_key` fields.

If we were to define multiple relationships from model A to model B, each relationship must have a corresponding `foreign_key` defined. Also you must use a `foreign_keys` parameter on each relationship field to specify which `foreign_key` each relationship uses.

E.g. if we were to add new relationship field `User.assigned_stories`, relationship fields on `User` would have to be defined like this:

```
"stories": {
  "_db_settings": {
    "type": "relationship",
    "document": "Story",
    "backref_name": "owner",
    "foreign_keys": "Story.owner_id"
  }
},
"assigned_stories": {
  "_db_settings": {
    "type": "relationship",
    "document": "Story",
    "backref_name": "assignee",
    "foreign_keys": "Story.assignee_id"
  }
}
```

And fields on `Story` like so:

```
"owner_id": {
  "_db_settings": {
    "type": "foreign_key",
    "ref_document": "User",
    "ref_column": "user.username",
    "ref_column_type": "string"
  }
},
"assignee_id": {
  "_db_settings": {
    "type": "foreign_key",
    "ref_document": "User",
    "ref_column": "user.username",
    "ref_column_type": "string"
  }
}
```

Complete example

example.raml

```
##RAML 0.8
---
title: Example REST API
documentation:
  - title: Home
    content: |
      Welcome to the example API.
baseUri: http://{host}:{port}/{version}
version: v1
```

```

/stories:
  displayName: All stories
  get:
    description: Get all stories
  post:
    description: Create a new story
    body:
      application/json:
        schema: !include story.json
/{id}:
  displayName: One story
  get:
    description: Get a particular story

/users:
  displayName: All users
  get:
    description: Get all users
  post:
    description: Create a new user
    body:
      application/json:
        schema: !include user.json
/{username}:
  displayName: One user
  get:
    description: Get a particular user

```

user.json

```

{
  "type": "object",
  "title": "User schema",
  "$schema": "http://json-schema.org/draft-04/schema",
  "required": ["username"],
  "properties": {
    "username": {
      "_db_settings": {
        "type": "string",
        "primary_key": true
      }
    },
    "stories": {
      "_db_settings": {
        "type": "relationship",
        "document": "Story",
        "backref_name": "owner"
      }
    }
  }
}

```

story.json

```

{
  "type": "object",
  "title": "Story schema",
  "$schema": "http://json-schema.org/draft-04/schema",

```

```
"properties": {
  "id": {
    "_db_settings": {
      "type": "id_field",
      "primary_key": true
    }
  },
  "owner_id": {
    "_db_settings": {
      "type": "foreign_key",
      "ref_document": "User",
      "ref_column": "user.username",
      "ref_column_type": "string"
    }
  }
}
```

Changelog

- : Scaffold defaults to Pyramid 1.6.1
- #99: Use ACL mixin from nefertari-guards (if enabled)
- #107: Fixed issue with hyphens in resource paths
- : Scaffold defaults to Pyramid 1.6.1
- #99: Use ACL mixin from nefertari-guards (if enabled)
- #88: Reworked the creation of related/auth_model models, order does not matter anymore
- : Fixed a bug using 'required' '_db_settings' property on 'relationship' field
- : Added support for the property '_nesting_depth' in schemas
- : ACL permission names in RAML now match real permission names instead of http methods
- : Simplified field processors, '_before_processors' is now called '_processors', removed '_after_processors'
- : Added support for Nefertari event handlers
- : Added support for Nefertari '_hidden_fields'
- : Added support for 'nefertari-guards'
- : Simplified ACLs (refactoring)
- : Error response bodies are now returned as JSON
- : Prefixed all Ramses schema properties by an underscore: '_auth_fields', '_public_fields', '_nested_relationships', '_auth_model', '_db_settings'
- : Properties 'type' and 'required' are now under '_db_settings'
- : Renamed schema's 'args' property to '_db_settings'
- : Added support for relationship processors and backref relationship processors ('backref_after_validation'/'backref_before_validation')
- : Field name and request object are now passed to field processors under 'field' and 'request' kwargs respectively
- : Renamed setting 'debug' to 'enable_get_tunneling'

- : Renamed setting 'ramses.auth' to 'auth'
- : Boolean values in RAML don't have to be strings anymore (previous limitation of pyraml-parser)
- : Fixed a limitation preventing collection names to use nouns that do not have plural forms
- : Fixed processors not applied on fields of type 'list' and type 'dict'
- : Added support for 'onupdate' field argument
- : Added support for callables in 'default' field argument
- : RAML is now parsed using ramlfications instead of pyraml-parser
- : Added support for JSON schema draft 04
- : Added support for 'onupdate' field argument
- : Added support for callables in 'default' field argument
- : Added python3 support
- : Forward compatibility with nefertari releases
- : Fixed race condition in Elasticsearch indexing
- : Fixed password minimum length support by adding before and after validation processors
- : Fixed custom processors
- : Fixed login issue
- : Fixed limiting fields to be searched
- : Add support for custom auth model
- : Add support for processors in schema definition
- : Added support for securitySchemes, authentication (Pyramid 'auth ticket') and ACLs
- : ES views now read from ES on update/delete_many
- : Improved docs
- : Added unit tests
- : Added several display options to schemas
- : Ramses could not be used in an existing Pyramid project
- : Initial release!



Image credit: Wikipedia