# Cornice Documentation

***Release 2.3.0***

**Mozilla Services**

February 09, 2017

Contents

**Cornice** provides helpers to build & document REST-ish Web Services with Pyramid, with decent default behaviors. It takes care of following the HTTP specification in an automated way where possible.

We designed and implemented cornice in a really simple way, so it is easy to use and you can get started in a matter of minutes.

CHAPTER 1

# Show me some code!

A **full** Cornice WGSI application looks like this (this example is taken from the demoapp project):

```
from collections import defaultdict

from pyramid.httpexceptions import HTTPForbidden
from pyramid.view import view_config

from cornice import Service


user_info = Service(name='users',
                    path='/{username}/info',
                    description='Get and set user data.')

_USERS = defaultdict(dict)


@user_info.get()
def get_info(request):
    """Returns the public information about a **user**.

    If the user does not exists, returns an empty dataset.
    """
    username = request.matchdict['username']
    return _USERS[username]


@user_info.post()
def set_info(request):
    """Set the public information for a **user**.

    You have to be that user, and *authenticated*.

    Returns *True* or *False*.
    """
    username = request.authenticated_userid
    if request.matchdict["username"] != username:
        raise HTTPForbidden()
    _USERS[username] = request.json_body
    return {'success': True}


@view_config(route_name="whoami", permission="authenticated", renderer="json")
```

```
def whoami(request):
    """View returning the authenticated user's credentials."""
    username = request.authenticated_userid
    principals = request.effective_principals
    return {"username": username, "principals": principals}
```

What Cornice will do for you here is:

- automatically raise a 405 if a DELETE or a PUT is called on **/{username}/info**
- provide a validation framework that will return a nice JSON structure in Bad Request 400 responses explaining what's wrong.
- provide an acceptable **Content-Type** whenever you send an HTTP "Accept" header to it, resulting in a *406 Not Acceptable* with the list of acceptable ones if it can't answer.

Please follow up with Exhaustive features list to get the picture.

CHAPTER 2

# Documentation content

## 2.1 QuickStart for people in a hurry

You are in a hurry, so we'll assume you are familiar with Pip ;)

To use Cornice, install it:

```
$ pip install cornice
```

To start from scratch, you can use a Cookiecutter project template:

```
$ pip install cookiecutter
$ cookiecutter gh:Cornices/cookiecutter-cornice
...
```

The template creates a working Cornice application.

**Note:** If you're familiar with Pyramid and just want to add *cornice* to an already existing project, you'll just need to include `cornice` in your project:

```
config.include("cornice")
```

You can then start poking at the `views.py` file.

For example, let's define a service where you can **GET** and **POST** a value at **/values/{value}**, where *value* is an ascii value representing the name of the value.

The `views` module can look like this:

```
from cornice import Service

_VALUES = {}

values = Service(name='foo',
                 path='/values/{value}',
                 description="Cornice Demo")

@values.get()
def get_value(request):
    """Returns the value.
    """
    key = request.matchdict['value']
    return _VALUES.get(key)
```

```
@values.post()
def set_value(request):
    """Set the value.

    Returns *True* or *False*.
    """
    key = request.matchdict['value']
    try:
        # json_body is JSON-decoded variant of the request body
        _VALUES[key] = request.json_body
    except ValueError:
        return False
    return True
```

**Note:** By default, Cornice uses a Json renderer.

Run your Cornice application with:

```
$ pserve project.ini --reload
```

Set a key-value using Curl:

```
$ curl -X POST http://localhost:6543/values/foo -d '{"a": 1}'
```

Check out what is stored in a `foo` value at http://localhost:6543/values/foo

## 2.2 Full tutorial

Let's create a full working application with **Cornice**. We want to create a light messaging service.

You can find its whole source code at https://github.com/Cornices/examples/blob/master/messaging

Features:

- users can register to the service
- users can list all registered users
- users can send messages
- users can retrieve the latest messages
- messages have three fields: sender, content, color (red or black)
- adding a message is done through authentication

Limitations:

- there's a single channel for all messages.
- if a user with the same name is already registered, he cannot register.
- all messages and users are kept in memory.

### 2.2.1 Design

The application provides two services:

- **users**, at **/users**: where you can list all users or register a new one
- **messages**, at **/**: where you can read the messages or add new ones

On the server, the data is kept in memory.

We'll provide a single CLI client in Python, using Curses.

### 2.2.2 Setting up the development environment

Make sure you have **virtualenv** (see http://pypi.python.org/pypi/virtualenv).

Create a new directory and a virtualenv in it:

```
$ mkdir messaging
$ cd messaging
$ virtualenv --no-site-packages .
```

Once you have it, install Cornice in it with Pip:

```
$ bin/pip install cornice
```

We provide a Cookiecutter template you can use to create a new application:

```
$ bin/pip install cookiecutter
$ bin/cookiecutter gh:Cornices/cookiecutter-cornice
repo_name [myapp]: messaging
project_title [My Cornice application.]: Cornice tutorial
```

Once your application is generated, go there and call *develop* against it:

```
$ cd messaging
$ ../bin/python setup.py develop
...
```

The application can now be launched via embedded Pyramid `pserve`, it provides a default "Hello" service check:

```
$ ../bin/pserve messaging.ini
Starting server in PID 7618.
serving on 0.0.0.0:6543 view at http://127.0.0.1:6543
```

Once the application is running, visit http://127.0.0.1:6543 in your browser and make sure you get:

```
{'Hello': 'World'}
```

You should also get the same results calling the URL via Curl:

```
$ curl -i http://0.0.0.0:6543/
```

This will result:

```
HTTP/1.1 200 OK
Content-Length: 18
Content-Type: application/json; charset=UTF-8
Date: Tue, 12 May 2015 13:23:32 GMT
Server: waitress

{"Hello": "World"}
```

### 2.2.3 Defining the services

Let's open the file in `messaging/views.py`, it contains all the Services:

```
from cornice import Service

hello = Service(name='hello', path='/', description="Simplest app")

@hello.get()
def get_info(request):
    """Returns Hello in JSON."""
    return {'Hello': 'World'}
```

#### Users management

We're going to get rid of the Hello service, and change this file in order to add our first service - the users management

```
from cornice import Service

_USERS = {}

users = Service(name='users', path='/users', description="User registration")

@users.get(validators=valid_token)
def get_users(request):
    """Returns a list of all users."""
    return {'users': _USERS.keys()}

@users.post(validators=unique)
def create_user(request):
    """Adds a new user."""
    user = request.validated['user']
    _USERS[user['name']] = user['token']
    return {'token': '%s-%s' % (user['name'], user['token'])}

@users.delete(validators=valid_token)
def delete_user(request):
    """Removes the user."""
    name = request.validated['user']
    del _USERS[name]
    return {'Goodbye': name}
```

What we have here is 3 methods on **/users**:

- **GET**: returns the list of users names – the keys of _USERS
- **POST**: adds a new user and returns a unique token
- **DELETE**: removes the user.

Remarks:

- **POST** uses the **unique** validator to make sure that the user name is not already taken. That validator is also in charge of generating a unique token associated with the user.
- **GET** users the **valid_token** to verify that a **X-Messaging-Token** header is provided in the request, with a valid token. That also identifies the user.
- **DELETE** also identifies the user then removes it.

Validators are filling the **request.validated** mapping, the service can then use.

```
import os
import binascii

from pyramid.httpexceptions import HTTPUnauthorized
from cornice import Service


def _create_token():
    return binascii.b2a_hex(os.urandom(20))


def valid_token(request):
    header = 'X-Messaging-Token'
    htoken = request.headers.get(header)
    if htoken is None:
        raise HTTPUnauthorized()
    try:
        user, token = htoken.split('-', 1)
    except ValueError:
        raise HTTPUnauthorized()

    valid = user in _USERS and _USERS[user] == token
    if not valid:
        raise HTTPUnauthorized()

    request.validated['user'] = user


def unique(request):
    name = request.body
    if name in _USERS:
        request.errors.add('url', 'name', 'This user exists!')
    else:
        user = {'name': name, 'token': _create_token()}
        request.validated['user'] = user
```

When the validator finds errors, it adds them to the **request.errors** mapping, and that will return a 400 with the errors.

Let's try our application so far with CURL:

```
$ curl http://localhost:6543/users
{"status": 401, "message": "Unauthorized"}

$ curl -X POST http://localhost:6543/users -d 'tarek'
{"token": "tarek-a15fa2ea620aac8aad3e1b97a64200ed77dc7524"}

$ curl http://localhost:6543/users -H "X-Messaging-Token:tarek-a15fa2ea620aac8aad3e1b97a64200ed77dc75
{"users": ["tarek"]}

$ curl -X DELETE http://localhost:6543/users -H "X-Messaging-Token:tarek-a15fa2ea620aac8aad3e1b97a642
{"Goodbye": "tarek"}
```

### Messages management

Now that we have users, let's post and get messages. This is done via two very simple functions we're adding in the `views.py` file:

```
_MESSAGES = []

messages = Service(name='messages', path='/', description="Messages")

@messages.get()
def get_messages(request):
    """Returns the 5 latest messages"""
    return _MESSAGES[:5]


@messages.post(validators=(valid_token, valid_message))
def post_message(request):
    """Adds a message"""
    _MESSAGES.insert(0, request.validated['message'])
    return {'status': 'added'}
```

The first one simply returns the five first messages in a list, and the second one inserts a new message in the beginning of the list.

The **POST** uses two validators:

- `valid_token()`: the function we used previously that makes sure the user is registered
- `valid_message()`: a function that looks at the message provided in the POST body, and puts it in the validated dict.

Here's the `valid_message()` function:

```
import json

def valid_message(request):
    try:
        message = json.loads(request.body)
    except ValueError:
        request.errors.add('body', 'message', 'Not valid JSON')
        return

    # make sure we have the fields we want
    if 'text' not in message:
        request.errors.add('body', 'text', 'Missing text')
        return

    if 'color' in message and message['color'] not in ('red', 'black'):
        request.errors.add('body', 'color', 'only red and black supported')
    elif 'color' not in message:
        message['color'] = 'black'

    message['user'] = request.validated['user']
    request.validated['message'] = message
```

This function extracts the json body, then checks that it contains a text key at least. It adds a color or use the one that was provided, and reuse the user name provided by the previous validator with the token control.

### 2.2.4 The Client

A simple client to use against our service can do three things:

1. let the user register a name

2. poll for the latest messages
3. let the user send a message !

Without going into great details, there's a Python CLI against messaging that uses Curses.

See https://github.com/Cornices/examples/blob/master/messaging/messaging/client.py

## 2.3 Defining services

As mentioned in the *QuickStart for people in a hurry* and *Full tutorial*, services are defined this way:

```
from cornice import Service

flush = Service(name='flush',
                description='Clear database content',
                path='/__flush__')

@flush.post()
def flush_post(request):
    return {"Done": True}
```

See *cornice.service.Service* for an exhaustive list of options.

### 2.3.1 Imperatively

Here is an example of how to define cornice services in an imperative way:

```
def flush_post(request):
    return {"Done": True}

flush = Service(name='flush',
                description='Clear database content',
                path='/__flush__')

flush.add_view("POST", flush_post, **kwargs):

def includeme(config):
    config.add_cornice_service(flush)
    # or
    config.scan("PATH_TO_THIS_MODULE")
```

### 2.3.2 Custom error handler

```
from pyramid.httpexceptions import HTTPBadRequest

def my_error_handler(request):
    first_error = request.errors[0]
    body = {'description': first_error['description']}

    response = HTTPBadRequest()
    response.body = json.dumps(body).encode("utf-8")
    response.content_type = 'application/json'
    return response
```

```
flush = Service(name='flush',
                path='/__flush__',
                error_handler=my_error_handler)
```

### 2.3.3 CORS

When enabling CORS, Cornice will take automatically define `OPTIONS` views and appropriate headers validation.

```
flush = Service(name='flush',
                description='Clear database content',
                path='/__flush__',
                cors_origins=('*',),
                cors_max_age=3600)
```

There are also a number of parameters that are related to the support of CORS (Cross Origin Resource Sharing). You can read the CORS specification at http://www.w3.org/TR/cors/ and see *the exhaustive list of options in Cornice*.

**See also:**

https://blog.mozilla.org/services/2013/02/04/implementing-cross-origin-resource-sharing-cors-for-cornice/

### 2.3.4 Route factory support

When defining a service, you can provide a route factory, just like when defining a pyramid route.

For example:

```
flush = Service(name='flush', path='/__flush__', factory=user_factory)
```

## 2.4 Defining resources

Cornice is also able to handle REST "resources" for you. You can declare a class with some put, post, get etc. methods (the HTTP verbs) and they will be registered as handlers for the appropriate methods / services.

Here is how you can register a resource:

```
from cornice.resource import resource

_USERS = {1: {'name': 'gawel'}, 2: {'name': 'tarek'}}

@resource(collection_path='/users', path='/users/{id}')
class User(object):

    def __init__(self, request):
        self.request = request

    def collection_get(self):
        return {'users': _USERS.keys()}

    def get(self):
        return _USERS.get(int(self.request.matchdict['id']))

    def collection_post(self):
        print(self.request.json_body)
```

```
        _USERS[len(_USERS) + 1] = self.request.json_body
        return True
```

### 2.4.1 Imperatively

Here is an example of how to define cornice resources in an imperative way:

```
from cornice import resource

class User(object):

    def __init__(self, request):
        self.request = request

    def collection_get(self):
        return {'users': _USERS.keys()}

    def get(self):
        return _USERS.get(int(self.request.matchdict['id']))

resource.add_view(User.get, renderer='json')
user_resource = resource.add_resource(User, collection_path='/users', path='/users/{id}')

def includeme(config):
    config.add_cornice_resource(user_resource)
    # or
    config.scan("PATH_TO_THIS_MODULE")
```

As you can see, you can define methods for the collection (it will use the **path** argument of the class decorator. When defining collection_* methods, the path defined in the **collection_path** will be used.

### 2.4.2 Validators and filters

You also can register validators and filters that are defined in your *@resource* decorated class, like this:

```
from cornice.resource import resource, view

@resource(path='/users/{id}')
class User(object):

    def __init__(self, request):
        self.request = request

    @view(validators=('validate_req',))
    def get(self):
        # return the list of users

    def validate_req(self, request):
        # validate the request
```

### 2.4.3 Registered routes

Cornice uses a default convention for the names of the routes it registers.

When defining resources, the pattern used is `collection_<service_name>` (it prepends `collection_` to the service name) for the collection service.

### 2.4.4 Route factory support

When defining a resource, you can provide a route factory, just like when defining a pyramid route. Cornice will then pass its result into the `__init__` of your service.

For example:

```
@resource(path='/users', factory=user_factory)
class User(object):

    def __init__(self, request, context=None):
        self.request = request
        self.user = context
```

## 2.5 Validation features

Cornice provides a way to control the request before it's passed to the code. A validator is a simple callable that gets the request object and some keywords arguments, and fills **request.errors** in case the request isn't valid.

Validators can also convert values and saves them so they can be reused by the code. This is done by filling the **request.validated** dictionary.

Once the request had been sent to the view, you can filter the results using so called filters. This document describe both concepts, and how to deal with them.

### 2.5.1 Disabling or adding filters/validators

Some validators and filters are activated by default, for all the services. In case you want to disable them, or if you

You can register a filter for all the services by tweaking the *DEFAULT_FILTER* parameter:

```
from cornice.validators import DEFAULT_FILTERS

def includeme(config):
    DEFAULT_FILTERS.append(your_callable)
```

(this also works for validators)

You also can add or remove filters and validators for a particular service. To do that, you need to define its *default_validators* and *default_filters* class parameters.

### 2.5.2 Dealing with errors

When validating inputs using the different validation mechanisms (described in this document), Cornice can return errors. In case it returns errors, it will do so in JSON by default.

The default returned JSON object is a dictionary of the following form:

```
{
    'status': 'error',
    'errors': errors
}
```

With `errors` being a JSON dictionary with the keys "location", "name" and "description".

- **location** is the location of the error. It can be "querystring", "header" or "body"
- **name** is the eventual name of the value that caused problems
- **description** is a description of the problem encountered.

You can override the default JSON error handler for a view with your own callable. The following function, for instance, returns the error response with an XML document as its payload:

```
def xml_error(request):
    errors = request.errors
    lines = ['<errors>']
    for error in errors:
        lines.append('<error>'
                     '<location>%(location)s</location>'
                     '<type>%(name)s</type>'
                     '<message>%(description)s</message>'
                     '</error>' % error)
    lines.append('</errors>')
    return HTTPBadRequest(body=''.join(lines),
                          content_type='application/xml')
```

Configure your views by passing your handler as `error_handler`:

```
@service.post(validators=my_validator, error_handler=xml_error)
def post(request):
    return {'OK': 1}
```

### 2.5.3 Validators

Cornice provide a simple mechanism to let you validate incoming requests before they are processed by your views.

#### Validation using custom callables

Let's take an example: we want to make sure the incoming request has an **X-Verified** header. If not, we want the server to return a 400:

```
from cornice import Service

foo = Service(name='foo', path='/foo')


def has_paid(request, **kwargs):
    if not 'X-Verified' in request.headers:
        request.errors.add('header', 'X-Verified', 'You need to provide a token')

@foo.get(validators=has_paid)
def get_value(request):
    """Returns the value.
    """
    return 'Hello'
```

Notice that you can chain the validators by passing a sequence to the **validators** option.

#### Changing the status code from validators

You also can change the status code returned from your validators. Here is an example of this:

```
def user_exists(request):
    if not request.POST['userid'] in userids:
        request.errors.add('body', 'userid', 'The user id does not exist')
        request.errors.status = 404
```

#### Doing validation and filtering at class level

If you want to use class methods to do validation, you can do so by passing the *klass* parameter to the *hook_view* or *@method* decorators, plus a string representing the name of the method you want to invoke on validation.

Take care, though, because this only works if the class you are using has an *__init__* method which takes a *request* as the first argument.

This means something like this:

```
class MyClass(object):
    def __init__(self, request):
        self.request = request

    def validate_it(self, request, **kw):
        # pseudo-code validation logic
        if whatever is wrong:
            request.errors.add('body', description="Something is wrong")

@service.get(klass=MyClass, validators=('validate_it',))
def view(request):
    return "ok"
```

### 2.5.4 Media type validation

There are two flavors of media/content type validations Cornice can apply to services:

- *Content negotiation* checks if Cornice is able to respond with an appropriate **response body** content type requested by the client sending an `Accept` header. Otherwise it will croak with a `406 Not Acceptable`.
- *Request media type* validation will match the `Content-Type` **request header** designating the **request body** content type against a list of allowed content types. When failing on that, it will croak with a `415 Unsupported Media Type`.

#### Content negotiation

Validate the `Accept` header in http requests against a defined or computed list of internet media types. Otherwise, signal `406 Not Acceptable` to the client.

##### Basics

By passing the *accept* argument to the service definition decorator, we define the media types we can generate http **response** bodies for:

```
@service.get(accept="text/html")
def foo(request):
    return 'Foo'
```

When doing this, Cornice automatically deals with egress content negotiation for you.

If services don't render one of the appropriate response body formats asked for by the requests HTTP **Accept** header, Cornice will respond with a http status of `406 Not Acceptable`.

The *accept* argument can either be a string or a list of accepted values made of internet media type(s) or a callable returning the same.

#### Using callables

When a callable is specified, it is called *before* the request is passed to the destination function, with the *request* object as an argument.

The callable obtains the request object and returns a list or a single scalar value of accepted media types:

```
def _accept(request):
    # interact with request if needed
    return ("text/xml", "text/json")

@service.get(accept=_accept)
def foo(request):
    return 'Foo'
```

**See also:**

https://developer.mozilla.org/en-US/docs/HTTP/Content_negotiation

#### Error responses

When requests are rejected, an appropriate error response is sent to the client using the configured *error_handler*. To give the service consumer a hint about the valid internet media types to use for the `Accept` header, the error response contains a list of allowed types.

When using the default json *error_handler*, the response might look like this:

```
{
    'status': 'error',
    'errors': [
        {
            'location': 'header',
            'name': 'Accept',
            'description': 'Accept header should be one of ["text/xml", "text/json"]'
        }
    ]
}
```

### Request media type

Validate the `Content-Type` header in http requests against a defined or computed list of internet media types. Otherwise, signal `415 Unsupported Media Type` to the client.

### Basics

By passing the *content_type* argument to the service definition decorator, we define the media types we accept as http **request** bodies:

```
@service.post(content_type="application/json")
def foo(request):
    return 'Foo'
```

All requests sending a different internet media type using the HTTP **Content-Type** header will be rejected with a http status of `415 Unsupported Media Type`.

The *content_type* argument can either be a string or a list of accepted values made of internet media type(s) or a callable returning the same.

### Using callables

When a callable is specified, it is called *before* the request is passed to the destination function, with the *request* object as an argument.

The callable obtains the request object and returns a list or a single scalar value of accepted media types:

```
def _content_type(request):
    # interact with request if needed
    return ("text/xml", "application/json")

@service.post(content_type=_content_type)
def foo(request):
    return 'Foo'
```

The match is done against the plain internet media type string without additional parameters like `charset=utf-8` or the like.

**See also:**

WebOb documentation: Return the content type, but leaving off any parameters

### Error responses

When requests are rejected, an appropriate error response is sent to the client using the configured *error_handler*. To give the service consumer a hint about the valid internet media types to use for the `Content-Type` header, the error response contains a list of allowed types.

When using the default json *error_handler*, the response might look like this:

```
{
    'status': 'error',
    'errors': [
        {
            'location': 'header',
            'name': 'Content-Type',
            'description': 'Content-Type header should be one of ["text/xml", "application/json"]'
        }
    ]
}
```

### 2.5.5 Managing ACLs

You can also specify a way to deal with ACLs: pass in a function that takes a request and returns an ACL, and that ACL will be applied to all views in the service:

```
foo = Service(name='foo', path='/foo', acl=_check_acls)
```

### 2.5.6 Filters

Cornice can also filter the response returned by your views. This can be useful if you want to add some behaviour once a response has been issued.

Here is how to define a validator for a service:

```
foo = Service(name='foo', path='/foo', filters=your_callable)
```

You can just add the filter for a specific method:

```
@foo.get(filters=your_callable)
def foo_get(request):
    """some description of the validator for documentation reasons"""
    pass
```

In case you would like to register a filter for all the services but one, you can use the *exclude* parameter. It works either on services or on methods:

```
@foo.get(exclude=your_callable)
```

## 2.6 Schema validation

Validating requests data using a schema is a powerful pattern.

As you would do for a database table, you define some fields and their type, and make sure that incoming requests comply.

There are many schema libraries in the Python ecosystem you can use. The most known ones are Colander & formencode.

You can do schema validation using either those libraries or either custom code.

Using a schema is done in 2 steps:

1/ linking a schema to your service definition 2/ implement a validator that uses the schema to verify the request

Here's a dummy example:

```
def my_validator(request, **kwargs):
    schema = kwargs['schema']
    # do something with the schema

schema = {'id': int, 'name': str}

@service.post(schema=schema, validators=(my_validator,))
def post(request):
    return {'OK': 1}
```

Cornice will call `my_validator` with the incoming request, and will provide the schema in the keywords.

### 2.6.1 Using Colander

Colander (http://docs.pylonsproject.org/projects/colander/en/latest/) is a validation framework from the Pylons project that can be used with Cornice's validation hook to control a request and deserialize its content into objects.

Cornice provides a helper to ease Colander integration.

To describe a schema, using Colander and Cornice, here is how you can do:

```
import colander

from cornice import Service
from cornice.validators import colander_body_validator

class SignupSchema(colander.MappingSchema):
    username = colander.SchemaNode(colander.String())

@signup.post(schema=SignupSchema(), validators=(colander_body_validator,))
def signup_post(request):
    username = request.validated['username']
    return {'success': True}
```

#### Dynamic schemas

If you want to do specific things with the schema at validation step, like having a schema per request method, you can provide whatever you want as the schema key and built a custom validator.

Example:

```
def dynamic_schema(request):
    if request.method == 'POST':
        schema = foo_schema
    elif request.method == 'PUT':
        schema = bar_schema
    return schema


def my_validator(request, **kwargs):
    kwargs['schema'] = dynamic_schema(request)
    return colander_body_validator(request, **kwargs)


@service.post(validators=(my_validator,))
def post(request):
    return request.validated
```

#### Multiple request attributes

If you have complex use-cases where data has to be validated accross several locations of the request (like querystring, body etc.), Cornice provides a validator that takes an additionnal level of mapping for `body`, `querystring`, `path` or `headers` instead of the former `location` attribute on schema fields.

The `request.validated` hences reflects this additional level.

```
from cornice.validators import colander_validator

class Querystring(colander.MappingSchema):
```

```
    referrer = colander.SchemaNode(colander.String(), missing=colander.drop)

class Payload(colander.MappingSchema):
    username = colander.SchemaNode(colander.String())

class SignupSchema(colander.MappingSchema):
    body = Payload()
    querystring = Querystring()

signup = cornice.Service()

@signup.post(schema=SignupSchema(), validators=(colander_validator,))
def signup_post(request):
    username = request.validated['body']['username']
    referrer = request.validated['querystring']['referrer']
    return {'success': True}
```

This allows to have validation at the schema level that validates data from several places on the request:

```
class SignupSchema(colander.MappingSchema):
    body = Payload()
    querystring = Querystring()

    def deserialize(self, cstruct=colander.null):
        appstruct = super(SignupSchema, self).deserialize(cstruct)
        username = appstruct['body']['username']
        referrer = appstruct['querystring'].get('referrer')
        if username == referred:
            self.raise_invalid('Referrer cannot be the same as username')
        return appstruct
```

Cornice provides built-in support for JSON and HTML forms (`application/x-www-form-urlencoded`) input validation using the provided colander validators.

If you need to validate other input formats, such as XML, you need to implement your own deserializer and pass it to the service.

The general pattern in this case is:

```
from cornice.validators import colander_body_validator

def my_deserializer(request):
    return extract_data_somehow(request)


@service.post(schema=MySchema(),
              deserializer=my_deserializer,
              validators=(colander_body_validator,))
def post(request):
    return {'OK': 1}
```

### 2.6.2 Using formencode

FormEncode (http://www.formencode.org/en/latest/index.html) is yet another validation system that can be used with Cornice.

For example, if you want to make sure the optional query option **max** is an integer, and convert it, you can use FormEncode in a Cornice validator like this:

```
from formencode import validators

from cornice import Service
from cornice.validators import extract_cstruct

foo = Service(name='foo', path='/foo')

def form_validator(request, **kwargs):
    data = extract_cstruct(request)
    validator = validators.Int()
    try:
        max = data['querystring'].get('max')
        request.validated['max'] = validator.to_python(max)
    except formencode.Invalid, e:
        request.errors.add('querystring', 'max', e.message)

@foo.get(validators=(form_validator,))
def get_value(request):
    """Returns the value.
    """
    return {'posted': request.validated}
```

### 2.6.3 See also

Several libraries exist in the wild to validate data in Python and that can easily be plugged with Cornice.

- JSONSchema (https://pypi.python.org/pypi/jsonschema)
- Cerberus (https://pypi.python.org/pypi/Cerberus)
- marshmallow (https://pypi.python.org/pypi/marshmallow)

## 2.7 Testing

### 2.7.1 Running tests

To run all tests in all Python environments configured in `tox.ini`, just setup `tox` and run it inside the toplevel project directory:

```
tox
```

To run a single test inside a specific Python environment, do e.g.:

```
tox -e py27 tests/test_validation.py:TestServiceDefinition.test_content_type_missing
```

or:

```
tox -e py27 tests.test_validation:TestServiceDefinition.test_content_type_missing
```

### 2.7.2 Testing cornice services

Testing is nice and useful. Some folks even said it helped saving kittens. And childs. Here is how you can test your Cornice's applications.

Let's suppose you have this service definition:

```
from pyramid.config import Configurator

from cornice import Service

service = Service(name="service", path="/service")


def has_payed(request, **kwargs):
    if not 'paid' in request.GET:
        request.errors.add('body', 'paid', 'You must pay!')


@service.get(validators=(has_payed,))
def get1(request):
    return {"test": "succeeded"}


def includeme(config):
    config.include("cornice")
    config.scan("absolute.path.to.this.service")


def main(global_config, **settings):
    config = Configurator(settings={})
    config.include(includeme)
    return config.make_wsgi_app()
```

We have done three things here:

- setup a service, using the *Service* class and define our services with it
- register the app and cornice to pyramid in the *includeme* function
- define a *main* function to be used in tests

To test this service, we will use **webtest**, and the *TestApp* class:

```
from webtest import TestApp
import unittest

from yourapp import main

class TestYourApp(unittest.TestCase):

    def test_case(self):
        app = TestApp(main({}))
        app.get('/service', status=400)
```

## 2.8 Exhaustive features list

As you may have noticed, Cornice does some validation for you. This document aims at documenting all those behaviours so you are not surprised if Cornice does it for you without noticing.

### 2.8.1 Validation

#### Errors

When validating contents, Cornice will automatically throw a 400 error if the data is invalid. Along with the 400 error, the body will contain a JSON dict which can be parsed to know more about the problems encountered.

#### Method not allowed

In cornice, one path equals one service. If you call a path with the wrong method, a *405 Method Not Allowed* error will be thrown (and not a 404), like specified in the HTTP specification.

#### Authorization

Authorization can be done using the *acl* parameter. If the authentication or the authorization fails at this stage, a 401 or 403 error is returned, depending on the cases.

#### Content negotiation

This relates to **response body** internet media types aka. egress content types.

Each method can specify a list of internet media types it can **respond** with. Per default, *text/html* is assumed. In the case the client requests an invalid media type via *Accept* header, cornice will return a *406 Not Acceptable* with an error message containing the list of available response content types for the particular URI and method.

#### Request media type

This relates to **request body** internet media types aka. ingress content types.

Each method can specify a list of internet media types it accepts as **request** body format. Per default, any media type is allowed. In the case the client sends a request with an invalid *Content-Type* header, cornice will return a *415 Unsupported Media Type* with an error message containing the list of available request content types for the particular URI and method.

#### Warning when returning JSON lists

JSON lists are subject to security threats, as defined in this document. In case you return a javascript list, a warning will be thrown. It will not however prevent you from returning the array.

This behaviour can be disabled if needed (it can be removed from the list of default filters)

### 2.8.2 URL prefix

It is possible to set a prefix for all your routes. For instance, if you want to prefix all your URIs by `/v1/`.

```
config.route_prefix = 'v2'
config.include("cornice")
```

### 2.8.3 CORS

Cornice can add CORS (Cross Origin Resource Sharing) support to your services. When enabled, it will define the appropriate views (`OPTIONS` methods) and validators (headers etc.).

See *more details...*

## 2.9 Cornice API

### 2.9.1 Service

This document describes the methods proposed by cornice. It is automatically generated from the source code.

**class** `cornice.service.`**`Service`**(*name*, *path*, *description=None*, *cors_policy=None*, *depth=1*, ***kw*)

Contains a service definition (in the definition attribute).

A service is composed of a path and many potential methods, associated with context.

All the class attributes defined in this class or in children are considered default values.

**Parameters**

- **`name`** – The name of the service. Should be unique among all the services.
- **`path`** – The path the service is available at. Should also be unique.
- **`renderer`** – The renderer that should be used by this service. Default value is 'simplejson'.
- **`description`** – The description of what the webservice does. This is primarily intended for documentation purposes.
- **`validators`** – A list of callables to pass the request into before passing it to the associated view.
- **`filters`** – A list of callables to pass the response into before returning it to the client.
- **`accept`** – A list of `Accept` header values accepted for this service (or method if overwritten when defining a method). It can also be a callable, in which case the values will be discovered at runtime. If a callable is passed, it should be able to take the request as a first argument.
- **`content_type`** – A list of `Content-Type` header values accepted for this service (or method if overwritten when defining a method). It can also be a callable, in which case the values will be discovered at runtime. If a callable is passed, it should be able to take the request as a first argument.
- **`factory`** – A factory returning callables which return boolean values. The callables take the request as their first argument and return boolean values. This param is exclusive with the 'acl' one.
- **`acl`** – A callable defining the ACL (returns true or false, function of the given request). Exclusive with the 'factory' option.
- **`permission`** – As for `pyramid.config.Configurator.add_view()`. Note: *acl* and *permission* can also be applied to instance method decorators such as `get()` and `put()`.
- **`klass`** – The class to use when resolving views (if they are not callables)
- **`error_handler`** – A callable which is used to render responses following validation failures. Defaults to 'json_error'.

- **traverse** – A traversal pattern that will be passed on route declaration and that will be used as the traversal path.

There are also a number of parameters that are related to the support of CORS (Cross Origin Resource Sharing). You can read the CORS specification at http://www.w3.org/TR/cors/

**Parameters**

- **cors_enabled** – To use if you especially want to disable CORS support for a particular service / method.
- **cors_origins** – The list of origins for CORS. You can use wildcards here if needed, e.g. ('list', 'of', '*.domain').
- **cors_headers** – The list of headers supported for the services.
- **cors_credentials** – Should the client send credential information (False by default).
- **cors_max_age** – Indicates how long the results of a preflight request can be cached in a preflight result cache.
- **cors_expose_all_headers** – If set to True, all the headers will be exposed and considered valid ones (Default: True). If set to False, all the headers need be explicitly mentioned with the cors_headers parameter.
- **cors_policy** – It may be easier to have an external object containing all the policy information related to CORS, e.g:

```
>>> cors_policy = {'origins': ('*',), 'max_age': 42,
...                'credentials': True}
```

You can pass a dict here and all the values will be unpacked and considered rather than the parameters starting by *cors_* here.

See https://pyramid.readthedocs.io/en/1.0-branch/glossary.html#term-acl for more information about ACLs.

Service cornice instances also have methods `get()`, `post()`, `put()`, `options()` and `delete()` are decorators that can be used to decorate views.

`cornice.service.`**`decorate_view`**(*view*, *args*, *method*, *route_args={}*)

Decorate a given view with cornice niceties.

This function returns a function with the same signature than the one you give as :param view:

**Parameters**

- **view** – the view to decorate
- **args** – the args to use for the decoration
- **method** – the HTTP method
- **route_args** – the args used for the associated route

### 2.9.2 Resource

`cornice.resource.`**`resource`**(*depth=2*, ***kw*)

Class decorator to declare resources.

All the methods of this class named by the name of HTTP resources will be used as such. You can also prefix them by `"collection_"` and they will be treated as HTTP methods for the given collection path (collection_path), if any.

**Parameters**

- **depth** – Witch frame should be looked in default 2.
- **kw** – Keyword arguments configuring the resource.

Here is an example:

```
@resource(collection_path='/users', path='/users/{id}')
```

cornice.resource.**view**(***kw*)

Method decorator to store view arguments when defining a resource with the @resource class decorator

**Parameters** **kw** – Keyword arguments configuring the view.

cornice.resource.**add_view**(*func*, ***kw*)

Method to store view arguments when defining a resource with the add_resource class method

**Parameters**

- **func** – The func to hook to
- **kw** – Keyword arguments configuring the view.

Example:

```
class User(object):

    def __init__(self, request):
        self.request = request

    def collection_get(self):
        return {'users': _USERS.keys()}

    def get(self):
        return _USERS.get(int(self.request.matchdict['id']))

add_view(User.get, renderer='json')
add_resource(User, collection_path='/users', path='/users/{id}')
```

cornice.resource.**add_resource**(*klass*, *depth=1*, ***kw*)

Function to declare resources of a Class.

All the methods of this class named by the name of HTTP resources will be used as such. You can also prefix them by "collection_" and they will be treated as HTTP methods for the given collection path (collection_path), if any.

**Parameters**

- **klass** – The class (resource) on witch to register the service.
- **depth** – Witch frame should be looked in default 2.
- **kw** – Keyword arguments configuring the resource.

Here is an example:

```
class User(object):
    pass

add_resource(User, collection_path='/users', path='/users/{id}')
```

### 2.9.3 Validation

cornice.validators.**extract_cstruct**(*request*)
Extract attributes from the specified *request* such as body, url, path, method, querystring, headers, cookies, and returns them in a single dict object.

**Parameters** **request** (Request) – Current request

**Returns** A mapping containing most request attributes.

**Return type** dict

cornice.validators.**colander_body_validator**(*request*, *schema=None*, *deserializer=None*, ***kwargs*)
Validate the body against the schema defined on the service.

The content of the body is deserialized, validated and stored in the `request.validated` attribute.

**Note:** If no schema is defined, this validator does nothing.

**Parameters**

- **request** (Request) – Current request
- **schema** – The Colander schema
- **deserializer** – Optional deserializer, defaults to *cornice.validators.extract_cstruct()*

cornice.validators.**colander_validator**(*request*, *schema=None*, *deserializer=None*, ***kwargs*)
Validate the full request against the schema defined on the service.

Each attribute of the request is deserialized, validated and stored in the `request.validated` attribute (eg. body in `request.validated['body']`).

**Note:** If no schema is defined, this validator does nothing.

**Parameters**

- **request** (Request) – Current request
- **schema** – The Colander schema
- **deserializer** – Optional deserializer, defaults to *cornice.validators.extract_cstruct()*

### 2.9.4 Errors

**class** cornice.errors.**Errors**(*status=400*)
Holds Request errors

## 2.10 Cornice internals

Internally, Cornice doesn't do a lot of magic. The logic is mainly split in two different locations: the *services.py* module and the *pyramid_hook.py* module.

That's important to understand what they are doing in order to add new features or tweak the existing ones.

### 2.10.1 The Service class

The *cornice.service.Service* class is a container for all the definition information for a particular service. That's what you use when you use the Cornice decorators for instance, by doing things like `@myservice.get(**kwargs)`. Under the hood, all the information you're passing to the service is stored in this class. Into other things you will find there:

- the *name* of the registered service.
- the *path* the service is available at.
- the *description* of the service, if any.
- the *defined_methods* for the current service. This is a list of strings. It shouldn't contain more than one time the same item.

That's for the basic things. The last interesting part is what we call the "definitions". When you add a view to the service with the *add_view* method, it populates the definitions list, like this:

```
self.definitions.append((method, view, args))
```

where *method* is the HTTP verb, *view* is the python callable and *args* are the arguments that are registered with this definition. It doesn't look this important, but this last argument is actually the most important one. It is a python dict containing the filters, validators, content types etc.

There is one thing I didn't talk about yet: how we are getting the arguments from the service class. There is a handy *get_arguments* method, which returns the arguments from another list of given arguments. The goal is to fallback on instance-level arguments or class-level arguments if no arguments are provided at the add_view level. For instance, let's say I have a default service which renders to XML. I set its renderer in the class to "XML".

When I register the information with `cornice.service.Service.add_view()`, `renderer='XML'` will be added automatically in the kwargs dict.

### 2.10.2 Registering the definitions into the Pyramid routing system

Okay, so once you added the services definition using the Service class, you might need to actually register the right routes into pyramid. The `cornice.pyramidhook` module takes care of this for you.

What it does is that it checks all the services registered and call some functions of the pyramid framework on your behalf.

What's interesting here is that this mechanism is not really tied to pyramid. for instance, we are doing the same thing in cornice_sphinx to generate the documentation: use the APIs that are exposed in the Service class and do something from it.

To keep close to the flexibility of Pyramid's routing system, a `traverse` argument can be provided on service creation. It will be passed to the route declaration. This way you can combine URL Dispatch and traversal to build an hybrid application.

## 2.11 Frequently Asked Questions (FAQ)

Here is a list of frequently asked questions related to Cornice.

### 2.11.1 Cornice registers exception handlers, how do I deal with it?

Cornice registers its own exception handlers so it's able to behave the right way in some edge cases (it's mostly done for CORS support).

Sometimes, you will need to register your own exception handlers, and Cornice might get on your way.

You can disable the exception handling by using the *handle_exceptions* setting in your configuration file or in your main app:

```
config.add_settings(handle_exceptions=False)
```

## 2.12 Upgrading

### 2.12.1 1.X to 2.X

#### Project template

We now rely on Cookiecutter instead of the deprecated Pyramid scaffolding feature:

```
$ cookiecutter gh:Cornices/cookiecutter-cornice
```

#### Sphinx documentation

The Sphinx extension now lives in a separate package, that must be installed:

```
pip install cornice_sphinx
```

Before in your `docs/conf.py`:

Now:

#### Validators

Validators now receive the kwargs of the related service definition.

Before:

```
def has_payed(request):
    if 'paid' not in request.GET:
        request.errors.add('body', 'paid', 'You must pay!')
```

Now:

```
def has_payed(request, **kwargs):
    free_access = kwargs.get('free_access')
    if not free_access and 'paid' not in request.GET:
        request.errors.add('body', 'paid', 'You must pay!')
```

### Colander validation

Colander schema validation now requires an explicit validator on the service view definition.

Before:

```
class SignupSchema(colander.MappingSchema):
    username = colander.SchemaNode(colander.String())

@signup.post(schema=SignupSchema)
def signup_post(request):
    username = request.validated['username']
    return {'success': True}
```

Now:

```
from cornice.validators import colander_body_validator

class SignupSchema(colander.MappingSchema):
    username = colander.SchemaNode(colander.String())

@signup.post(schema=SignupSchema(), validators=(colander_body_validator,))
def signup_postt(request):
    username = request.validated['username']
    return {'success': True}
```

This makes declarations a bit more verbose, but decorrelates Cornice from Colander. Now any validation library can be used.

---

**Important:** Some of the validation messages may have changed from version 1.2. For example `Invalid escape sequence` becomes `Invalid \\uXXXX escape`.

---

### Complex Colander validation

If you have complex use-cases where data has to be validated accross several locations of the request (like querystring, body etc.), Cornice provides a validator that takes an additionnal level of mapping for `body`, `querystring`, `path` or `headers` instead of the former `location` attribute on schema fields.

The `request.validated` hences reflects this additional level.

Before:

```
class SignupSchema(colander.MappingSchema):
    username = colander.SchemaNode(colander.String(), location='body')
    referrer = colander.SchemaNode(colander.String(), location='querystring',
                                   missing=colander.drop)

@signup.post(schema=SignupSchema)
def signup_post(request):
    username = request.validated['username']
    referrer = request.validated['referrer']
    return {'success': True}
```

Now:

```
from cornice.validators import colander_validator
```

```
class Querystring(colander.MappingSchema):
    referrer = colander.SchemaNode(colander.String(), missing=colander.drop)

class Payload(colander.MappingSchema):
    username = colander.SchemaNode(colander.String())

class SignupSchema(colander.MappingSchema):
    body = Payload()
    querystring = Querystring()

signup = cornice.Service()

@signup.post(schema=SignupSchema(), validators=(colander_validator,))
def signup_post(request):
    username = request.validated['body']['username']
    referrer = request.validated['querystring']['referrer']
    return {'success': True}
```

This now allows to have validation at the schema level that validates data from several locations:

```
class SignupSchema(colander.MappingSchema):
    body = Payload()
    querystring = Querystring()

    def deserialize(self, cstruct=colander.null):
        appstruct = super(SignupSchema, self).deserialize(cstruct)
        username = appstruct['body']['username']
        referrer = appstruct['querystring'].get('referrer')
        if username == referred:
            self.raise_invalid('Referrer cannot be the same as username')
        return appstruct
```

### Deferred validators

Colander deferred validators allow to access runtime objects during validation, like the current request for example.

Before, the binding to the request was implicitly done by Cornice, and now has to be explicit.

```
import colander

@colander.deferred
def deferred_validator(node, kw):
    request = kw['request']
    if request['x-foo'] == 'version_a':
        return colander.OneOf(['a', 'b'])
    else:
        return colander.OneOf(['c', 'd'])

class Schema(colander.MappingSchema):
    bazinga = colander.SchemaNode(colander.String(), validator=deferred_validator)
```

Before:

```
signup = cornice.Service()

@signup.post(schema=Schema())
def signup_post(request):
    return {}
```

After:

```
def bound_schema_validator(request, **kwargs):
    schema  = kwargs['schema']
    kwargs['schema'] = schema.bind(request=request)
    return colander_validator(request, **kwargs)

signup = cornice.Service()

@signup.post(schema=Schema(), validators=(bound_schema_validator,))
def signup_post(request):
    return {}
```

### Error handler

- The `error_handler` callback of services now receives a `request` object instead of `errors`.

Before:

```
def xml_error(errors):
    request = errors.request
    ...
```

Now:

```
def xml_error(request):
    errors = request.errors
    ...
```

### Deserializers

The support of `config.add_deserializer()` and `config.registry.cornice_deserializers` was dropped.

Deserializers are still defined via the same API:

```
def dummy_deserializer(request):
    if request.headers.get("Content-Type") == "text/dummy":
        values = request.body.decode().split(',')
        return dict(zip(['foo', 'bar', 'yeah'], values))
    request.errors.add(location='body', description='Unsupported content')

@myservice.post(schema=FooBarSchema(),
                deserializer=dummy_deserializer,
                validators=(my_validator,))
```

But now, instead of using the application registry, the `deserializer` is accessed via the validator kwargs:

```
from cornice.validators import extract_cstruct

def my_validator(request, deserializer=None, **kwargs):
    if deserializer is None:
        deserializer = extract_cstruct
    data = deserializer(request)
    ...
```

**Note:** The built-in `colander_validator` supports custom deserializers and defaults to the built-in JSON deserializer `cornice.validators.extract_cstruct`.

**Note:** The attributes `registry.cornice_deserializers` and `request.deserializer` are not set anymore.

### Services schemas introspection

The `schema` argument of services is now treated as service kwarg. The `service.schemas_for()` method was dropped as well as the `service.schemas` property.

Before:

```
schema = service.schemas_for(method="POST")
```

Now:

```
schema = [kwargs['schema'] for method, view, kwargs in service.definitions
          if method == "POST"][0]
```

CHAPTER 3

# Contribution & Feedback

Cornice is a project initiated at Mozilla Services, where we build Web Services for features like Firefox Sync. All of what we do is built with open source, and this is one brick of our stack.

We welcome Contributors and Feedback!

- Developers Mailing List: https://mail.mozilla.org/listinfo/services-dev
- Repository: https://github.com/mozilla-services/cornice

## C

## A



## C



## D



## E



## R



## S



## V