RESTArt Documentation

Release 0.1.3

RussellLuo

February 21, 2016

Contents

Welcome to RESTArt's documentation. I recommend that you get started with *Installation* and then head over to the *Quickstart*.

CHAPTER 1

User's Guide

This section covers most things you need to know to build REST APIs with RESTArt.

1.1 Installation

1.1.1 Release Version

Install RESTArt with *pip*:

```
$ pip install Python-RESTArt
```

1.1.2 Development Version

Install development version from *GitHub*:

```
$ git clone https://github.com/RussellLuo/restart.git
$ cd restart
$ python setup.py install
```

1.2 Quickstart

Eager to get started? This page gives you a good introduction to RESTArt.

It assumes you already have RESTArt installed. If you do not, head over to the *Installation* section.

1.2.1 A Minimal API

A minimal RESTArt API looks something like this:

```
from restart.api import RESTArt
from restart.resource import Resource

api = RESTArt()

@api.route(methods=['GET'])
class Greeting(Resource):
    name = 'greeting'
```

```
    def read(self, request):
        return {'hello': 'world'}
```

Just save it as `helloworld.py` and run it with **`restart`** command:

```
$ restart helloworld:api
```

Then you can consume the API now:

```
$ curl http://127.0.0.1:5000/greeting
{"hello": "world"}
```

So what does the above code do?

1. First we import two classes *`RESTArt`* and *`Resource`* for later use.
2. Next we create an instance of the *`RESTArt`* class, which represents the whole RESTArt API.
3. We then use the *`route()`* decorator to register the *Greeting* class which only cares HTTP verb *GET*.
4. The *Greeting* class is defined as a resource by subclassing the *`Resource`* class. It has a `read()` method which is a handler for HTTP verb *GET*.

1.2.2 Resources

In the world of REST APIs, resource is the first-class citizen. That is to say, when you are implementing a REST API, resources are your building blocks.

There are two types of resources: plural resources and singular resources.

Plural Resources

Most resources are conceptually equivalent to a collection. These resources are called **Plural Resources**.

As a commonly-accepted practice, you should always use plurals in URIs for pluaral resources. Let's take the classical *Todo* application as an example. If we implement Todo as a resource, there will be two basic URIs for it:

```
/todos
/todos/123
```

And the frequently-used HTTP verbs (or methods) are:

```
GET /todos
POST /todos
GET /todos/123
PUT /todos/123
PATCH /todos/123
DELETE /todos/123
```

Singular Resources

Sometimes, there are resources that have no collection concept, then we can treat them as **Singular Resources**.

The *Greeting* resource is just an example of singular resources. There are only one URI for it:

```
/greeting
```

Although we only care HTTP verb *GET* then, the possible and frequently-used HTTP verbs are as follows:

```
GET /greeting
PUT /greeting
PATCH /greeting
DELETE /greeting
```

Note the lack of a greeting ID and usage of POST verb.

1.2.3 Routing

With the above concepts and conventions in mind, RESTArt provides three methods to route a resource: *register()*, *route()* and *add_rule()*.

register()

The *register()* decorator is provided as a convenient helper specially for plural resources.

Take the *Todo* resource as an example, we may define and register it with the *register()* decorator like this:

```
@api.register
class Todo(Resource):
    name = 'todos'

    # define methods here
```

See here for the full code of the *Todo* resource.

Now six different routes are created:

HTTP Verb	Path	Resource:Action	Used for
GET	/todos	Todo:index()	display a list of all todos
POST	/todos	Todo:create()	create a new todo
GET	/todos/<pk>	Todo:read()	display a specific todo
PUT	/todos/<pk>	Todo:replace()	replace a specific todo
PATCH	/todos/<pk>	Todo:update()	update a specific todo
DELETE	/todos/<pk>	Todo:delete()	delete a specific todo

Note: You can also register a plural resource by using *route()* instead of *register()*, although it is more complicated.

For example, the following registration is equivalent to the above one:

```
@api.route(uri='/todos', endpoint='todos_list',
           methods=['GET', 'POST'], actions={'GET': 'index'})
@api.route(uri='/todos/<pk>', endpoint='todos_item',
           methods=['GET', 'PUT', 'PATCH', 'DELETE'])
class Todo(Resource):
    name = 'todos'

    # define methods here
```

---

route()

The *route()* decorator is provided mainly for singular resources, but you can also use it for plural resources to customize more details.

For example, if we want to provide a global and single configuration object, we can create it as a singular resource like this:

```
@api.route(methods=['GET', 'PUT', 'PATCH', 'DELETE'])
class Configuration(Resource):
    name = 'configuration'

    # define methods here
```

Now four different routes are created:

HTTP Verb	Path	Resource:Action	Used for
GET	/configuration	Configuration:read()	display the configuration
PUT	/configuration	Configuration:replace()	replace the configuration
PATCH	/configuration	Configuration:update()	update the configuration
DELETE	/configuration	Configuration:delete()	delete the configuration

add_rule()

The *add_rule()* method is the fundamental method both for *register()* and *route()*. If you do not like the decorator style, and you want to customize more behaviors, you should use it.

1.3 Serving

This section shows you how to run your APIs in RESTArt.

As a concrete example, consider the following `Greeting` API:

```
# helloworld.py

from restart.api import RESTArt
from restart.resource import Resource

api = RESTArt()

@api.route(methods=['GET'])
class Greeting(Resource):
    name = 'greeting'

    def read(self, request):
        return {'hello': 'world'}
```

1.3.1 Development

The Pythonic Way

As a Pythonista, chances are you like to run the API just as a normal Python script. That's good!

The Pythonic way you want is supported by RESTArt, at the cost of a little wrapper code with the help of the *Service* class:

```
# runserver.py

from restart.serving import Service
from helloworld import api
```

```
service = Service(api)

if __name__ == '__main__':
    service.run()
```

Now, you can run the API like this:

```
$ python runserver.py
```

The Command Line Utility

To make the serving step as simple as possible, RESTArt also provides a command line utility called `restart`. You may have seen it in *Quickstart*. Yes! It's born for serving, and you will not be disappointed to use it:

```
$ restart helloworld:api
```

That's all. Isn't it amazing?

`restart` has only one argument:

Argument	Example	Description
entry-point	helloworld:api	A string in the form `module_path:api` where *api* is the central RESTArt API object and *module_path* is the path to the module where *api* is defined.

For the options supported by `reatart`, see the help messages:

```
$ restart --help
```

1.3.2 Deployment

RESTArt's primary deployment platform is WSGI, the Python standard for web servers and applications.

To make RESTArt APIs easy to deploy, it's recommended to create a file named *wsgi.py* as follows:

```
# wsgi.py

from restart.serving import Service
from helloworld import api

application = Service(api)
```

Then use awesome WSGI servers to communicate with the *application* callable.

Gunicorn

Gunicorn ('Green Unicorn') is a pure-Python WSGI server for UNIX. It has no dependencies and is easy to install and use.

1. Install Gunicorn:

```
    $ pip install gunicorn
```

2. Use Gunicorn:

```
$ gunicorn wsgi -b 127.0.0.1:5000
```

1.4 Configuration

This section covers all configuration options for you to customize the behavior of RESTArt APIs.

1.4.1 Options

Server

Option name	Default value	Description
SERVER_NAME	''	The server name (scheme + domain + port)

Action mapping

Option name	Default value	Description
ACTION_MAP	{'HEAD': 'head', 'TRACE': 'trace', 'GET': 'read', 'PUT': 'replace', 'POST': 'create', 'DELETE': 'delete', 'OPTIONS': 'options', 'PATCH': 'update'}	The mapping from request methods to resource actions, which is used to find the specified action to handle the request.

Parsers and Renderers

Option name	Default value	Description
PARSER_CLASSES	('restart.parsers.JSONParser', 'restart.parsers.URLEncodedParser', 'restart.parsers.MultiPartParser')	The default Parser classes.
RENDERER_CLASSES	('restart.renderers.JSONRenderer',)	The default Renderer classes.

Logger

Option name	Default value	Description
LOGGER_ENABLED	True	Enable or disable the global logger.
LOGGER_METHODS	('GET', 'POST', 'PUT', 'PATCH', 'DELETE')	A sequence of HTTP methods whose messages should be logged.
LOGGER_LEVEL	'INFO'	The logging level.
LOGGER_FORMAT	'%(asctime)s.%(msecs)03d %(name)-10s %(levelname)-8s %(message)s'	The logging format for strings.
LOGGER_DATE_FORMAT	'%Y-%m-%d %H:%M:%S'	The logging format for date/time.

Middlewares

Option name	Default value	Description
MIDDLEWARE_CLASSES	()	The middleware classes used to alter RESTArt's requests and responses.

1.4.2 Customization

You can customize all of the above configuraion options by following the steps below:

1. Create a Python module to set your preferred values:

```
$ vi restart_config.py

LOGGER_METHODS = ['POST', 'PUT', 'PATCH']
LOGGER_LEVEL = 'DEBUG'
```

2. Set the environment variable `RESTART_CONFIG_MODULE` to the Python path of the above module:

```
$ export RESTART_CONFIG_MODULE=pythonpath.to.restart_config
```

That's all. Then, while your API is running, messages with DEBUG (or higher) level will be logged for any request whose HTTP method is *POST*, *PUT* or *PATCH*.

1.5 Middleware

Middleware is a framework of hooks into RESTArt's request/response processing. It's a light, low-level "plugin" system for globally altering RESTArt's input or output.

In RESTArt, any Python class that has a `process_request()` method or a `process_response()` method can be used as a middleware. See *`perform_action`* for more information about middleware behaviors.

1.5.1 Write a middleware class

Suppose you already have an API, and now you want to only allow the authenticated users to access it. To add this limit, you can write a simple middleware class (based on HTTP Basic authentication) like this:

```
# my_middlewares.py

from restart.exceptions import Unauthorized

class AuthMiddleware(object):
    """The middleware used for authentication."""

    def process_request(self, request):
        """Authenticate the request.

        :param request: the request object.
        """
        username = request.auth.get('username')
        password = request.auth.get('password')
        if not (username == 'YOUR_USERNAME' and password == 'YOUR_PASSWORD'):
            raise Unauthorized()
```

For a real-world middleware implementation, see RESTArt-CrossDomain for an example.

1.5.2 Use a middleware class

RESTArt supports middlewares in two styles:

- Global middlewares
- Resource-level middlewares

The processing order of the two styles of middlewares is as follows:

- During request phase, the `process_request()` methods of global middlewares are called before those of resource-level middlewares.
- During response phase, the `process_response()` methods of resource-level middlewares are called before those of global middlewares.

Global middlewares

To use a middleware class as a global middleware, just add it to the *MIDDLEWARE_CLASSES* tuple in your RESTArt configuration module.

In the *MIDDLEWARE_CLASSES* tuple, each middleware is represented by a string: the full Python path to the middleware's class name. For example, here's how to enable the above `AuthMiddleware` middleware class:

```
MIDDLEWARE_CLASSES = (
    'my_middlewares.AuthMiddleware',
)
```

Resource-level middlewares

To use a middleware class as a resource-level middleware, just add it to the `middleware_classes` tuple as the class attribute of your resource class.

In the `middleware_classess` tuple, each middleware is represented by a class. For example, here's how to enable the above `AuthMiddleware` middleware class:

```
from restart.api import RESTArt
from restart.resource import Resource

from my_middlewares import AuthMiddleware

api = RESTArt()

@api.route(methods=['GET'])
class Demo(Resource):
    name = 'demo'

    middleware_classes = (AuthMiddleware,)

    def read(self, request):
        return 'this is a demo'
```

1.6 Testing

RESTArt provides a small set of tools that come in handy when writing tests.

1.6.1 The test client

The test client is a Python class that acts as a dummy Web client, allowing you to test your resources and interact with your RESTArt-powered APIs programmatically.

class `restart.testing.`**`Client`**(*api*)
The class used as a test client.

Example:

```
client = Client(api)
# GET /examples
response = client.get('/examples')
# POST /examples
response = client.post('/examples', data='{"name": "test"}',
                       content_type='application/json')
```

Parameters **`api`** – the RESTArt API object.

1.6.2 The request factory

The *`RequestFactory`* provides a way to generate a request instance that can be used as the first argument to any resource. This means you can test a resource very easy.

class `restart.testing.`**`RequestFactory`**(*keep_initial_request=False*)
The class used to generate request objects.

Example:

```
factory = RequestFactory()
# GET /examples
request = factory.get('/examples')
# POST /examples
request = factory.post('/examples', data='{"name": "test"}',
                       content_type='application/json')
```

Parameters **`keep_initial_request`** – a boolean value. If set to *True*, the request object generated by the factory will be the initial request object, which is framework-specific. If not specified, defaults to *False*, then the final adapted request object will be generated.

1.7 Framework Integration

RESTArt looks like a micro-framework. Like many frameworks, RESTArt handles requests and responses in its own way, and you can build REST APIs based on RESTArt without the help of any other framework.

Actually, RESTArt is designed as a library, which is framework-agnostic. It's the underlying library Werkzeug that gives RESTArt the ability to serve APIs independently. Strictly speaking, RESTArt consists of the framework-agnostic core library and the framework-specific Werkzeug integration.

1.7.1 Why to integrate with other frameworks?

With the built-in Werkzeug integration, RESTArt works well for serving standalone APIs. You may ask why we need to integrate RESTArt with other frameworks? The following are the reasons I can think of:

- You are working with an existing or legacy application, which uses a specific framework
- Your API must be based on a useful library or an awesome extension, but it is framework-specific
- The integration with a specific framework can improve the performance of your API (e.g. RESTArt-Falcon)

1.7.2 How to integrate with other frameworks?

In RESTArt, framework integration is made easy by using adapters. For a real-world example, see the source code of *WerkzeugAdapter*, which is the adapter for the built-in Werkzeug integration. As another example, We can write an adapter for integrating RESTArt into Flask. Since both RESTArt and Flask are based on Werkzeug, it's an easy job:

```
from six import iteritems
from restart.adapter import Adapter, WerkzeugAdapter
from flask import Flask, request


class FlaskAdapter(Adapter):

    def __init__(self, *args, **kwargs):
        super(FlaskAdapter, self).__init__(*args, **kwargs)
        self.werkzeug_adapter = WerkzeugAdapter(*args, **kwargs)
        self.app = Flask(__name__)
        # Add Flask-specific URI routes
        for rule in self.get_embedded_rules():
            self.app.add_url_rule(**rule)

    def adapt_handler(self, handler, *args, **kwargs):
        """Adapt the request object and the response object for
        the `handler` function.

        :param handler: the handler function to be adapted.
        :param args: a list of positional arguments that will be passed
                     to the handler.
        :param kwargs: a dictionary of keyword arguments that will be passed
                       to the handler.
        """
        return self.werkzeug_adapter.adapt_handler(handler, request,
                                                   *args, **kwargs)

    def wsgi_app(self, environ, start_response):
        """The actual Flask-specific WSGI application.

        See :meth:`~restart.serving.Service.wsgi_app` for the
        meanings of the parameters.
        """
        return self.app(environ, start_response)

    def get_embedded_rules(self):
        """Get the Flask-specific rules used to be embedded into
        an existing or legacy application.

        Usage:
```

```
            # The existing Flask application
            from flask import Flask
            app = Flask()
            ...

            # The RESTArt API
            from restart.api import RESTArt
            api = RESTArt()
            ...

            # Embed RESTArt into Flask
            from restart.serving import Service
            from restart.ext.flask.adapter import FlaskAdapter
            service = Service(api, FlaskAdapter)
            for rule in service.embedded_rules:
                app.add_url_rule(**rule)
        """
        rules = [
            dict(rule=rule.uri, endpoint=endpoint,
                 view_func=rule.handler, methods=rule.methods)
            for endpoint, rule in iteritems(self.adapted_rules)
        ]
        return rules
```

1.7.3 Framework Adapters

As a summary, the following list gives the adapters for some frameworks:

Framework	Adapter	Support Type
Werkzeug	WerkzeugAdapter	Built-in class
Flask	*FlaskAdapter*	Extension class
Falcon	RESTArt-Falcon	Extension library

Feel free to contribute adapters for other frameworks.

1.8 Best Practices

Some best practices for using RESTArt are recommendated here.

1.8.1 Project structure

There are many different ways to organize your RESTArt API, but here I will describe one that scales well with larger applications and maintains a nice level organization.

Here's an example directory structure:

```
blog/
    blog/
        __init__.py
        api.py              # contains the central API object
        wsgi.py             # contains the WSGI application
        resources/
            __init__.py
            posts/          # contains logic for /posts
```

```
            __init__.py
            resource.py
        tags/           # contains logic for /tags
            __init__.py
            resource.py
tests/                  # optional, contains the test code
```

See examples/blog for details.

1.9 Thanks

RESTArt is not an innovative product created from scratch. It is based on Werkzeug, and is inspired by many other awesome libraries and frameworks.

RESTArt has learned from:

- Flask

  How to use Werkzeug properly, how to elegantly support extensions and *Testing*, and how to write documentations based on Sphinx.

- Django REST framework

  How to support *Parsers* and *Renderers*.

- RestExpress

  How to map HTTP methods to resource actions, which is the inspiration of *Action mapping*.

- Nameko

  The convenience of providing a helper command-line utility *restart* (like the `nameko` utility), and the simplicity and consistency of class-based REST `resources` (like the class-based nameko services).

- Django

  How to support *Middlewares*.

- Flask-API

  The last paragraph of the Roadmap gives me the original inspiration to create framework-agnostic REST libraries, such as Resource and RESTArt.

- Resource

  Its experimental work about REST, which is valuable for RESTArt. The MongoDB-related part also becomes the predecessor of RESTArt-Mongo.

- Flask-RESTful

  The good style of its documentations.

CHAPTER 2

API Reference

If you are looking for information on a specific function, class or method, this section is for you.

2.1 API

This section covers all the interfaces of RESTArt.

2.1.1 RESTArt Object

class `restart.api.`**`RESTArt`**

The class that represents the RESTArt API and acts as the central object.

`add_rule`(*resource_class*, *uri*, *endpoint*, *methods=None*, *actions=None*)

Register a resource for the given URI rule.

Parameters

- **`resource_class`** – the resource class.
- **`uri`** – the URI registered. Werkzeug-style converters are supported here. See Rule Format for more information.
- **`endpoint`** – the endpoint for the URI.
- **`methods`** – a sequence of allowed HTTP methods. If not specified, all methods are allowed.
- **`actions`** – a dictionary with the specific action mapping pairs used to update the default *ACTION_MAP*. If not specified, the default *ACTION_MAP* will be used. See *Configuration* for more information.

`add_rule_with_format_suffix`(*resource_class*, *uri*, *endpoint*, *methods=None*, *actions=None*, *format_suffix='disabled'*)

Register a resource for the given URI rule with a possible format suffix.

Parameters **`format_suffix`** – a string indicating whether or how to support content negotiation via format suffixes on URIs. If specified, its value must be *'disabled'* (not supported), *'optional'* (supported and optional) or *'mandatory'* (supported and mandatory). If not specified, defaults to *'disabled'*.

See `add_rule()` for the meanings of other parameters.

register(*cls=None*, *prefix=None*, *pk='<pk>'*, *list_actions=None*, *item_actions=None*, *format_suffix='disabled'*)

A special decorator that is used to register a plural resource. See *Routing* for more information.

Important note:

Unlike the `route()` and `add_rule()` methods, *'OPTIONS'* is always allowed implicitly in `register()` to handle potential CORS. In order to achieve the same purpose, you must add *'OPTIONS'* into the *methods* parameter explicitly when using the `route()` and `add_rule()` methods.

Parameters

- **cls** – the class that will be decorated.
- **prefix** – the URI prefix for the resource. If not specified, the resource `name` with a leading slash will be used. For example, the *prefix* will be *'/todos'* if the resource name is *'todos'*.
- **pk** – the primary key name used to identify a specific resource. Werkzeug-style converters are supported here. See Rule Format for more information.
- **list_actions** – the action mapping pairs for the list-part URI (the URI without the primary key, see *Plural Resources* for more information). If not specified, the default *ACTION_MAP* will be used. See *Configuration* for more information.
- **item_actions** – the action mapping pairs for the item-part URI (the URI with the primary key, see *Plural Resources* for more information). If not specified, the default *ACTION_MAP* will be used. See *Configuration* for more information.
- **format_suffix** – see `add_rule_with_format_suffix()`.

route(*cls=None*, *uri=None*, *endpoint=None*, *methods=None*, *actions=None*, *format_suffix='disabled'*)

A decorator that is used to register a resource for a given URI rule. See *Routing* for more information.

Parameters

- **cls** – the class that will be decorated.
- **uri** – the URI registered. Werkzeug-style converters are supported here. See Rule Format for more information. If not specified, the resource `name` with a leading slash will be used. For example, the *uri* will be *'/todos'* if the resource name is *'todos'*.
- **endpoint** – the endpoint for the URI. If not specified, the resource `name` will be used.
- **methods** – a sequence of allowed HTTP methods. If not specified, all methods are allowed.
- **actions** – a dictionary with the specific action mapping pairs used to update the default *ACTION_MAP*. If not specified, the default *ACTION_MAP* will be used. See *Configuration* for more information.
- **format_suffix** – see `add_rule_with_format_suffix()`.

rules

A dictionary of all registered rules, which is a mapping from URI endpoints to URI rules. See `Rule` for more information about URI rules.

class `restart.api.`**Rule**(*uri*, *methods*, *handler*)

A simple class that holds a URI rule.

Parameters

- **uri** – the URI.
- **methods** – the allowed HTTP methods for the URI.
- **handler** – the handler for the URI.

2.1.2 Resource Object

class restart.resource.**Resource**(*action_map*)
The core class that represents a REST resource.

Parameters **action_map** – the mapping of request methods to resource actions.

dispatch_request(*request*, **args*, ***kwargs*)
Does the request dispatching. Matches the HTTP method and return the return value of the bound action.

Parameters

- **request** – the request object.
- **args** – the positional arguments captured from the URI.
- **kwargs** – the keyword arguments captured from the URI.

find_action(*request*)
Find the appropriate action according to the request method.

Parameters **request** – the request object.

get_parser_context(*request*, *args*, *kwargs*)
Return a dictionary that represents a parser context.

Parameters

- **request** – the request object.
- **args** – the positional arguments captured from the URI.
- **kwargs** – the keyword arguments captured from the URI.

get_renderer_context(*request*, *args*, *kwargs*, *response*)
Return a dictionary that represents a renderer context.

Parameters

- **request** – the request object.
- **args** – the positional arguments captured from the URI.
- **kwargs** – the keyword arguments captured from the URI.
- **response** – the response object.

handle_exception(*exc*)
Handle any exception that occurs, by returning an appropriate response, or re-raising the error.

Parameters **exc** – the exception to be handled.

http_method_not_allowed(*request*, **args*, ***kwargs*)
The default action handler if the corresponding action for *request.method* is not implemented.

See *dispatch_request()* for the meanings of the parameters.

log_exception(*exc*)
Logs an exception with *ERROR* level.

Parameters **exc** – the exception to be logged.

log_message(*msg*)
Logs a message with *DEBUG* level.

Parameters **msg** – the message to be logged.

logger
A `logging.Logger` object for this API.

make_response(*rv*)
Converts the return value to a real response object that is an instance of *Response*.

The following types are allowed for *rv*:

`Response`	the object is returned unchanged
`str`	the string becomes the response body
`unicode`	the unicode string becomes the response body
`tuple`	A tuple in the form `(data, status)` or `(data, status, headers)` where *data* is the response body, *status* is an integer and *headers* is a dictionary with header values.

negotiator_class
alias of `Negotiator`

perform_action(**args*, ***kwargs*)
Perform the appropriate action. Also apply all possible *process_** methods of middleware instances in *self.middlewares*.

During request phase:

`process_request()` methods are called on each request, before RESTArt calls the *action*, in order.

It should return `None` or any other value that *make_response* can recognize. If it returns `None`, RESTArt will continue processing the request, executing any other `process_request()` and, then, the *action*. If it returns any other value (e.g. a *Response* object), RESTArt won't bother calling any other middleware or the *action*.

During response phase:

`process_response()` methods are called on all responses before they'are returned to the client, in reverse order.

It must return a value that can be converted to a *Response* object by *make_response*. It could alter and return the given *response*, or it could create and return a brand-new value.

Unlike `process_request()` methods, the `process_response()` method is always called, even if the `process_request()` of the same middleware were skipped (because an earlier middleware method returned a *Response*).

Parameters

- **args** – a list of positional arguments that will be passed to the action.
- **kwargs** – a dictionary of keyword arguments that will be passed to the action.

2.1.3 Request Objects

class `restart.request.`**Request**(*initial_request*)
The base request class used in RESTArt.

Parameters **initial_request** – the initial request, which is framework-specific.

get_args()
Get the request URI parameters.

get_auth()
Get the request authorization data.

get_environ()
Get the request WSGI environment.

get_headers()
Get the request headers.

get_method()
Get the request method.

get_path()
Get the request path.

get_scheme()
Get the request scheme.

get_stream()
Get the request stream.

get_uri()
Get the request URI.

parse(*negotiator*, *parser_classes*, *parser_context=None*)
Return a request object with the data parsed, which is a dictionary. If the request payload is empty, the parsed data will be an empty dictionary.

Parameters

- **negotiator** – the negotiator object used to select the proper parser, which will be used to parse the request payload.
- **parser_classes** – the parser classes to select from. See *Parser Objects* for information about parsers.
- **parser_context** – a dictionary containing extra context data that can be useful to the parser.

class restart.request.**WerkzeugRequest**(*initial_request*)
The Werkzeug-specific request class.

get_args()
Get the request URI parameters from the Werkzeug-specific request object.

get_auth()
Get the request authorization data from the Werkzeug-specific request object.

get_environ()
Get the WSGI environment from the Werkzeug-specific request object.

get_headers()
Get the request headers from the Werkzeug-specific request object.

get_method()
Get the request method from the Werkzeug-specific request object.

get_path()
Get the request path from the Werkzeug-specific request object.

get_scheme()
Get the request scheme from the Werkzeug-specific request object.

get_stream()
Get the request stream from the Werkzeug-specific request object.

get_uri()
Get the request URI from the Werkzeug-specific request object.

2.1.4 Response Objects

class restart.response.**Response**(*data*, *status=200*, *headers=None*)
The base response class used in RESTArt.

Parameters

- **data** – the response body.
- **status** – an integer that represents an HTTP status code.
- **headers** – a dictionary with HTTP header values.

get_specific_response()
Get the framework-specific response.

render(*negotiator*, *renderer_classes*, *format_suffix*, *renderer_context=None*)
Return a response object with the data rendered.

Parameters

- **negotiator** – the negotiator object used to select the proper renderer, which will be used to render the response payload.
- **renderer_classes** – the renderer classes to select from. See *Renderer Objects* for information about renderers.
- **format_suffix** – the format suffix of the request uri.
- **renderer_context** – a dictionary containing extra context data that can be useful to the renderer.

class restart.response.**WerkzeugResponse**(*data*, *status=200*, *headers=None*)
The Werkzeug-specific response class.

get_specific_response()
Get the Werkzeug-specific response.

2.1.5 Negotiator Object

class restart.negotiator.**Negotiator**
The class used to select the proper parser and renderer.

select_parser(*parser_classes*, *content_type*)
Select the proper parser class.

Parameters

- **parser_classes** – the parser classes to select from.
- **content_type** – the target content type.

select_renderer(*renderer_classes*, *format_suffix*)
Select the proper renderer class.

Note: For simplicity, the content-negotiation here is only based on the format suffix specified in the request uri. The more standard (and also complex) Accept header is ignored.

Parameters

- **renderer_classes** – the renderer classes to select from.
- **format_suffix** – the format suffix of the request uri.

2.1.6 Parser Objects

class restart.parsers.**Parser**
The base parser class.

parse(*stream*, *content_type*, *content_length*, *context=None*)
Parse the *stream*.

Parameters

- **stream** – the stream to be parsed.
- **content_type** – the content type of the request payload.
- **content_length** – the content length of the request payload.
- **context** – a dictionary containing extra context data that can be useful for parsing.

class restart.parsers.**JSONParser**
The parser class for JSON data.

parse(*stream*, *content_type*, *content_length*, *context=None*)
Parse the *stream* as JSON.

Parameters

- **stream** – the stream to be parsed.
- **content_type** – the content type of the request payload.
- **content_length** – the content length of the request payload.
- **context** – a dictionary containing extra context data that can be useful for parsing.

class restart.parsers.**MultiPartParser**
The parser class for multipart form data, which may include file data.

parse(*stream*, *content_type*, *content_length*, *context=None*)
Parse the *stream* as a multipart encoded form.

Parameters

- **stream** – the stream to be parsed.
- **content_type** – the content type of the request payload.
- **content_length** – the content length of the request payload.
- **context** – a dictionary containing extra context data that can be useful for parsing.

class restart.parsers.**URLEncodedParser**
The parser class for form data.

parse(*stream*, *content_type*, *content_length*, *context=None*)
Parse the *stream* as a URL encoded form.

Parameters

- **stream** – the stream to be parsed.
- **content_type** – the content type of the request payload.
- **content_length** – the content length of the request payload.
- **context** – a dictionary containing extra context data that can be useful for parsing.

2.1.7 Renderer Objects

class restart.renderers.**Renderer**
The base renderer class.

render(*data*, *context=None*)
Render *data*.

Parameters

- **data** – the data to be rendered.
- **context** – a dictionary containing extra context data that can be useful for rendering.

class restart.renderers.**JSONRenderer**
The JSON renderer class.

render(*data*, *context=None*)
Render *data* into JSON.

Parameters

- **data** – the data to be rendered.
- **context** – a dictionary containing extra context data that can be useful for rendering.

2.1.8 Adapter Objects

class restart.adapter.**Adapter**(*api*)
The class used to adapt the RESTArt API to a specific framework.

Parameters **api** – the RESTArt API to adapt.

adapt_handler(*handler*, **args*, ***kwargs*)
Adapt the request object and the response object for the *handler* function.

Parameters

- **handler** – the handler function to be adapted.
- **args** – a list of positional arguments that will be passed to the handler.
- **kwargs** – a dictionary of keyword arguments that will be passed to the handler.

adapt_rules(*rules*)
Adapt the rules to be framework-specific.

get_embedded_rules()
Get the framework-specific rules used to be embedded into an existing or legacy application.

wsgi_app(*environ*, *start_response*)
The actual framework-specific WSGI application.

See *wsgi_app()* for the meanings of the parameters.

class restart.adapter.**WerkzeugAdapter**(**args*, ***kwargs*)

adapt_handler(*handler*, *request*, **args*, ***kwargs*)
Adapt the request object and the response object for the *handler* function.

Parameters

- **handler** – the handler function to be adapted.
- **request** – the Werkzeug request object.
- **args** – a list of positional arguments that will be passed to the handler.
- **kwargs** – a dictionary of keyword arguments that will be passed to the handler.

get_embedded_rules()
Get the Werkzeug-specific rules used to be embedded into an existing or legacy application.

Example:

```
# The existing Werkzeug application,
# whose URL map is `app.url_map`
app = ...
...

# The RESTArt API
from restart.api import RESTArt
api = RESTArt()
...

# Embed RESTArt into Werkzeug
from restart.serving import Service
service = Service(api)
for rule in service.embedded_rules:
    app.url_map.add(rule)
```

wsgi_app(*environ*, *start_response*)
The actual Werkzeug-specific WSGI application.

See *wsgi_app()* for the meanings of the parameters.

2.1.9 Service Object

class restart.serving.**Service**(*api*, *adapter_class=<class 'restart.adapter.WerkzeugAdapter'>*)
The service class for serving the RESTArt API.

Parameters

- **api** – the RESTArt API.
- **adapter_class** – the class that is used to adapt the api object. See *WerkzeugAdapter* for more information.

__call__(*environ*, *start_response*)
Make the Service object itself to be a WSGI application.

Parameters

- **environ** – a WSGI environment.
- **start_response** – a callable accepting a status code, a list of headers and an optional exception context to start the response

embedded_rules
The framework-specific rules used to be embedded into an existing or legacy application.

rules
The framework-specific API rules.

run(*host=None*, *port=None*, *debug=None*, ***options*)
Runs the API on a local development server.

Parameters

- **host** – the hostname to listen on. Set this to *'0.0.0.0'* to have the server available externally as well. Defaults to *'127.0.0.1'*.
- **port** – the port of the webserver. Defaults to *5000*.
- **debug** – if given, enable or disable debug mode.
- **options** – the options to be forwarded to the underlying Werkzeug server. See `werkzeug.serving.run_simple()` for more information.

wsgi_app(*environ*, *start_response*)
The actual WSGI application.

Parameters

- **environ** – a WSGI environment.
- **start_response** – a callable accepting a status code, a list of headers and an optional exception context to start the response

2.1.10 Utilities

class `restart.utils.`**locked_cached_property**(*method=None*, *name=None*)
A decorator that converts a method into a lazy property.

The method wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value.

This decorator has a lock for thread safety.

Inspired by `Flask`.

Parameters

- **method** – the method that will be decorated.
- **name** – the name of the cached property, which holds the calculated result. If not specified, the `<method-name>` (the name of the decorated method) will be used.

class `restart.utils.`**classproperty**(*fget*, **args*, ***kwargs*)
A decorator that converts a method into a read-only class property.

Note: You ought not to set the value of classproperty-decorated attributes! The result of the behavior is undefined.

class `restart.utils.`**locked_cached_classproperty**(*method=None*, *name=None*)
The lazy version of `classproperty`, which converts a method into a lazy class property.

Parameters

- **method** – the method that will be decorated.
- **name** – the name of the cached class property, which holds the calculated result. If not specified, the name with the form of `_locked_cached_classproperty_<method-name>` will be used.

`restart.utils.`**load_resources**(*module_names*)

Import all modules in `module_names` to load resources.

Example usage:

```
load_resources(['yourapi.resources.users.resource'])
load_resources(['yourapi.resources.orders.resource'])

# the equivalent of the above two lines
load_resources(['yourapi.resources.*.resource'])
```

`restart.utils.`**expand_wildcards**(*module_name*)

Expand the wildcards in `module_name` based on `sys.path`.

Suppose the directory structure of "yourapi" is as below:

```
yourapi
|-- __init__.py
`-- resources
    |-- users
    |   |-- __init__.py
    |   `-- resource.py
    `-- orders
        |-- __init__.py
        `-- resource.py
```

Then:

```
expand_wildcards('yourapi.resources.*.resource')
=>
['yourapi.resources.users.resource',
 'yourapi.resources.orders.resource']
```

`restart.utils.`**make_location_header**(*request*, *pk*)

Make the Location header for the newly-created resource.

Parameters

- **request** – the POST request object.
- **pk** – the primary key of the resource.

CHAPTER 3

Additional Notes

Design notes and changelog are here for the interested.

3.1 RESTArt Changelog

Here you can see the full list of changes between each RESTArt release.

3.1.1 Version 0.1.3

Released on Feb 21st 2016.

- Always render `HTTPException` messages into JSON
- Move `CORSMiddleware` out of RESTArt (use the RESTArt-CrossDomain extension for CORS instead)
- Remove tests for `CORSMiddleware`
- Remove the configuration options for CORS
- Update documentation
- Upgrade the Python-EasyConfig dependency

3.1.2 Version 0.1.2

Released on Dec 30th 2015.

- Refactor the Adapter module for better usage
- Add `-a, --adapter` argument to the restart utility
- Select the first renderer class if no format suffix is specified
- Add `context` keyword argument to *`Parser.parse()`* and *`Renderer.render()`*
- Use `_locked_cached_classproperty_<method-name>` (instead of `<method-name>`) as the default name of the cached class property, which holds the calculated result for the *`locked_cached_classproperty`* decorated class property
- Implement the *`get_embedded_rules`* method of WerkzeugAdapter
- Update documentation
- Update examples

3.1.3 Version 0.1.0

Released on Oct 3rd 2015.

- Add support for resource-level middleware classes
- Bind a mutable attribute (whose name starts with an underscore) to each request property
- Fix bugs for importing extensions
- Refactor the logic for parsing request data or files
- Refactor the logic for rendering response data
- Add the *SERVER_NAME* configuration option
- Add support for registering URIs with format suffixes
- Add changelog
- Add support for Python 2/3 compatibility
- Re-raise unhandled exceptions with their tracebacks
- Add `http_method_not_allowed` as the default action
- Get multiple query arguments from the request correctly

3.1.4 Version 0.0.8

Released on Jul 19th 2015.

- Update documentation
- Add makefile
- Add support for extension development
- Add the `Adapter` classes to handle framework adaptions
- Add testing tools
- Add support for Middleware
- Add support for CORS

3.1.5 Version 0.0.5

Released on Jun 26th 2015.

- Add `RESTArt` and `Service`
- Refactor `Request` and `Response`
- Add `Parser` and `Renderer`
- Handle exceptions
- Add documentation
- Add more tests
- Add logging

3.1.6 Version 0.0.2

Released on May 17th 2015.

The first release.

r

Symbols

A

C

D

E

F

G

H

J

L