Python Rest Api Framework Documentation

Release 0.1

Yohann Gabory

May 02, 2017

Contents

Python REST API framework is a set of utilities based on werkzeug to easily build Restful API with a MVC pattern. Main features includes: Pagination, Authentication, Authorization, Filters, Partials Response, Error handling, data validators, data formaters... and more...

Contents:

CHAPTER 1

What is Python REST API Framework

Python REST API framework is a set of utilities based on werkzeug to easily build Restful API. It keep a clean codebase and is easy to configure and extends.

It does not decide how you want to render your data, or where they lives, or other decisions.

Instead it give you a good start with an extensible architecture to build your own API.

Python REST API Framework has not been create for the usual case. Instead it give you some hook to your very special ressource provider, your very special view management and your very special way of displaying data.

Python REST API Framework is fully REST compilant; It implement the common verbs:

- GET
- POST
- UPDATE
- DELETE
- HEAD

It also implement:

- PAGINATION
- AUTHENTICATION
- RATE-LIMIT
- DATA VALIDATION
- PARTIAL RESPONSE

Architecture

Python REST API Framework is base on the MVC pattern. You define some endpoints defining a Ressource, a Controller and a View with a set of options to configure them.

Controller

Manage the way you handle request. Controller create the urls endpoints for you. List, Unique and Autodocumented endpoints.

Controller also manage pagination, formaters, authentication, authorization, rate-limit and allowed method.

DataStore

Each method of a Controller call the DataStore to interact with data. The DataStore must be able to retreive data from a ressource.

Each datastore act on a particular type of ressource (database backend, api backend, csv backend etc...). It must be able to validate data, create new ressources, update existing ressources, manage filters and pagination.

Optional configuration option, that can be unique for a particular datastore like Ressource level validation (unique together and so), ForeignKey management...

View

Views defines how the data must be send to the client. It send a Response object and set the needed headers, mime-type and other presentation options like formaters.

How To use it

To create as many endpoint as you need. Each endpoints defining a ressource, a controller and a view. Then add them to the `rest_api_framework.controllers.WSGIDispatcher`

See *QuickStart* for an example or the *Tutorial: building an adressebook API* for the whole picture.

QuickStart

A Simple API

For this example, we will use a python list containing dicts. This is our data:

```
ressources = [
    {"name": "bob",
    "age": a,
    "id": a
    } for a in range(100)
    ]
```

Then we have to describe this ressource. To describe a ressouce, you must create a Model class inheriting from base Model class:

```
from rest_api_framework import models

class ApiModel(models.Model):

    fields = [models.IntegerField(name="age", required=True),
              models.StringField(name="name", required=True),
              models.PkField(name="id")
              ]
```

Each Field contain validators. When you reuse an existing Field class you get his validators for free.

There is already a datastore to handle this type of data: PythonListDataStore. We can reuse this store:

```
from rest_api_framework.datastore import PythonListDataStore
```

then we need a Controller class to hanlde our API:

```
from rest_api_framework.controllers import Controller
```

and a view to render our data

```
from rest_api_framework.views import JsonResponse


class ApiApp(Controller):
    ressource = {
        "ressource_name": "address",
        "ressource": ressources,
        "model": ApiModel,
        "datastore": PythonListDataStore
        }

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"],
        }

    view = {"response_class": JsonResponse}
```

A controller is build with 3 dicts:

Ressource

Ressource define your data. Where are your data ? How can they be accessed ? What they look likes?

- **ressource_name: will be used to build the url endpoint to your** ressource.
- **ressource: where your ressource lies.this argument tell the** datastore how they can be accessed. It can be the database name and the database table for a SQL datastore or the url endpoint to a distant API for exemple.
- **model: describe how your data look like. Wich field it show, how** to validate data and so on.
- **datastore: the type of your data. There is datastore for simple** Python list of dict and SQLite datastore. They are exemple on how to build your own datastore depending on your needs.

Controller

The controller define the way your data should be accessed. Should the results be paginated ? Authenticated ? Rate-limited ? Wich it the verbs you can use on the resource ? and so on.

- **list_verbs: define the verbs you can use on the main endpoint of** your ressource. If you dont' use "POST", a user cannot create new ressources on your datastore.

- **unique_verbs: define the verbs you can use on the unique** identifier of the ressource. actions depending on the verbs follows the REST implementation: PUT to modify an existing ressource, DELETE to delete a ressource.

View

view define How your ressoources should be rendered to the user. It can be a Json format, XML, or whatever. It can also render pagination token, first page, last page, number of objects and other usefull informations for your users.

- response_class: the response class you use to render your data.

To test you application locally, you can add:

```
if __name__ == '__main__':
    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([ApiApp])
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

then type "python app.py" and your API is up and running

Options

Each of this dicts can take an optional parameter: "option". This parameter is a dict containing all the options you want to use with either the datastore, the view or the controller.

You can learn more about optional parameters in the documentation of each topic : datastore, view, controller

Using a database

Instead of using a python dict, you may want to actualy save your data in a database. To do so, you just have to change your datastore and define your ressources in a way SQL datastore can understand.

SQLiteDataStore use sqlite3 as database backend. ressources will be a dict with database name and table name. The rest of the configuration is the same as with the PythonListDataStore.

Note: if the sqlite3 database does not exist, REST API Framework create it for you

```
from rest_api_framework.datastore import SQLiteDataStore
from rest_api_framework.controllers import Controller
from rest_api_framework.views import JsonResponse
from rest_api_framework import models
from rest_api_framework.pagination import Pagination

class ApiModel(models.Model):
    fields = [models.StringField(name="message", required=True),
              models.StringField(name="user", required=True),
              models.PkField(name="id", required=True),
              ]

class ApiApp(Controller):
    ressource = {
       "ressource_name": "tweets",
       "ressource": {"name": "twitter.db", "table": "tweets"},
        "datastore": SQLiteDataStore,
        "model": ApiModel
    }
    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"]
        "options": {"pagination": Pagination(20)}
    }
    view = {"response_class": JsonResponse}


if __name__ == '__main__':
    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([ApiApp])
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

Where to go from here

- *Tutorial: building an adressebook API*
- *REST API Framework API references*

CHAPTER 2

Tutorial: building an adressebook API

First Step Building a user endpoint

For this project we need users. Users will be helpfull for our adress book and for our authentication process.

Users will be define with at least a first name and a last name. We also need an unique identifier to retreive the user.

Note: For this tutorial the file yyou create will be named app.py To launch your application then just type in a terminal:

```
python app.py
```

Define a model

```
from rest_api_framework import models

class UserModel(models.Model):

    fields = [models.StringField(name="first_name", required=True),
              models.StringField(name="last_name", required=True),
              models.PkField(name="id", required=True)
              ]
```

The use of required_true will ensure that a user without this field cannot be created

Chose a DataStore

We also need a datastore to get a place where we can save our users. For instance we will use a sqlite3 database. The SQLiteDataStore is what we need

```
from rest_api_framework.datastore import SQLiteDataStore
```

Chose a view

We want results to be rendered as Json. We use the JsonResponse view for that:

```
from rest_api_framework.views import JsonResponse
```

Create The user endpoint

To create an endpoint, we need a controller. This will manage our endpoint in a RESTFUL fashion.

```
from rest_api_framework.controllers import Controller

class UserEndPoint(Controller):
    ressource = {
        "ressource_name": "users",
        "ressource": {"name": "adress_book.db", "table": "users"},
        "model": UserModel,
        "datastore": SQLiteDataStore
        }

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"]
        }

    view = {"response_class": JsonResponse}
```

then we must run our application:

```
if __name__ == '__main__':
    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([UserEndPoint])
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

Summary

So far, all of the code should look like this:

```
from rest_api_framework import models
from rest_api_framework.datastore import SQLiteDataStore
from rest_api_framework.views import JsonResponse
from rest_api_framework.controllers import Controller


class UserModel(models.Model):

    fields = [models.StringField(name="first_name", required=True),
              models.StringField(name="last_name", required=True),
              models.PkField(name="id", required=True)
              ]
```

```
class UserEndPoint(Controller):
    ressource = {
        "ressource_name": "users",
        "ressource": {"name": "adress_book.db", "table": "users"},
        "model": UserModel,
        "datastore": SQLiteDataStore
        }

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"]
        }

    view = {"response_class": JsonResponse}

if __name__ == '__main__':
    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([UserEndPoint])
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

Note: to launch your application, just type in a terminal:

```
python app.py
```

Next: *Playing with the newly created endpoint*

Playing with the newly created endpoint

First you can check that your endpoint is up

```
curl -i "http://localhost:5000/users/"

HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 44
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 11:13:44 GMT

{
"meta": {
    "filters": {}
},
"object_list": []
}
```

Your endpoint is responding but does not have any data. Let's add some:

Create a user

```
curl -i -H "Content-type: application/json" -X POST -d '{"first_name":"John", "last_
↪name": "Doe"}'  http://localhost:5000/users/

HTTP/1.0 201 CREATED
Location: http://localhost:5000/users/1/
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 13:00:13 GMT
```

If you look carfully at the response, you can see the header “Location” giving you the ressource uri of the ressource you just created. This is usefull if you want to retreive your object. Let’s get a try:

List and Get

```
curl -i "http://localhost:5000/users/1/"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 51
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 16:53:19 GMT

{
"first_name": "John",
"id": 1,
"last_name": "Doe",
"ressource_uri": "/users/1/"
}
```

You can see that ressource_uri was not part of the ressource. It have been added by the View itself. View can add multiple metadata, remove or change some fields and so on. More on that in *Show data to users*

The list of users is also updated:

```
curl -i "http://localhost:5000/users/"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 83
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 17:03:00 GMT

{
"meta": {
    "filters": {}
},
"object_list": [
    {
        "first_name": "John",
        "id": 1,
        "last_name": "Doe",
        "ressource_uri": "/users/1/"
    }
]
}
```

Delete a user

Let’s add a new user:

```
curl -i -H "Content-type: application/json" -X POST -d '{"first_name":"Peter", "last_
↪name": "Something"}'  http://localhost:5000/users/

HTTP/1.0 201 CREATED
Location: http://localhost:5000/users/2/
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 13:00:13 GMT
```

and now delete it:

```
curl -i -X DELETE "http://localhost:5000/users/2/"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 20:41:46 GMT
```

You can check that the user no longer exists:

```
curl -i "http://localhost:5000/users/2/"
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Connection: close
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 11:16:33 GMT

{ "error": "<p>The requested URL was not found on the
server.</p><p>If you entered the URL manually please check your
spelling and try again.</p>" }
```

And the list is also updated:

```
curl -i "http://localhost:5000/users/"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 125
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 11:17:46 GMT

{
"meta": {
    "filters": {}
},
"object_list": [
    {
        "first_name": "John",
        "id": 1,
        "last_name": "Doe",
        "ressource_uri": "/users/1/"
    }
]
}
```

Update a User

Let’s go another time to the creation process:

```
curl -i -H "Content-type: application/json" -X POST -d '{"first_name":"Steve", "last_
↪name": "Roger"}'  http://localhost:5000/users/
HTTP/1.0 201 CREATED
Location: http://localhost:5000/users/3/
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 20:45:38 GMT
```

But well everybody now that Steve Roger real name is Captain America. Let’s update this user:

```
curl -i -H "Content-type: application/json" -X PUT -d '{"first_name":"Capitain",
↪"last_name": "America"}'  http://localhost:5000/users/3/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 58
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 20:57:47 GMT

{"first_name": "Capitain", "last_name": "America", "id": 3, "ressource_uri": "/users/
↪3/"}
```

Argh! Thats a typo. the fist name is “Captain”, not “Capitain”. Let’s correct this:

```
curl -i -H "Content-type: application/json" -X PUT -d '{"first_name":"Captain"}' ␣
↪http://localhost:5000/users/3/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 59
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 21:08:04 GMT

{"first_name": "Captain", "last_name": "America", "id": 3, "ressource_uri": "/users/3/
↪"}
```

Filtering

Ressources can be filtered easily using parameters:

```
curl -i "http://localhost:5000/users/?last_name=America"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 236
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 12:07:21 GMT

{"meta": {"filters": {"last_name": "America"}}, "object_list":
[{"first_name": "Joe", "last_name": "America", "id": 1,
"ressource_uri": "/users/1/"}, {"first_name": "Bob", "last_name":
"America", "id": 3, "ressource_uri": "/users/3/"}]
```

Multiple filters are allowed:

```
curl -i "http://localhost:5000/users/?last_name=America&first_name=Joe"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 171
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 12:09:32 GMT

{"meta": {"filters": {"first_name": "Joe", "last_name": "America"}},
"object_list": [{"first_name": "Joe", "last_name": "America", "id": 1,
"ressource_uri": "/users/1/"}]}
```

Error handling

Of course, If data is not formated as expected by the API, the base error handling take place.

Missing data

If you don't provide a last_name, the API will raise a BAD REQUEST explaining your error:

```
curl -i -H "Content-type: application/json" -X POST -d '{"first_name":"John"}'  http:/
↪/localhost:5000/users/

HTTP/1.0 400 BAD REQUEST
Content-Type: application/json
Content-Length: 62
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 13:21:10 GMT

{"error": "last_name is missing. Cannot create the ressource"}
```

Invalid Data

The same apply if you dont give coherent data:

```
curl -i -H "Content-type: application/json" -X POST -d '{"first_name":45, "last_name
↪": "Doe"}'  http://localhost:5000/users/

HTTP/1.0 400 BAD REQUEST
Content-Type: application/json
Content-Length: 41
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 13:24:53 GMT
{"error": "first_name does not validate"}
```

however, there is no duplicate check. So you can create as many "John Doe" you want. This could be a huge problem if your not able to validate uniqueness of a user. For the API, this is not a problem because each user is uniquely identified by his id.

If you need to ensure it can be only one John Doe, you must add a validator on your datastore.

Autodocumentation

Your API is autodocumented by Python REST API Framework.

```
curl -i -X GET http://localhost:5000/schema/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 268
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 08:24:13 GMT


{
    "users": {
        "allowed list_verbs": [
            "GET",
            "POST"
        ],
        "allowed unique ressource": [
            "GET",
            "PUT",
            "DELETE"
        ],
        "list_endpoint": "/users/",
        "schema_endpoint": "/schema/users/"
    }
}
```

```
url -i -X GET http://localhost:5000/schema/users/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 206
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 09:04:16 GMT

{
    "first_name": {
        "example": "Hello World",
        "required": "true",
        "type": "string"
    },
    "last_name": {
        "example": "Hello World",
        "required": "true",
        "type": "string"
    }
}
```

Next: *Adding validators to your DataStore*

Adding validators to your DataStore

In this exemple, you want to check that a user with the same last_name and same first_name does not exist in your datastore before creating a new user.

For this you can use UniqueTogether:

UniqueTogether

Change your UserEndPoint to get:

```
from rest_api_framework.datastore.validators import UniqueTogether

class UserEndPoint(Controller):
    ressource = {
        "ressource_name": "users",
        "ressource": {"name": "adress_book.db", "table": "users"},
        "model": UserModel,
        "datastore": SQLiteDataStore,
        "options":{"validators": [UniqueTogether("first_name", "last_name")]}
        }

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"]
        }

    view = {"response_class": JsonResponse}
```

each of ressource, controller and views can have various options to add new functionality to them. The “validators” option of ressource enable some datastore based validators. As you can see, validators are a list. This meen that you can add many validators for a single datastore.

UniqueTogether will ensure that a user with first_name: John and last_name: Doe cannot be created.

Let’s try:

```
curl -i -H "Content-type: application/json" -X POST -d '{"first_name": "John", "last_
↪name": "Doe"}'  http://localhost:5000/users/
HTTP/1.0 400 BAD REQUEST
Content-Type: application/json
Content-Length: 57
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 17:13:41 GMT

{"error": "first_name,last_name must be unique together"}
```

Next: *Show data to users*

Show data to users

The view you have used so far just added a ressource_uri. But preserve the id attribut. As id is an internal representation of the data you may wich to remove it.

Define a formater function

To do so you’ll have to write a simple function to plug on the view. This function is a formater. When the View instanciate the formater, it give you access to the response object and the object to be rendered.

Because you want to remove the id of the reprensentaion of your ressource, you can write:

```
def remove_id(response, obj):
    obj.pop("id")
    return obj
```

and change the view part of your UserEndPoint as follow:

```
view = {"response_class": JsonResponse,
        "options": {"formaters": ["add_ressource_uri",
        remove_id]}}
```

add_ressource_uri is the default formatter for this View. You dont need to remove it for now. But if you try, then it will work as expected. The ressource_uri field will be removed.

The idea behind Python REST API Framework is to always get out of your way.

You can check that it work as expected:

```
curl -i "http://localhost:5000/users/1/"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 80
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Mon, 14 Oct 2013 23:41:55 GMT

{"first_name": "Captain", "last_name": "America",
"ressource_uri": "/users/1/"}
```

Make things generics

This implementation work on your endpoint because you each object has an id. But, if later you create another endpoint with ressources lacking the "id" key, you'll have to re-write your function.

Instead, you can take advantage of the response wich is part of the parameters of your function.

response object carry the attribut model who define your ressources fields. You can then get the name of the Pk field used with this ressource with:

```
response.model.pk_field.name
```

Your code then become:

```
def remove_id(response, obj):
    obj.pop(response.model.pk_field.name)
    return obj
```

And reuse this formatter as long as you need.

Formaters are here to help you build clean and meaningful ressources representations. It should hide internal representation of your ressources and return all of the fields needed to manipulate and represent your data.

Next *Working with Pagination*

Working with Pagination

Creating fixtures

When your address book will be full of entry, you will need to add a pagination on your API. As it is a common need, REST API Framework implement a very easy way of doing so.

Before you can play with the pagination process, you will need to create more data. You can create those records the way you want:

- direct insert into the database

```
sqlite3 adress_book.db
INSERT INTO users VALUES ("Nick", "Furry", 6);
```

- using the datastore directly

```
store = SQLiteDataStore({"name": "adress_book.db", "table": "users"}, UserModel)
store.create({"first_name": "Nick", "last_name": "Furry"})
```

- using your API

```
curl -i -H "Content-type: application/json" -X POST -d '{"first_name": "Nick", "last_
↪name": "Furry"}'  http://localhost:5000/users/
```

each on of those methods have advantages and disavantages but they all make the work done. For this example, I propose to use the well know requests package with a script to create a bunch of random records:

For this to work you need to install resquests : http://docs.python-requests.org/en/latest/user/install/#install

```
import json
import requests
import random
import string

def get_random():
    return ''.join(
                   random.choice(
                     string.ascii_letters) for x in range(
                     int(random.random() * 20)
                     )
                   )

for i in range(200):
    requests.post("http://localhost:5000/users/", data=json.dumps({"first_name": get_
↪random(), "last_name": get_random()}))
```

Pagination

Now your datastore is filled with more than 200 records, it's time to paginate. To do so import Pagination and change the controller part of your app.

```
from rest_api_framework.pagination import Pagination

class UserEndPoint(Controller):
    ressource = {
        "ressource_name": "users",
        "ressource": {"name": "adress_book.db", "table": "users"},
        "model": UserModel,
        "datastore": SQLiteDataStore,
        "options": {"validators": [UniqueTogether("first_name", "last_name")]}
        }

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"],
        "options": {"pagination": Pagination(20)}
        }

    view = {"response_class": JsonResponse,
            "options": {"formaters": ["add_ressource_uri", remove_id]}}
```

and try your new pagination:

```
curl -i "http://localhost:5000/users/"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 1811
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 11:32:55 GMT

{
"meta": {
    "count": 20,
    "filters": {},
    "next": "?offset=20",
    "offset": 0,
    "previous": "null",
    "total_count": 802
},
"object_list": [
    {
        "first_name": "Captain",
        "last_name": "America",
        "ressource_uri": "/users/1/"
    },
    {
        "first_name": "Captain",
        "last_name": "America",
        "ressource_uri": "/users/3/"
    },
    {
        "first_name": "John",
        "last_name": "Doe",
        "ressource_uri": "/users/4/"
    },
    {
        "first_name": "arRFOSYZT",
        "last_name": "",
        "ressource_uri": "/users/5/"
    },
    {
        "first_name": "iUJsYORMuYeMUDy",
        "last_name": "TqFpmcBQD",
        "ressource_uri": "/users/6/"
    },
    {
        "first_name": "EU",
        "last_name": "FMSAbcUJBSBDPaF",
        "ressource_uri": "/users/7/"
    },
    {
        "first_name": "mWAwamrMQARXW",
        "last_name": "yMNpEnYOPzY",
        "ressource_uri": "/users/8/"
    },
    {
        "first_name": "y",
        "last_name": "yNiKP",
        "ressource_uri": "/users/9/"
    },
    {
        "first_name": "s",
        "last_name": "TRT",
        "ressource_uri": "/users/10/"
    },
    {
        "first_name": "",
        "last_name": "zFUaBd",
        "ressource_uri": "/users/11/"
    },
    {
        "first_name": "WA",
        "last_name": "priJ",
        "ressource_uri": "/users/12/"
    },
    {
        "first_name": "XvpLttDqFmR",
        "last_name": "liU",
        "ressource_uri": "/users/13/"
    },
    {
        "first_name": "ZhJqTgYoEUzmcN",
        "last_name": "KKDqHJwJMxPSaTX",
        "ressource_uri": "/users/14/"
    },
    {
        "first_name": "qvUxiKIATdKdkC",
        "last_name": "wIVzfDlKCkjkHIaC",
        "ressource_uri": "/users/15/"
    },
    {
        "first_name": "YSSMHxdDQQsW",
        "last_name": "UaKCKgKsgEe",
        "ressource_uri": "/users/16/"
    },
    {
        "first_name": "EKLFTPJLKDINZio",
        "last_name": "nuilPTzHqattX",
        "ressource_uri": "/users/17/"
    },
    {
        "first_name": "SPcDBtmDIi",
        "last_name": "MrytYqElXiIxA",
        "ressource_uri": "/users/18/"
    },
    {
        "first_name": "OHxNppXiYp",
        "last_name": "AUvUXFRPICsJIB",
        "ressource_uri": "/users/19/"
    },
    {
        "first_name": "WBFGxnoe",
        "last_name": "KG",
        "ressource_uri": "/users/20/"
    },
    {
        "first_name": "i",
        "last_name": "ggLOcKPpMfgvVGtv",
        "ressource_uri": "/users/21/"
    }
]
}
```

Browsering Through Paginated objects

Of course you get 20 records but the most usefull part is the meta key:

```
{"meta":
    {"count": 20,
    "total_count": 802,
    "next": "?offset=20",
    "filters": {},
    "offset": 0,
    "previous": "null"}
}
```

You can use the "next" key to retreive the 20 next rows:

```
curl -i "http://localhost:5000/users/?offset=20"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 1849
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 11:38:59 GMT
```

```
{"meta": {"count": 20, "total_count": 802, "next": "?offset=40",
"filters": {}, "offset": 20, "previous": "?offset=0"}, "object_list":
[<snip for readability>]}
```

Note: The count and offset keywords can be easily changed to match your needs. pagination class may take an offset_key and count_key parameters. So if you prefer to use first_id and limit, you can change your Paginator class to do so:

```
"options": {"pagination": Pagination(20,
                                     offset_key="first_id",
                                     count_key="limit")
```

Wich will results in the following:

```
curl -i "http://localhost:5000/users/"
{"meta": {"first_id": 0, "total_count": 802, "next": "?first_id=20",
"limit": 20, "filters": {}, "previous": "null"}, "object_list": [<snip
for readability>]
```

Pagination and Filters

Pagination and filtering play nice together

```
curl -i "http://localhost:5000/users/?last_name=America"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 298
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 12:14:59 GMT

{"meta": {"count": 20,
          "total_count": 2,
          "next": "null",
          "filters": {"last_name": "America"},
          "offset": 0,
          "previous": "null"},
          "object_list": [
              {"first_name": "Joe",
               "last_name": "America",
               "ressource_uri": "/users/1/"},
              {"first_name": "Bob",
               "last_name": "America",
               "ressource_uri": "/users/3/"}
          ]
 }
```

Next: *Loading multiple endpoint*

Loading multiple endpoint

Now that your fist endpoint work as expected, you will need to add an address field on the user model. But as some users can have the same address, and because you want to retreive some user using an address, you will need to create an new endpoint:

Define a new model

```
class AddressModel(models.Model):

    fields = [models.StringField(name="country", required=True),
              models.StringField(name="city", required=True),
              models.StringField(name="street", required=True),
              models.IntegerField(name="number", required=True),
              models.PkField(name="id", required=True)
              ]
```

Inherite from previous apps

The only thing that change here in comparisson to the UserEndPoint you created earlier is the ressource dict. So instead of copy pasting a lot of lines, let's heritate from your first app:

```
class AddressEndPoint(UserEndPoint):
    ressource = {
        "ressource_name": "address",
        "ressource": {"name": "adress_book.db", "table": "address"},
        "model": AddressModel,
        "datastore": SQLiteDataStore
        }
```

All the options already defined in the UserEndPoint will be available with this new one. Pagination, formater and so on.

Of course, if you change the controller or the view of UserEndPoint, AddressEndPoint will change too. If it become a problem, you'll have to create a base class with common options and configurations and each of your endpoints will inherit from this base class. Each endpoint will be able to change some specifics settings.

The last thing to do to enable your new endpoint is to add it to the WSGIDispatcher

Add the app to the dispatcher

```
if __name__ == '__main__':
    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([AddressEndPoint, UserEndPoint])
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

Note: For now the order you register AddressEndPoint and UserEndPoint doesn't make a difference. But we will add a reference from the user table to the address table. At this point, you will need to reference AddressEndPoint before UserEndPoint.

Check that everything work

```
curl -i "http://localhost:5000/address/"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 124
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 15:45:34 GMT

{
"meta": {
    "count": 20,
    "filters": {},
    "next": "null",
    "offset": 0,
    "previous": "null",
    "total_count": 0
},
"object_list": []
}
```

next: *Linking ressource together*

Linking ressource together

Now that you have users and address, you want to link them together. Adding a reference from a user to his user.

Not all the datastore can handle this type of relation but hopefully, the SQLiteDataStore does.

First you will need to change your UserModel definition:

```
fields = [models.StringField(name="country", required=True),
          models.StringField(name="city", required=True),
          models.StringField(name="street", required=True),
          models.IntegerField(name="number", required=True),
          models.IntForeign(name="user",
                            foreign={"table": "users",
                                     "column": "id",
                                     }
                            ),
          models.PkField(name="id", required=True)
          ]
```

The part we added is:

```
models.IntForeign(name="address",
                  foreign={"table": "address",
                           "column": "id",
                           }
                  ),
```

This will add a foreign key constrain on the user ensuring the address id you give corresspond to an existing address.

- table : is the table of the ressource your are linking
- column: is the column you will check for the constrain

Note: unfortunately, at the time of writing, there is no way to update the schema automaticaly. You will need either to destroy your database (Python Rest Framework will create a fresh one) or do an alter table by hands. As this is just a tutorial, we will choose the second option and delete the file “adress.db”

It’s also important to note the your endpoints must be listed in the Wrapper in the order of foreing keys. First the model to link to, then the model that will be linked

Adding an adress

```
curl -i -H "Content-type: application/json" -X POST -d
'{"country":"France", "city": "Paris", "street": "quais de Valmy",
"number": 45}' http://localhost:5000/address/

HTTP/1.0 201 CREATED
Location: http://localhost:5000/address/1/
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 17:23:49 GMT
```

Create a user linked to an address

Because, as the API developper you know that http://localhost:5000/address/1/ corresond to the address with the "id" 1 you can create a user:

```
curl -i -H "Content-type: application/json" -X POST -d
'{"first_name":"Super", "last_name": "Dupont", "address": 1}'
http://localhost:5000/users/

HTTP/1.0 201 CREATED
Location: http://localhost:5000/users/1/
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 17:27:34 GMT
```

You can check that your Foreign constrain is working with:

```
curl -i -H "Content-type: application/json" -X POST -d
'{"first_name":"Super", "last_name": "Man", "address": 2}'
http://localhost:5000/users/

HTTP/1.0 400 BAD REQUEST
Content-Type: application/json
Content-Length: 38
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 17:30:03 GMT

{"error": "address does not validate"}
```

This fail because address 2 does not exists.

Retreive the adress of a user

If you now the user, it's easy to get the adress.

First get the user:

```
curl -i http://localhost:5000/users/1/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 90
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 17:42:18 GMT
```

```
{
"address": 1,
"first_name": "Super",
"last_name": "Dupont",
"ressource_uri": "/users/1/"
}
```

His adress has the id “1”. We can issue a request:

```
curl -i http://localhost:5000/address/1/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 112
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 17:44:07 GMT

{
    "city": "Paris",
    "country": "France",
    "number": 45,
    "ressource_uri": "/address/1/",
    "street": "quais de Valmy"
}
```

Retreive users from an adress

The same apply in the other side. As we know the adress id:

```
curl -i http://localhost:5000/users/?address=1
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 228
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 17:46:13 GMT

{
    "meta": {
        "count": 20,
        "filters": {
            "address": "1"
        },
        "next": "null",
        "offset": 0,
        "previous": "null",
        "total_count": 1
    },
    "object_list": [
        {
            "address": 1,
            "first_name": "Super",
            "last_name": "Dupont",
            "ressource_uri": "/users/1/"
        }
    ]
}
```

next: *Representing relations*

Representing relations

Even if now can query adress from a user and users from an adress, your users cannot know that the field "address": 1 correspond to /address/1/ plus it break a common rule. The id of the relation correspond to yor internal logic. Users doesn't have to know how it work, they just have to use it.

What we will try to do in this part of the tutorial is the following:

- http://localhost:5000/users/1/ should return:

```
{
    "address": /address/1/,
    "first_name": "Super",
    "last_name": "Dupont",
    "ressource_uri": "/users/1/"
}
```

- this request should work

```
curl -i -H "Content-type: application/json" -X POST -d
'{"first_name":"Super", "last_name": "Dupont", "address":
"/adress/1/"}'  http://localhost:5000/users/
```

- Of course, http://localhost:5000/users/?address=/adress/1/" should return the users with this address.

Representiing the relation on the user side

This is the simplest task because you already changed the response result by adding remove_id function to the list of View formater in *Show data to users*

```
def format_address(response, obj):
    obj['address'] = "/address/{0}".format(obj['address'])
    return obj
```

Sure this method will work but if you get a close look on how ForeignKeyField (IntForeign inherit from this class) You will see that the ForeignKeyField is filled with th options parameter you gave at the foreign key creation. You can so write:

```
def format_foreign_key(response, obj):
    from rest_api_framework.models.fields import ForeignKeyField
    for f in response.model.get_fields():
        if isinstance(f, ForeignKeyField):
            obj[f.name] = "/{0}/{1}/".format(f.options["foreign"]["table"],
                                              obj[f.name])
    return obj
```

This function can then be used in all your project when you need to translate a foreignkey into a meaning full ressource uri

For now, you can add this function to the list of formaters in your UserEndPoint views:

```
view = {"response_class": JsonResponse,
        "options": {"formaters": ["add_ressource_uri",
                                  remove_id,
                                  format_foreign_key

                                  ]}}
```

Check the formater

```
curl -i http://localhost:5000/users/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 226
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 21:21:44 GMT

{
    "meta": {
        "count": 20,
        "filters": {},
        "next": "null",
        "offset": 0,
        "previous": "null",
        "total_count": 1
    },
    "object_list": [
        {
            "address": "/address/1/",
            "first_name": "Super",
            "last_name": "Dupont",
            "ressource_uri": "/users/1/"
        }
    ]
}
```

Formating data for the system

Because you hide the internal implementation of your API to your user, you have to give him a way to interact with your API.

To do so, you need to create a formater, exactly like you have done for the View. But this time you must do it for the Controller.

```python
def foreign_keys_format(view, obj):
    from rest_api_framework.models.fields import ForeignKeyField
    for f in view.datastore.model.get_fields():
        if isinstance(f, ForeignKeyField):
            if obj.get(f.name):
                obj[f.name] = int(obj[f.name].split("/")[-2])
    return obj
```

and add it to the controller formater. Change the UserEndPoint controller:

```
controller = {
    "list_verbs": ["GET", "POST"],
    "unique_verbs": ["GET", "PUT", "DELETE"],
    "options": {"pagination": Pagination(20),
                "formaters": [foreign_keys_format]}
    }
```

Now, each time the endpoint will deal with a data fields corresponding to a ForeignKeyField it will retreive the id from the url supplied

"/address/1/" will be translated in 1

Check the Controller translation

```
curl -i -H "Content-type: application/json" -X POST -d
'{"first_name":"Captain", "last_name": "America", "address":
"/adress/1/"}'  http://localhost:5000/users/


HTTP/1.0 201 CREATED
Location: http://localhost:5000/users/2/
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 22:23:43 GMT
```

```
curl -i http://localhost:5000/users/?address=/adress/1/
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 341
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Tue, 15 Oct 2013 22:33:47 GMT

{
    "meta": {
        "count": 20,
        "filters": {
            "address": 1
        },
        "next": "null",
        "offset": 0,
        "previous": "null",
        "total_count": 2
    },
    "object_list": [
        {
            "address": "/address/1/",
            "first_name": "Super",
            "last_name": "Dupont",
            "ressource_uri": "/users/1/"
        },
        {
            "address": "/address/1/",
            "first_name": "Supe",
            "last_name": "Dupont",
            "ressource_uri": "/users/2/"
        }
    ]
}
```

next: *Authentication and Authorization: Protecting your API*

Authentication and Authorization: Protecting your API

Authentication and Authorization are different topics as you can implement Authentication without Authorization (For rate-limiting or loggin for example).

Authentication

The fist thing you can do is to add an Authentication backend. Authentication backend needs a datastore to retreive the user accessing the API. This datastore can be used by another endpoint of your API or a datastore aimed for this purpose only.

In this example, we will use a very simple datastore, meant for testing purpose: the PythonListDataStore.

Define a backend

The PythonListDataStore is just a list of python dictionnary. So let's first create this list:

```
ressources = [{"accesskey": "hackme"}, {"accesskey": "nopassword"}]
```

Like any other datastore, you need a Model to describe your datastore:

```
class KeyModel(models.Model):
    fields = [
        models.StringPkField(name="accesskey", required=True)
        ]
```

Then you can instanciate your datastore:

```
from rest_api_framework.datastore import PythonListDataStore

datastore = PythonListDataStore(ressources, KeyModel)
```

Instanciate the Authentication backend

To keep this example simple we will use another testing tool, the ApiKeyAuthentication

ApiKeyAuthentication will inspect the query for an "apikey" parameter. If the "apikey" correspond to an existing object in the datastore, it will return this object. Otherwise, the user is anonymous.

```
from rest_api_framework.authentication import ApiKeyAuthentication
authentication = ApiKeyAuthentication(datastore, identifier="accesskey")
```

Then you can plug this authentication backend to your endpoint:

```
controller = {
    "list_verbs": ["GET", "POST"],
    "unique_verbs": ["GET", "PUT", "DELETE"],
    "options": {"pagination": Pagination(20),
                "formaters": [foreign_keys_format],
                "authentication": authentication}
    }
```

Instanciate the Authorization backend

The Authorization backend relies on the Authentication backend to retreive a user. With this user and the request, it will grant access or raise an Unauthorized error.

For this example we will use the base Authentication class. This class tries to authenticate the user. If the user is authenticated, then access is granted. Otherwise, it is not.

from rest_api_framework.authentication import Authorization then add it to the controller options:

```
controller = {
    "list_verbs": ["GET", "POST"],
    "unique_verbs": ["GET", "PUT", "DELETE"],
    "options": {"pagination": Pagination(20),
                "formaters": [foreign_keys_format],
                "authentication": authentication,
                "authorization": Authorization,
                }
    }
```

Testing Authentication and Authorization Backend

Let's give a try:

```
curl -i -X GET http://localhost:5000/users/?accesskey=hackme
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 350
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 12:18:52 GMT


curl -i -X GET http://localhost:5000/users/?accesskey=helloworld
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 350
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 12:19:26 GMT

curl -i -X GET http://localhost:5000/users/
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 350
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 12:19:45 GMT
```

next: *Rate Limiting your endpoints*

Rate Limiting your endpoints

Now that your users are authenticated and that you put an authorization backend, you can add a rate limit on your api. Rate limit will prevent your users to over use your endpoints.

With rate limit, a user can call your API at a certain rate. A number of calls per an interval. You have to decide how many call and wich interval.

For this example, let say something like 100 call per 10 minutes. For Python REST Framework, interval are counted in seconds so 10 minutes equals 10*60 = 600

Create a datastore for rate-limit:

The rate-limit implementation need a datastore to store rate-limit. Let's create one:

```
class RateLimitModel(models.Model):
    fields = [models.StringPkField(name="access_key"),
              models.IntegerField(name="quota"),
              models.TimestampField(name="last_request")]
```

You can then add your new datastore to the list of options of you controller:

Add Rate-limit to your API

```
from rest_api_framework.ratelimit import RateLimit

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"],
        "options": {"pagination": Pagination(20),
                    "formaters": [foreign_keys_format],
                    "authentication": authentication,
                    "authorization": Authorization,
                    "ratelimit": RateLimit(
                PythonListDataStore([],RateLimitModel),
                interval=10*60,
                quota=100)
                    }
        }
```

Test!

```
curl -i -X GET http://localhost:5000/users/?accesskey=hackme
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 350
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 15:22:12 GMT

{"meta": {"count": 20, "total_count": 2, "next": "null", "filters":
{"accesskey": "hackme"}, "offset": 0, "previous": "null"},
"object_list": [{"ressource_uri": "/users/1/", "first_name": "Super",
"last_name": "Dupont", "address": "/address/1/"}, {"ressource_uri":
"/users/2/", "first_name": "Supe", "last_name": "Dupont", "address":
"/address/1/"}]}
```

```
curl -i -X GET http://localhost:5000/users/?accesskey=hackme
HTTP/1.0 429 TOO MANY REQUESTS
```

```
Content-Type: application/json
Content-Length: 23
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 15:22:14 GMT
```

next: *Implementing Partial Response*

Implementing Partial Response

You can give your user the ability to retreive only the data they need instead of all of an object representation. For the adress field, some can want to retreive only the country and the city field but do not care about the others.

with Python REST API Framework, it's easy to make this happend.

First import the Partial base class:

```
from rest_api_framework.partials import Partial
```

Then add the partial option to the AddressEndPoint:

```
class AddressEndPoint(UserEndPoint):
    ressource = {
        "ressource_name": "address",
        "ressource": {"name": "adress_book.db", "table": "address"},
        "model": AddressModel,
        "datastore": SQLiteDataStore,
        "options": {"partial": Partial()}
        }
```

Test the Partial

```
curl -i -X GET "http://localhost:5000/address/?accesskey=hackme&fields=city,country"
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 241
Server: Werkzeug/0.8.3 Python/2.7.2
Date: Wed, 16 Oct 2013 15:50:27 GMT

{
    "meta": {
        "count": 20,
        "filters": {
            "accesskey": "hackme",
            "fields": "city,country"
        },
        "next": "null",
        "offset": 0,
        "previous": "null",
        "total_count": 1
    },
    "object_list": [
        {
            "city": "Paris",
            "country": "France",
            "ressource_uri": "/address/1/"
        }
    ]
}
```

next: *The Whole Application*

The Whole Application

To let you have a look on the application you have build so far, here is the whole application you have build:

```
from rest_api_framework import models
from rest_api_framework.models.fields import ForeignKeyField
from rest_api_framework.datastore import SQLiteDataStore, PythonListDataStore
from rest_api_framework.datastore.validators import UniqueTogether
from rest_api_framework.controllers import Controller
from rest_api_framework.pagination import Pagination
from rest_api_framework.authentication import (ApiKeyAuthentication,
                                               Authorization)
from rest_api_framework.ratelimit import RateLimit
from rest_api_framework.partials import Partial
from rest_api_framework.views import JsonResponse


ressources = [{"accesskey": "hackme"}, {"accesskey": "nopassword"}]


class KeyModel(models.Model):
    fields = [
        models.StringPkField(name="accesskey", required=True)
        ]


class RateLimitModel(models.Model):
    fields = [models.StringPkField(name="access_key"),
              models.IntegerField(name="quota"),
              models.TimestampField(name="last_request")]


datastore = PythonListDataStore(ressources, KeyModel)
authentication = ApiKeyAuthentication(datastore, identifier="accesskey")


class UserModel(models.Model):

    fields = [models.StringField(name="first_name", required=True),
              models.StringField(name="last_name", required=True),
              models.PkField(name="id", required=True),
              models.IntForeign(name="address",
                                foreign={"table": "address",
                                         "column": "id",
                                         }
                                ),

              ]
```

```
class AddressModel(models.Model):

    fields = [models.StringField(name="country", required=True),
              models.StringField(name="city", required=True),
              models.StringField(name="street", required=True),
              models.IntegerField(name="number", required=True),
              models.PkField(name="id", required=True)
              ]


def remove_id(response, obj):
    obj.pop(response.model.pk_field.name)
    return obj


def format_foreign_key(response, obj):

    for f in response.model.get_fields():
        if isinstance(f, ForeignKeyField):
            obj[f.name] = "/{0}/{1}/".format(f.options["foreign"]["table"],
                                             obj[f.name])
    return obj


def foreign_keys_format(view, obj):
    for f in view.datastore.model.get_fields():
        if isinstance(f, ForeignKeyField):
            if obj.get(f.name):
                obj[f.name] = int(obj[f.name].split("/")[-2])
    return obj


class UserEndPoint(Controller):
    ressource = {
        "ressource_name": "users",
        "ressource": {"name": "adress_book.db", "table": "users"},
        "model": UserModel,
        "datastore": SQLiteDataStore,
        "options": {"validators": [UniqueTogether("first_name", "last_name")],
                    }
        }

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"],
        "options": {"pagination": Pagination(20),
                    "formaters": [foreign_keys_format],
                    "authentication": authentication,
                    "authorization": Authorization,
                    "ratelimit": RateLimit(
                PythonListDataStore([],RateLimitModel),
                interval=100,
                quota=200),
                    }
        }

    view = {"response_class": JsonResponse,
            "options": {"formaters": ["add_ressource_uri",
                                      remove_id,
                                      format_foreign_key
                                      ]}}


class AddressEndPoint(UserEndPoint):
    ressource = {
        "ressource_name": "address",
        "ressource": {"name": "adress_book.db", "table": "address"},
        "model": AddressModel,
        "datastore": SQLiteDataStore,
        "options": {"partial": Partial()}
        }
if __name__ == '__main__':

    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([AddressEndPoint, UserEndPoint])
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

CHAPTER 3

REST API Framework API references

Main modules

Controllers

class `rest_api_framework.controllers.`**`ApiController`**`(`**args*, ***kwargs*`)`

Inherit from *`WSGIWrapper`* implement the base API method. Should be inherited to create your own API

`index``(`*request*`)`

The root url of your ressources. Should present a list of ressources if method is GET. Should create a ressource if method is POST :param request: :type request: `werkzeug.wrappers.Request` if self.auth is set call `authentication.Authentication.check_auth()`

Returns *`ApiController.get_list()`* if request.method is GET, *`ApiController.create()`* if request.method is POST

`paginate``(`*request*`)`

invoke the Pagination class if the optional pagination has been set. return the objects from the datastore using datastore.get_list :param request: :type request: `werkzeug.wrappers.Request`

`get_list``(`*request*`)`

On the base implemetation only return self.paginate(request). Placeholder for pre pagination stuff.

Parameters **`request`** (`werkzeug.wrappers.Request`) –

`unique_uri``(`*request*, *identifier*`)`

Retreive a unique object with his URI. Act on it accordingly to the Http verb used.

Parameters **`request`** (`werkzeug.wrappers.Request`) –

`get``(`*request*, *identifier*`)`

Return an object or 404

Parameters **`request`** (`werkzeug.wrappers.Request`) –

create (*request*)
Try to load the data received from json to python, format each field if a formater has been set and call the datastore for saving operation. Validation will be done on the datastore side

If creation is successfull, add the location the the headers of the response and render a 201 response with an empty body

Parameters **request** (`werkzeug.wrappers.Request`) –

update_list (*request*)
Try to mass update the data.

update (*request*, *identifier*)
Try to retreive the object identified by the identifier. Try to load the incomming data from json to python.

Call the datastore for update.

If update is successfull, return the object updated with a status of 200

Parameters **request** (`werkzeug.wrappers.Request`) –

delete (*request*, *identifier*)
try to retreive the object from the datastore (will raise a NotFound Error if object does not exist) call the delete

method on the datastore.

return a response with a status code of 204 (NO CONTENT)

Parameters **request** (`werkzeug.wrappers.Request`) –

class `rest_api_framework.controllers.`**Controller** (**args*, ***kwargs*)
Controller configure the application. Set all configuration options and parameters on the Controller, the View and the Ressource

load_urls ()

Parameters **urls** (*`list`*) – A list of tuple in the form (url(string), view(string), permitted Http verbs(list))

return a `werkzeug.routing.Map`

this method is automaticaly called by __init__ to build the `Controller` urls mapping

make_options (*options*)
Make options enable Pagination, Authentication, Authorization, RateLimit and all other options an application need.

class `rest_api_framework.controllers.`**WSGIWrapper**
Base Wsgi application loader. WSGIWrapper is an abstract class. Herited by `ApiController` it make the class callable and implement the request/response process

wsgi_app (*environ*, *start_response*)
instanciate a Request object, dispatch to the needed method, return a response

dispatch_request (*request*)
Using the `werkzeug.routing.Map` constructed by `load_urls()` call the view method with the request object and return the response object.

class `rest_api_framework.controllers.`**AutoDocGenerator** (*apps*)
Auto generate a documentation endpoint for each endpoints registered.

schema (*request*)
Generate the schema url of each endpoints

`ressource_schema` (*request*, *ressource*)

Generate the main endpoint of schema. Return the list of all print app.datastore.modelendpoints available

DataStores

class `rest_api_framework.datastore.base.`**`DataStore`**(*ressource_config*, *model*, ***options*)
define a source of data. Can be anything fron database to other api, files and so one

`get` (*identifier*)
Should return a dictionnary representing the ressource matching the identifier or raise a NotFound exception.

Note: Not implemented by base DataStore class

`create` (*data*)
data is a dict containing the representation of the ressource. This method should call `validate()`, create the data in the datastore and return the ressource identifier

Not implemented by base DataStore class

`update` (*obj*, *data*)
should be able to call `get()` to retreive the object to be updated, `validate_fields()` and return the updated object

Note: Not implemented by base DataStore class

`delete` (*identifier*)
should be able to validate the existence of the object in the ressource and remove it from the datastore

Note: Not implemented by base DataStore class

`get_list` (*offset=None*, *count=None*, ***kwargs*)
This method is called each time you want a set of data. Data could be paginated and filtered. Should call `filter()` and return `paginate()`

Note: Not implemented by base DataStore class

`filter` (***kwargs*)
should return a way to filter the ressource according to kwargs. It is not mandatory to actualy retreive the ressources as they will be paginated just after the filter call. If you retreive the wole filtered ressources you loose the pagination advantage. The point here is to prepare the filtering. Look at SQLiteDataStore.filter for an example.

Note: Not implemented by base DataStore class

`paginate` (*data*, *offset*, *count*)
Paginate sould return all the object if no pagination options have been set or only a subset of the ressources if pagination options exists.

validate(*data*)
Check if data send are valid for object creation. Validate Chek that each required fields are in data and check for their type too.

Used to create new ressources

validate_fields(*data*)
Validate only some fields of the ressource. Used to update existing objects

class rest_api_framework.datastore.simple.**PythonListDataStore**(*ressource_config*, *model*, ***options*)
Bases: *rest_api_framework.datastore.base.DataStore*

a datastore made of list of dicts

get(*identifier*)
return an object matching the uri or None

get_list(*offset=0*, *count=None*, ***kwargs*)
return all the objects. paginated if needed

update(*obj*, *data*)
Update a single object

class rest_api_framework.datastore.sql.**SQLiteDataStore**(*ressource_config*, *model*, ***options*)
Bases: *rest_api_framework.datastore.base.DataStore*

Define a sqlite datastore for your ressource. you have to give __init__ a data parameter containing the information to connect to the database and to the table.

example:

```
data={"table": "tweets",
      "name": "test.db"}
model = ApiModel
datastore = SQLiteDataStore(data, **options)
```

SQLiteDataStore implement a naive wrapper to convert Field types into database type.

- int will be saved in the database as INTEGER
- float will be saved in the database as REAL
- basestring will be saved in the database as TEXT
- if the Field type is PKField, is a will be saved as PRIMARY KEY AUTOINCREMENT

As soon as the datastore is instanciated, the database is create if it does not exists and table is created too

Note:

- It is not possible to use :memory database either. The connection is closed after each operations

get_connector()
return a sqlite3 connection to communicate with the table define in self.db

filter(***kwargs*)
Change kwargs["query"] with "WHERE X=Y statements". The filtering will be done with the actual evaluation of the query in *paginate()* the sql can then be lazy

paginate (*data*, ***kwargs*)
paginate the result of filter using ids limits. Obviously, to work properly, you have to set the start to the last ids you receive from the last call on this method. The max number of row this method can give back depend on the paginate_by option.

get_list (***kwargs*)
return all the objects, paginated if needed, fitered if filters have been set.

get (*identifier*)
Return a single row or raise NotFound

create (*data*)
Validate the data with `base.DataStore.validate()` And, if data is valid, create the row in database and return it.

update (*obj*, *data*)
Retreive the object to be updated (`get()` will raise a NotFound error if the row does not exist)

Validate the fields to be updated and return the updated row

delete (*identifier*)
Retreive the object to be updated

(`get()` will raise a NotFound error if the row does not exist)

Return None on success, Raise a 400 error if foreign key constrain prevent delete.

Views

class `rest_api_framework.views.`**JsonResponse** (*model*, *ressource_name*, *formaters=['add_ressource_uri']*, ***options*)
A werkzeug Response rendering a json representation of the object(s) This class is callable. you should do :

```
view = JsonResponse(model, ressource_name, formaters=formaters,
                    **options)
return view(objects)
```

format (*objs*)
Format the output using formaters listed in self.formaters

Optional modules

Authentication

class `rest_api_framework.authentication.`**Authentication**
Manage the authentication of a request. Must implement the get_user method

get_user (*identifier*)
Must return a user if authentication is successfull, None otherwise

class `rest_api_framework.authentication.`**ApiKeyAuthentication** (*datastore*, *identifier='apikey'*)
Authentication based on an apikey stored in a datastore.

get_user (*request*)
return a user or None based on the identifier found in the request query parameters.

class `rest_api_framework.authentication.`**`BasicAuthentication`**(*datastore*)
Implement the Basic Auth authentication http://fr.wikipedia.org/wiki/HTTP_Authentification

`get_user`(*request*)
return a user or None based on the Authorization: Basic header found in the request. login and password are Base64 encoded string : "login:password"

Authorization

class `rest_api_framework.authentication.`**`Authorization`**(*authentication*)
Check if an authenticated request can perform the given action.

`check_auth`(*request*)
Return None if the request user is authorized to perform this action, raise Unauthorized otherwise

Parameters **`request`** (`werkzeug.wrappers.Request`) –

class `rest_api_framework.authentication.`**`Authorization`**(*authentication*)
Check if an authenticated request can perform the given action.

`check_auth`(*request*)
Return None if the request user is authorized to perform this action, raise Unauthorized otherwise

Parameters **`request`** (`werkzeug.wrappers.Request`) –

Pagination

class `rest_api_framework.pagination.`**`Pagination`**(*max_result*, *offset_key='offset'*, *count_key='count'*)
The base implementation of Pagination. __init__ define max, offset and count.

`paginate`(*request*)
return an offset, a count and the request kwargs without pagination parameters

Partials

Enable partials response from the api. With partials response, only a subset of fields are send back to the request user.

DataStore are responsible for implementing partial options

class `rest_api_framework.partials.`**`Partial`**(*partial_keyword='fields'*)
The base implementation of partial response.

`get_partials`(***kwargs*)
This partial implementation wait for a list of fields separated by comma. Other implementations are possible. Just inherit from this base class and implement your own get_partials method.

get_partials does not check that the fields are part of the model. Datastore get_list will check for it and raise an error if needed.

Rate Limit

Handle the rate limit option for a Controller.

exception `rest_api_framework.ratelimit.`**`TooManyRequest`**(*description=None*)
Implement the 429 status code (see http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html for details)

class `rest_api_framework.ratelimit.`**`RateLimit`**(*datastore*, *interval=6000*, *quota=100*)

Rate limit a user depending on the datetime of the request, the number of previous requests and the rate-limit strategy

`check_limit`(*request*)

Implment the rate-limit method should first authenticate the user, then check his rate-limit quota based on the request. If request is not rate-limited, should increment the rate-limit counter.

Return None if the request is not rate-limited. raise HttpError with a 429 code otherwise

CHAPTER 4

A Full working example

```
from rest_api_framework import models
from rest_api_framework.datastore import SQLiteDataStore
from rest_api_framework.views import JsonResponse
from rest_api_framework.controllers import Controller
from rest_api_framework.datastore.validators import UniqueTogether
from rest_api_framework.pagination import Pagination


class UserModel(models.Model):
    """
    Define how to handle and validate your data.
    """
    fields = [models.StringField(name="first_name", required=True),
              models.StringField(name="last_name", required=True),
              models.PkField(name="id", required=True)
              ]


def remove_id(response, obj):
    """
    Do not show the id in the response.
    """
    obj.pop(response.model.pk_field.name)
    return obj


class UserEndPoint(Controller):
    ressource = {
        "ressource_name": "users",
        "ressource": {"name": "adress_book.db", "table": "users"},
        "model": UserModel,
        "datastore": SQLiteDataStore,
        "options": {"validators": [UniqueTogether("first_name", "last_name")]}
        }
    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"],
        "options": {"pagination": Pagination(20)}
        }

    view = {"response_class": JsonResponse,
            "options": {"formaters": ["add_ressource_uri", remove_id]}}


if __name__ == '__main__':

    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([UserEndPoint])
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index