# CorePost Documentation

***Release 0.0.16***

**Jacek Furmankiewicz**

April 20, 2012

# CONTENTS

CHAPTER

ONE

# A TWISTED REST MICRO-FRAMEWORK

## 1.1 Introduction

### 1.1.1 What is CorePost?

CorePost is a Python REST micro-framework. It is meant for building enterprise-grade REST server applications that provide API services to other applications and/or a UI layer (coded in any framework or language).

More importantly, CorePost is an asynchronous I/O web framework (similar to Node.js). Hence it relies on asynchronous I/O operations, which are extremely efficient, but somewhat more complicated to code.

Fortunately, CorePost does not create it's own async I/O library, but instead uses under the mature, well documented and extremely well designed Twisted library, in particular its web layer (known simply as twisted.web)

Coupled with a JIT runtime like PyPy, this should give you the ability to develop REST server side applications that will be extremely performant in production, yet (hopefully) fun and productive to develop.

### 1.1.2 What is Twisted?

Twisted is a very mature Python async I/O network toolkit:

http://twistedmatrix.com/trac/

Understanding core principles behind Twisted and its APIs is required (at least at a basic level) before coding any CorePost application.

Hence we recommend either reading the very thorough developer's guide:

http://twistedmatrix.com/documents/current/core/howto/book.pdf

or the excellent Twisted tutorials from Dave Peticolas:

http://krondo.com/blog/?page_id=1327

In particular, understanding the core Twisted Deferred object (and its productive inline callback approach) are crucial to productive usage of Twisted APIs for writing asynchronous web applications.

### 1.1.3 What does CorePost add on top of Twisted Web?

Mostly productivity features that take of low-level plumbing such as:

- routing request to handler methods
- automatic parsing of JSON/YAML/XML input
- automatic conversion of Python objects and classes to JSON / YAML / XML formats
- simplified exception handling
- custom request / response filters

However, this is a very thin layer. Once you get to write some serious code that interacts with an external system (e.g. a SQL database) you are writing a hard-code Twisted web application. CorePost is just there to make it easier for you and let you focus on business logic, while letting it take care of common required plumbing. That's it.

A CorePost application is nothing more than a *twisted.web* application under the hood.

### 1.1.4 Why would I use CorePost instead of Node.js?

As you develop more Twisted code, you will realize how its elegant and powerful *Deferred* object (and especially inline callbacks) make developing *readable* asynchronous code much more pleasant than any other solution.

CHAPTER

TWO

# FEATURES

## 2.1 URL Routing

### 2.1.1 @route decorator

Via a simple *@route* decorator you can automatically route *twisted.web* Request objects to your class method based on URL (with dynamic paths), HTTP method, expected content type, etc:

```
from corepost.web import route, RESTResource
from corepost.enums import Http

class RESTService():

    @route("/",Http.GET)
    def root(self,request,**kwargs):
        return request.path

    @route("/test",Http.GET)
    def test(self,request,**kwargs):
        return request.path

    @route("/test/<int:numericid>",Http.GET)
    def test_get_resources(self,request,numericid,**kwargs):
        return "%s" % numericid

if __name__ == '__main__':
    app = RESTResource((RESTService,))
    app.run()
```

*Note*:

> This piece of code:
>
> ```
> app.run()
> ```
>
> is just for convenience when showing code samples and writing unit tests. In a real production application you would use existing Twisted *twistd* functionality:
>
> - http://twistedmatrix.com/documents/current/core/howto/basics.html
> - http://twistedmatrix.com/documents/current/core/howto/application.html
> - http://twistedmatrix.com/documents/current/core/howto/tap.html

### 2.1.2 Path argument extraction

CorePort can easily extract path arguments from an URL and convert them to the desired type.

The supported types are:

- *int*
- *float*
- *string*

Example:

```
@route("/int/<int:intarg>/float/<float:floatarg>/string/<stringarg>",Http.GET)
def test(self,request,intarg,floatarg,stringarg,**kwargs):
        pass
```

### 2.1.3 Routing requests by incoming content type

Based on the incoming content type in POST/PUT requests, the *same* URL can be hooked up to different router methods:

```
@route("/post/by/content",(Http.POST,Http.PUT),MediaType.APPLICATION_JSON)
def test_content_app_json(self,request,**kwargs):
    return request.received_headers[HttpHeader.CONTENT_TYPE]

@route("/post/by/content",(Http.POST,Http.PUT),(MediaType.TEXT_XML,MediaType.APPLICATION_XML))
def test_content_xml(self,request,**kwargs):
    return request.received_headers[HttpHeader.CONTENT_TYPE]

@route("/post/by/content",(Http.POST,Http.PUT),MediaType.TEXT_YAML)
def test_content_yaml(self,request,**kwargs):
    return request.received_headers[HttpHeader.CONTENT_TYPE]

@route("/post/by/content",(Http.POST,Http.PUT))
def test_content_catch_all(self,request,**kwargs):
    return MediaType.WILDCARD
```

## 2.2 Argument parsing

CorePost can automatically parse query arguments, form arguments, as well as basic JSON, YAML and XML documents and extract those as direct arguments to a REST router method.

Let's say we have a basic method that responds to GET, POST and PUT requests. It expects a first name and last name and outputs them back in the response:

```
@router("/name",(Http.GET,Http.POST,Http.PUT))
def getName(self,request,first,last,**kwargs):
        return "%s %s" % (first, last)
```

### 2.2.1 Query arguments

For GET requests, the query arguments will be automatically parsed, e.g.:

```
curl http://127.0.0.1/name?first=John&last=Doe
```

### 2.2.2 Form encoded arguments

For POST/PUT requests, any form-encoded arguments will be automatically parsed, e.g.:

```
curl -X POST http://localhost/name -d "first=John&last=Doe"
```

### 2.2.3 JSON document arguments

For the same method, you could just post a JSON document instead that looks like this:

```
{"first":"John","last":"Doe"}
```

CorePost will automatically pass all the root elements of the document as arguments into a method. Requires the *'application/json'* content type to be passed.

### 2.2.4 YAML document arguments

For the same method, you could just post a YAML document that looks like this:

```
first:John
last:Doe
```

CorePost will automatically pass all the root elements of the document as arguments into a method. Requires the *'text/yaml'* content type to be passed.

### 2.2.5 XML document arguments

XML documents are supported as well. In that case, CorePost will first parse all the attributes on the root node and then all of the children underneath the main root node.

Hence all of the XML formats below are valid and would generate the same parameters to a method.

Attributes only:

```
<root first="John" last="Doe"/>
```

Mix of attributes and child nodes:

```
<root first="John">
        <last>Doe</last>
</root>
```

Child nodes only:

```
<root>
        <first>John</first>
        <last>Doe</last>
</root>
```

Requires the *'text/xml'* OR *'application/xml'* content type to be passed.

As you can see from the examples above, a single CorePost router method can handle all these varied forms of argument parsing for you without any additional effort.

## 2.3 Argument validation

CorePost integrates the popular 'formencode' package to implement form and query argument validation. Validators can be specified using a *formencode* Schema object, or via custom field-specific validators.

Example:

```
from corepost.web import validate, route
from corepost.enums import Http
from formencode import Schema, validators

class TestSchema(Schema):
    allow_extra_fields = True
    childId = validators.Regex(regex="^value1|value2$")

class MyApp():

        @route("/validate/<int:rootId>/schema",Http.POST)
        @validate(schema=TestSchema())
        def postValidateSchema(self,request,rootId,childId,**kwargs):
            '''Validate using a common schema'''
            return "%s - %s - %s" % (rootId,childId,kwargs)

        @route("/validate/<int:rootId>/custom",Http.POST)
        @validate(childId=validators.Regex(regex="^value1|value2$"))
        def postValidateCustom(self,request,rootId,childId,**kwargs):
            '''Validate using argument-specific validators'''
            return "%s - %s - %s" % (rootId,childId,kwargs)
```

Please see the *FormEncode* documentation:

http://www.formencode.org/en/latest/Validator.html

for list of available validators:

- Common : http://www.formencode.org/en/latest/modules/validators.html#module-formencode.validators
- National : http://www.formencode.org/en/latest/modules/national.html#module-formencode.national

## 2.4 HTTP codes

By default, CorePost returns the appropriate HTTP code based on the HTTP method:

Success:

- 200 (OK) - GET, DELETE, PUT
- 201 (Created) - POST

Errors:

- 404 - not able to match any URL.
- 400 - missing mandatory argument (driven from the arguments on the actual functions)
- 400 - argument failed validation
- 500 - server error

## 2.5 Returning responses

There are a number of ways in which you can return a response from a REST service

### 2.5.1 String

You can simply return a plain text String. CorePost will return the appropriate HTTP code for you:

```
@route("/",Http.GET)
def root(self,request,**kwargs):
    return "Hello"
```

### 2.5.2 Dictionaries, lists or classes

You can return straight dictionaries:

```
@route("/",Http.GET)
def root(self,request,**kwargs):
    return {"test":"test"}
```

or lists:

```
@route("/",Http.GET)
def root(self,request,**kwargs):
    return [{"test":"test"},{"test":"test2"}]
```

or classes:

```
@route("/",Http.GET)
def root(self,request,**kwargs):
    return SomeClass()
```

CorePost will serialize each of them to the appropriate content type (JSON,YAML or XML), depending on what the caller can accept.

### 2.5.3 Response objects

This option gives you the most control, as you can explicitly specify the response content, headers and HTTP code. You need to return an instance of *corepost.Response* object:

```
class Response:
    """
    Custom response object, can be returned instead of raw string response
    """
    def __init__(self,code=200,entity=None,headers={}):
        pass
```

Example:

```
@route("/",Http.POST)
def post(self,request,customerId,addressId,streetNumber,streetName,stateCode,countryCode):
    c = DB.getCustomer(customerId)
    address = CustomerAddress(streetNumber,streetName,stateCode,countryCode)
    c.addresses[addressId] = address
    return Response(201)
```

## 2.6 Content types

CorePost integrates support for JSON, YAML and XML (partially) based on request content types.

### 2.6.1 Parsing of incoming content

Based on the incoming content type in POST/PUT requests, the body will be automatically parsed to JSON, YAML and XML (ElementTree)

- request.json
- request.yaml
- request.xml

and attached to the request:

```
@route("/post/json",(Http.POST,Http.PUT))
def test_json(self,request,**kwargs):
    return "%s" % json.dumps(request.json)

@route("/post/xml",(Http.POST,Http.PUT))
def test_xml(self,request,**kwargs):
    return "%s" % ElementTree.tostring(request.xml)

@route("/post/yaml",(Http.POST,Http.PUT))
def test_yaml(self,request,**kwargs):
    return "%s" % yaml.dump(request.yaml)
```

### 2.6.2 Converting Python objects to expected content type

Instead of returning string responses, the code can just return Python objects. Depending whether the caller can accept JSON (default) or YAML, the Python objects will be automatically converted:

```
@route("/return/by/accept")
def test_return_content_by_accepts(self,request,**kwargs):
    val = [{"test1":"Test1"},{"test2":"Test2"}]
    return val
```

Calling this URL with "Accept: application/json" will return:

```
[{"test1": "Test1"}, {"test2": "Test2"}]
```

Calling it with "Accept: text/yaml" will return:

```
- {test1: Test1}
- {test2: Test2}
```

## 2.7 Modular REST applications

A typical case in REST is where you have parent/child resources (business entities), e.g.

```
Customer

        Customer Address
```

```
        Customer Phone

        Customer Order

                Customer Invoice

                        Customer Invoice Payment
```

etc.

This can create a URL structure like:

```
/customer

/customer/<customerId>

/customer/<customerId>/address

/customer/<customerId>/address/<addressId>

/customer/<customerId>/phone

/customer/<customerId>/phone/<phoneId>

/customer/<customerId>/invoice

/customer/<customerId>/invoice/<invoiceId>

/customer/<customerId>/invoice/<invoiceId>/payment

/customer/<customerId>/invoice/<invoiceId>/payment/<paymentId>
```

CorePost allows you to write small, modular classes that implement a REST service for just a single entity, driven by URL paths with dynamic elements in them (e.g. the *customerId*, *invoiceId*, *paymentId* path parameters in the sample above). You do not have to mesh all these different entities in a single class.

At the end, you wrap all of the different REST services in a single *RESTResource* object (which extends the regular Twisted Web Resource object) and it takes care of routing the request to the appropriate class.

Here is a full-blown example of two REST services for Customer and Customer Address:

```
from corepost import Response, NotFoundException, AlreadyExistsException
from corepost.web import RESTResource, route, Http

class CustomerRESTService():
    path = "/customer"

    @route("/")
    def getAll(self,request):
        return DB.getAllCustomers()

    @route("/<customerId>")
    def get(self,request,customerId):
        return DB.getCustomer(customerId)

    @route("/",Http.POST)
    def post(self,request,customerId,firstName,lastName):
        customer = Customer(customerId, firstName, lastName)
        DB.saveCustomer(customer)
```

```
        return Response(201)

    @route("/<customerId>",Http.PUT)
    def put(self,request,customerId,firstName,lastName):
        c = DB.getCustomer(customerId)
        (c.firstName,c.lastName) = (firstName,lastName)
        return Response(200)

    @route("/<customerId>",Http.DELETE)
    def delete(self,request,customerId):
        DB.deleteCustomer(customerId)
        return Response(200)

    @route("/",Http.DELETE)
    def deleteAll(self,request):
        DB.deleteAllCustomers()
        return Response(200)

class CustomerAddressRESTService():
    path = "/customer/<customerId>/address"

    @route("/")
    def getAll(self,request,customerId):
        return DB.getCustomer(customerId).addresses

    @route("/<addressId>")
    def get(self,request,customerId,addressId):
        return DB.getCustomerAddress(customerId, addressId)

    @route("/",Http.POST)
    def post(self,request,customerId,addressId,streetNumber,streetName,stateCode,countryCode):
        c = DB.getCustomer(customerId)
        address = CustomerAddress(streetNumber,streetName,stateCode,countryCode)
        c.addresses[addressId] = address
        return Response(201)

    @route("/<addressId>",Http.PUT)
    def put(self,request,customerId,addressId,streetNumber,streetName,stateCode,countryCode):
        address = DB.getCustomerAddress(customerId, addressId)
        (address.streetNumber,address.streetName,address.stateCode,address.countryCode) = (streetNumb
        return Response(200)

    @route("/<addressId>",Http.DELETE)
    def delete(self,request,customerId,addressId):
        DB.getCustomerAddress(customerId, addressId) #validate address exists
        del(DB.getCustomer(customerId).addresses[addressId])
        return Response(200)

    @route("/",Http.DELETE)
    def deleteAll(self,request,customerId):
        c = DB.getCustomer(customerId)
        c.addresses = {}
        return Response(200)


def run_rest_app():
    app = RESTResource((CustomerRESTService(),CustomerAddressRESTService()))
    app.run(8080)
```

```
if __name__ == "__main__":
    run_rest_app()
```

## 2.8 Filters

There is support for CorePost resource filters via the two following *corepost.filter* interfaces:

```
class IRequestFilter(Interface):
    """Request filter interface"""
    def filterRequest(self,request):
        """Allows to intercept and change an incoming request"""
        pass

class IResponseFilter(Interface):
    """Response filter interface"""
    def filterResponse(self,request,response):
        """Allows to intercept and change an outgoing response"""
        pass
```

A filter class can implement either of them or both (for a wrap around filter), e.g.:

```
class AddCustomHeaderFilter():
    """Implements a request filter that adds a custom header to the incoming request"""
    zope.interface.implements(IRequestFilter)

    def filterRequest(self,request):
        request.received_headers["Custom-Header"] = "Custom Header Value"

class Change404to503Filter():
    """Implements just a response filter that changes 404 to 503 statuses"""
    zope.interface.implements(IResponseFilter)

    def filterResponse(self,request,response):
        if response.code == 404:
            response.code = 503

class WrapAroundFilter():
    """Implements both types of filters in one class"""
    zope.interface.implements(IRequestFilter,IResponseFilter)

    def filterRequest(self,request):
        request.received_headers["X-Wrap-Input"] = "Input"

    def filterResponse(self,request,response):
        response.headers["X-Wrap-Output"] = "Output"
```

In order to activate the filters on a RESTResource instance, you need to pass a list of them in the constructor as the *filters* parameter, e.g.:

```
class FilterApp:

    @route("/",Http.GET)
    def root(self,request,**kwargs):
        return request.received_headers

def run_filter_app():
```

```
    app = RESTResource(services=(FilterApp(),),filters=(Change404to503Filter(),AddCustomHeaderFilter
    app.run(8083)
```

## 2.9 Asynchronous Operations

### 2.9.1 @defer.inlineCallbacks support

If you want a deferred async method, just use *defer.returnValue()*:

```
@route("/",Http.GET)
@defer.inlineCallbacks
def root(self,request,**kwargs):
        val1 = yield db.query("SELECT ....")
        val2 = yield db.query("SELECT ....")
        defer.returnValue(val1 + val2)
```

This is standard Twisted functionality.