RestORM Documentation

Release 0.2

Joeri Bekker

December 06, 2012

CONTENTS

RestORM allows you to interact with *resources* as if they were objects (object relational mapping), *mock* an entire API and incorporate custom *client* logic.

CHAPTER

ONE

DESCRIPTION

RestORM structures the way you access a RESTful API and allows you to access related resources. On top of that, you can easily mock an entire API and replace the real client with a mock version in unit tests. RestORM is very extensible but offers many functionalities out of the box to get up and running quickly.

Currently, RestORM works on Python 2.5+ with Python 3 support on its way.

CHAPTER

TWO

FEATURES

- Object relational mapping of API resources (Django-like but does not depend on Django at all).
- Flexible client architecture that can be used with your own or third party clients (like oauth).
- Extensive mocking module allows you to mock API responses, or even complete API's.
- Examples for Twitter and Flickr API.

CHAPTER

THREE

DOCUMENTATION

3.1 Tutorial

In this tutorial we'll walk you through the creation of a simple client-side implementation of a RESTful library API. This shows you the basics of RestORM. This example is included in the source code of RestORM and has more features, see: `restorm.examples.mock`.

Let's examine the server-side of the library API. Normally you would read the documentation of a RESTful API since there is no standard (yet) to describe a RESTful API and have a computer generate a proxy.

3.1.1 The library API

Below, you'll find an example of how the library API could be documented. The library contains books and each book is ofcourse written by an author. For the sake of this tutorial, it doesn't expose a lot of features:

Welcome to the documentation for our library API! All resources are available on `http://www.example.com/api/`. No authentication is required and responses are in JSON format.

- **GET** `book/` – Returns a list of available books in the library:

```
[
    {
        "isbn": 1,
        "title": "Dive into Python",
        "resource_url": "http://www.example.com/api/book/1"
    },
    # ...
]
```

- **GET** `book/{id}` – Returns a specific book, identified by its `isbn` number:

```
{
    "isbn": 1,
    "title": "Dive into Python",
    "author": "http://www.example.com/api/author/1"
}
```

- **GET** `author/` – Returns a list of authors that wrote the books in our library:

```
[
    {
        "id": 1,
        "name": "Mark Pilgrim",
```

```
        "resource_url": "http://www.example.com/api/author/1"
    },
    # ...
]
```

- **GET** `author/{id}` – Returns a specific author, identified by its `id`:

```
{
    "id": 1,
    "name": "Mark Pilgrim",
}
```

- **POST** `search/` – Searches the library and returns matching books:

```
{
    "query": "Python"
}
```

```
[
    {
        "isbn": 1,
        "title": "Dive into Python",
        "resource_url": "http://www.example.com/api/book/1"
    },
    # ...
]
```

3.1.2 Create a client

A typical client that can talk to a RESTful API using JSON, is no more then:

```
from restorm.clients.jsonclient import JSONClient

client = JSONClient(root_uri='http://www.example.com/api/')
```

Since this tutorial uses a non-existant library API, the client doesn't work. We can however mock its intended behaviour.

3.1.3 Create a mock API

In order to test your client, you can emulate a whole API using the `MockApiClient`. However, sometimes it's faster or easier to use a single, predefined response, using the `MockClient`.

Since our library API is not that complex it is very straighforward to mock the entire API, so we'll do just that. The `MockApiClient` takes two arguments. The `root_uri` is the same as for regular clients but in addition, there is the `responses` argument. The `responses` argument takes a `dict` of available resource URLs, supported methods, response headers and data. It's best to just look at the example below to understand its structure.

The mock API below contains a list of books and a list of authors. To keep it simple, both list resources contain only 1 item:

```
from restorm.clients.mockclient import MockApiClient

mock_client = MockApiClient(
    responses={
        'book/': {'GET': ({'Status': 200}, [{'isbn': 1, 'title': 'Dive into Python', 'resource_url':
        'book/1': {'GET': ({'Status': 200}, {'isbn': 1, 'title': 'Dive into Python', 'author': 'http
```

```
        'author/': {'GET': ({'Status': 200}, [{'id': 1, 'name': 'Mark Pilgrim', 'resource_url': 'http
        'author/1': {'GET': ({'Status': 200}, {'id': 1, 'name': 'Mark Pilgrim'})}
        'search/': {'POST': ({'Status': 200}, [{'isbn': 1, 'title': 'Dive into Python', 'resource_url
    },
    root_uri='http://www.example.com/api/'
)
```

It's worth mentioning that you are not creating an API here, you are mocking it. Simple and limited responses are usually fine. If the API would contain huge responses, you can also use the `FileResponse` class to read the mock response from a file.

3.1.4 Define resources

We start with the most basic resource, the `Author` resource:

```
from restorm.resource import Resource

class Author(Resource):
    class Meta:
        list = r'^author/$'
        item = r'^author/(?P<id>\d+)$'
```

We subclass `Resource` and add an inner `Meta` class. In the `Meta` class we add two attributes that are internally used by the `ResourceManager` to perform `get` and `all` operations:

- **list** – The URL-pattern to retrieve the list of authors.
- **item** – The URL-pattern to retrieve a specific author by `id`.

For our `Book` resource, it's also possible to search for books. We can add this functionality with a custom `ResourceManager`:

```
from restorm.resource import ResourceManager

class BookManager(ResourceManager):
    def search(self, query, client=None):
        response = client.post('search/', '{ "query": "%s" }' % query)
        return response.content
```

No validation or exceptions in the request and response are handled in the above example for readability reasons. In a production environment, you should.

We also need to define the `Book` resource itself and add our custom manager by adding an instance of it to the `objects` attribute on the resource.

```
class Book(Resource):

    objects = BookManager()

    class Meta:
        list = r'^book/$'
        item = r'^book/(?P<isbn>\d)$'
```

3.1.5 Bringing it all together

You can access the `Book` resource and the related `Author` resource using the `mock_client`, or if the library API was real, use the `client`. We can pass the client to use as an argument to all manager functions (like `get`, `all` and

also the `search` function we defined earlier).

```
>>> book = Book.objects.get(isbn=1, client=mock_client) # Get book with ISBN number 1.
>>> book.data['title'] # Get the value of the key "name".
u'Dive into Python'
>>> book.data['author'] # Get the value of the key "author".
u'http://www.example.com/api/author/1'
>>> author = book.data.author # Perform a GET on the "author" resource.
>>> author.data['name']
u'Mark Pilgrim'
```

Our custom manager added a search function, let's use it:

```
>>> Book.objects.search(query='python', client=mock_client)
[{'isbn': 1, 'title': 'Dive into Python', 'resource_url': 'http://www.example.com/api/book/1'}]
```

Since it's mocked, we could search for anything and the same response would come back over and over.

Note: As you may have noticed, the response content contains actual Python objects. The `MockApiClient` simply returns the content as is. If you prefer using JSON, you can achieve the same behaviour with:

```
from restorm.clients.mockclient import BaseMockApiClient
from restorm.clients.jsonclient import JSONClientMixin

class MockJSONApiClient(BaseMockApiClient, JSONClientMixin):
    pass

client = MockJSONApiClient(
    responses={
        # Note the difference. The content is now JSON.
        'book/1': {'GET': ({'Status': 200, 'Content-Type': 'application/json'}, '{"id": 1, "title": "
        # ...
    },
    root_uri='http://www.example.com/api/'
)
```

3.2 Clients

At the heart of RestORM you have a `Client`. The client simply allows you to communicate with an API and the one RestORM uses is built on top of the excellent httplib2 library. However, you are free to use any HTTP client library as long as you add the RestORM mixins.

3.2.1 Create a client

Most RESTful API support the JSON format for their responses. A client that can handle JSON is therefore included in RestORM.

class `restorm.clients.jsonclient.`**`JSONClient`**(**args*, ***kwargs*)

Client that handles JSON requests and responses.

```
from restorm.clients.jsonclient import JSONClient

client = JSONClient(root_uri='http://www.example.com/api/')
```

The `JSONClient` is actually a combination of the classes `BaseClient` and `JSONClientMixin`. The `BaseClient` is responsible for communicating with the API while the `JSONClientMixin` adds serialization and deserialization for JSON. `JSONClient` itself is a subclass of `ClientMixin` which exposes various convenience methods.

class `restorm.clients.base.`**`BaseClient`**(**args*, ***kwargs*)

Simple RESTful client based on `httplib2.Http`.

`request`(*uri*, *method='GET'*, *body=None*, *headers=None*, *redirections=5*, *connection_type=None*)

Creates a `Request` object by calling `self.create_request(uri, method, body, headers)` and performs the low level HTTP-request using this request object. A `Response` object is created with the data returned from the request, by calling `self.create_response(response_headers, response_content, request)` and is returned.

class `restorm.clients.base.`**`ClientMixin`**

This mixin contains the attribute `MIME_TYPE` which is `None` by default. Subclasses can set it to some mimetype that will be used as `Content-Type` and `Accept` header in requests.

If the `MIME_TYPE` is also found in the `Content-Type` response headers, the response contents will be deserialized.

`serialize`(*data*)

Produces a serialized version suitable for transfer over the wire.

Subclasses should override this function to implement their own serializing scheme. This implementation simply returns the data passed to this function.

Data from the serialize function passed to the deserialize function, and vice versa, should return the same value.

Parameters **data** – Data.

Returns Serialized data.

`deserialize`(*data*)

Deserialize the data from the raw data.

Subclasses should override this function to implement their own deserializing scheme. This implementation simply returns the data passed to this function.

Data from the serialize function passed to the deserialize function, and vice versa, should return the same value.

Parameters **data** – Serialized data.

Returns Data.

`create_request`(*uri*, *method*, *body=None*, *headers=None*)

Returns a `Request` object.

`create_response`(*response_headers*, *response_content*, *request*)

Returns a `Response` object.

`get`(*uri*)

Convenience method that performs a GET-request.

`post`(*uri*, *data*)

Convenience method that performs a POST-request.

`put`(*uri*, *data*)

Convenience method that performs a PUT-request.

delete(*uri*)
Convenience method that performs a DELETE-request.

3.2.2 Writing your own client

You can tweak almost everything about the client architecture but the minimum requirements to work with RestORM resources are:

1. You use the `ClientMixin` in your client.
2. Have a `request` function that returns a `Response` object.

A minimal implementation would be:

```
from restorm.clients.base import ClientMixin

class MyClient(ClientMixin):
    def request(self, uri, method, body=None, headers=None):
        # Create request.
        request = self.create_request(uri, method, body, headers)

        # My very own client doesn't need an internet connection!
        response_headers, response_content = {'Status': 200}, 'Hello world!'

        # Create response.
        return self.create_response(response_headers, response_content, request)
```

The above client doesn't do much but it shows how to create your own client:

```
>>> client = MyClient()
>>> response = client.get('/hello/')
>>> response.content
'Hello world!'
>>> response.headers
{'Status': 200}
>>> response.request.uri
'/hello/'
```

You can override any of the `ClientMixin` functions to add custom behaviour:

```
class MyClient(ClientMixin):
    # ...

    def create_request(uri, method, body=None, headers=None):
        """
        Make sure the URI is absolute.
        """
        if not uri.startswith('/'):
            uri = '/%s' % uri
        return super(MyClient, self).create_request(uri, method, body, headers)

    def create_response(response_headers, response_content, request):
        """
        Let everone know that it was MyClient that processed the response.
        """
        response_headers.update({
            'X-Response-Updated-By': 'MyClient'
        })
        return super(MyClient, self).create_response(response_headers, response_content, request)
```

```
>>> client = MyClient()
>>> response = client.get('hello/')
>>> response.content
'Hello world!'
>>> response.headers
{'Status': 200, 'X-Response-Updated-By': 'MyClient'}
>>> response.request.uri
'/hello/'
```

RestORM can handle JSON as response format from RESTful API's. Implementing your own format requires you to override the `serialize` and `deserialize` function in your own client class.

3.2.3 Using different HTTP client libraries

There are lots of different client libraries. RestORM chose for `httplib2` as default HTTP client library because it's an active project with built-in caching and overall has the best performance.

Do not let the above stop you from using your own preferred HTTP client library like requests, oauth2, or even the standard library httplib

Example: OAuth

Many API's use OAuth, an open standard for authorization. It's quite simple to incorporate the oauth2 library in combination with one of the client mixins, for example the `JSONClientMixin` and override the `request` method to make a request using OAuth:

```
import oauth2 as oauth
from restorm.clients.jsonclient import JSONClientMixin

class OauthClient(oauth.Client, JSONClientMixin):
    def request(self, uri, method='GET', body=None, headers=None, *args, **kwargs):
        # Create request.
        request = self.create_request(uri, method, body, headers)

        # Perform request.
        response_headers, response_content = super(OauthClient, self).request(request.uri, request.me

        # Create response.
        return self.create_response(response_headers, response_content, request)
```

Once we have this, we can do:

```
>>> consumer = oauth.Consumer(key='YOUR_KEY', secret='YOUR_SECRET')
>>> token = oauth.Token(key='YOUR_TOKEN', secret='YOUR_TOKEN_SECRET')
>>> client = OauthClient(consumer, token)
```

3.3 Resources

REST-style architectures consist of *Clients* and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of *Resources*. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

In RestORM, a `Resource` is the single, definitive source of data about a specific API endpoint. It contains the essential properties and behaviors of the data you're accessing. Generally, each resource maps to a single API endpoint.

3.3.1 Defining resources

Imagine a RESTful library API, like described in the *Tutorial* and the *Mocking* part of this documentation. You can request a list of books in the library and a list of authors. The API provides data about a specific book, like its title and author. To represent the book on our client side, we define a `Book` resource that inherits from `Resource`.

We also define an inner `Meta` class that contains meta properties about the book resource. It contains for example an attribute `item` that holds a relative URL pattern for retrieving a single book.

```
from restorm.resource import Resource

class Book(Resource):
    class Meta:
        item = r'^book/(?P<isbn>\w+)$'
```

The `item` attribute holds a URL pattern that is a regular expression describing on what URL a single book representation can be retrieved. The `r` in front of the string in Python means to take the string "raw" and nothing should be escaped.

In Python regular expressions, the syntax for named regular-expression groups is `(?P<name>pattern)`, where `name` is the name of the group and `pattern` is some pattern to match. In the example above the name `isbn` can be any word of any length. A valid relative URL would be: `book/1` or `book/abc123`.

As you may have noticed, nothing is said about the book's representation. There is no strict definition of what should be a book. The server decides this for you, or you can manually pass in data. All information passed to the `Resource` constructor as first argument, is available in the `data` attribute.

```
>>> book = Book({'title': 'Hello world', 'subtitle': 'A good start'})
>>> book.absolute_url
None
>>> book
<Book: None>
>>> book.data['title']
'Hello world'
```

You can add any custom function to your resource class to help you work with the data representation.

```
from restorm.resource import Resource

class Book(Resource):
    class Meta:
        item = r'^book/(?P<isbn>\w+)$'

    @property
    def full_title(self):
        return '%s: %s' % (self.data['title'], self.data['subtitle'])

>>> book = Book({'title': 'Hello world', 'subtitle': 'A good start'})
>>> book.full_title
'Hello world: A good start'
```

The default representation of a `Resource` is the class name followed by the (absolute) URL of the retrieved representation, in the example there was none so the value is `None`.

You can override this with the `__unicode__` function:

```
class Book(Resource):
    # ...

    def __unicode__(self):
        if 'title' in self.data:
            return self.data['title']
        else:
            return '(unknown title)'
```

```
>>> book = Book({'title': 'Hello world', 'subtitle': 'A good start'})
>>> book.absolute_url
None
>>> book
<Book: Hello world>
```

3.3.2 Resource managers

A `ResourceManager` is the interface through which API requests can be performed on a `Resource`. At least one manager exists for every resource.

By default, RestORM adds a `ResourceManager` with the name `objects` to every RestORM resource class.

Let's assume we already have a client ready, as described in *Clients* but we use our mock client so you can test the demonstrated code snippets yourself.

```
>>> from restorm.examples.mock.api import LibraryApiClient
>>> client = LibraryApiClient()
```

Our `Book` resource already allows us to get a single book from the library API:

```
>>> book = Book.objects.get(isbn=1, client=client)
>>> book.data['title']
u'Dive into Python'
```

To make life a little easier, we can stop passing the `client` argument by setting our client as the default client:

```
>>> from restorm.conf import settings
>>> settings.DEFAULT_CLIENT = client
```

You can typically add this to your (Django) project settings so you won't have to bother about it anymore. We can now do:

```
>>> book = Book.objects.get(isbn=1)
>>> book.data['title']
u'Dive into Python'
```

class `restorm.resource.`**`ResourceManager`**

`get`(*client=None*, *query=None*, *uri=None*, ***kwargs*)

Returns the object matching the given lookup parameters. You should pass all arguments required by the resource URL pattern `item`, as specified in the `Meta` class of the resource.

Parameters

- **client** – The client to retrieve the object from the API. By default, the default client is used. If no client and no default client are specified, a `ValueError` is raised.
- **query** – A `dict` with additional query string arguments. An empty `dict` by default.

- **uri** – Rather than passing the resource URL pattern arguments, you can also provide a complete URL. This URL must match the resource URL pattern.

```
>>> Book.objects.get(isbn=1)
<Book: http://www.example.com/api/book/1>
```

all(*client=None*, *query=None*, *uri=None*, ***kwargs*)
Returns the raw response for the list of objects. You should pass all arguments required by the resource URL pattern `collection`, as specified in the `Meta` class of the resource.

Parameters

- **client** – The client to retrieve the object from the API. By default, the default client is used. If no client and no default client are specified, a `ValueError` is raised.
- **query** – A `dict` with additional query string arguments. An empty `dict` by default.
- **uri** – Rather than passing the resource URL pattern arguments, you can also provide a complete URL. This URL must match the resource URL pattern.

```
>>> Book.objects.all() # Returns a raw response.
```

3.3.3 Related resources

You can access all API resources by creating a `Resource` class for each API resource.

..sourecode:: python

class Author(Resource):

class Meta: item = r'^author/(?P<id>d+)$'

```
>>> book = Book.objects.get(isbn=1)
>>> book.data['author']
u'http://www.example.com/api/author/1'
>>> author = Author.objects.get(uri=book.data['author'])
>>> author.data['name']
u'Mark Pilgrim'
```

RestORM is aware of the API endpoint URL in the book resource. We can simply do:

..sourcecode:: python

```
>>> book = Book.objects.get(isbn=1)
>>> book.data.author.data['name']
u'Mark Pilgrim'
```

Even if we did not define the `Author` resource, the above would be valid. A generic resource is then used to represent the author.

3.4 Mocking

The mock client that comes with RestORM is an excellent method to simulate server side responses that you expect from a real REST API. You typically use it in your unit tests.

You can switch out your client with a mock client and simply work as usual, given that you mocked some typical responses. The mocking works transparently with `Resource` classes. Just pass your mock client to it.

Where you would normally have:

```
from restorm.resource import Resource

class Book(Resource):
    class Meta:
        item = r'^book/(?P<isbn>\w+)$'

>>> from restorm.clients.jsonclient import JSONClient
>>> client = JSONClient(root_uri='http://www.example.com/api/')

>>> book = Book.objects.get(isbn='978-1441413024', client=client)
>>> book.data['title']
u'Dive into Python'
```

You can replace it all with mocking behaviour that does not rely on actual communication with the external API:

```
from restorm.clients.mockclient import MockApiClient
mock_client = MockApiClient(
    root_uri='http://www.example.com/api/',
    responses={
        'book/978-1441413024': {
            'GET': ({'Status': 200}, {'title': 'Dive into Python'})
        }
    }
)

>>> book = Book.objects.get(isbn='978-1441413024', client=mock_client)
>>> book.data['title']
u'Dive into Python'
```

There are several approaches.

3.4.1 Mocking single predefined responses

You can mock a simple specific response in small tests using the `MockResponse` class and subclasses, and `MockClient`. You typically use these classes where you do not want to mock an entire API and can suffice with just a few responses that don't happen irregularly.

class `restorm.clients.mockclient.`**`MockResponse`**(*headers*, *content*)

Main class for mocked responses. Headers can be provided as dict. The content is simply returned as response and is usually a string but can be any type of object.

```
>>> from restorm.clients.mockclient import MockResponse
>>> response = MockResponse({'Status': 200}, {'foo': 'bar'})
>>> response.headers
{'Status': 200}
>>> response.content
{'foo': 'bar'}
```

class `restorm.clients.mockclient.`**`StringResponse`**(*headers*, *content*)

A response with stringified content.

```
>>> from restorm.clients.mockclient import StringResponse
>>> response = StringResponse({'Status': 200}, '{}')
>>> response.content
'{}'
```

class `restorm.clients.mockclient.`**`FileResponse`**(*headers*, *filepath*)

A response with the contents of a file, read by absolute file path.

```
>>> from restorm.clients.mockclient import FileResponse
>>> response = FileResponse({'Status': 200}, 'response.json')
```

Using the above classes, you can pass desired responses to your mock client.

class restorm.clients.mockclient.**MockClient**(**args*, ***kwargs*)

A mock client, emulating the rest client. The client returns predefined responses. You can add any `MockResponse` sub class to the `responses` argument.

It doesn't matter what URI you request or what data you pass in the body, the first response you added is just returned on the first request.

Responses are popped from a queue. This means that if you add 3 responses, you can only make 3 requests before a `ValueError` is raised.

```
>>> from restorm.clients.mockclient import MockClient, StringResponse
>>> desired_response = StringResponse({'Status': 200}, '{}')
>>> mock_client = MockClient('http://mockserver/', responses=[desired_response,])
>>> response = mock_client.get('/') # Can be any URI.
>>> response.content
u'{}'
>>> response.status_code
200
>>> response = mock_client.get('/') # Another call.
ValueError: Ran out of responses when requesting: /
```

3.4.2 Mocking entire servers

If you are going to test alot against a certain API (your own, or an external one), it might be a good idea to make mock the a part of the API. You typically use this to make functional tests or broader unit tests. You can use the `MockApiClient` for this purpose.

You can even create a web server from a `MockApiClient` instance to "browse" through your mock API using a browser or keep it running to let your application talk to it.

class restorm.clients.mockclient.**MockApiClient**(**args*, ***kwargs*)

A client that emulates communicating with an entire mock API.

Specify each resource and some headers and/or content to return. You can use a `tuple` as response containing the headers and content, or use one of the available `MockResponse` (sub)classes to return the contents of a string or file.

The structure of the `responses` is:

```
{<relative URI>: {<HTTP method>: ({<header key>: <header value>, ...}, <response content>)}}
```

```
>>> from restorm.clients.mockclient import MockApiClient, StringResponse
>>> mock_api_client = MockApiClient(responses={
...     'book/': {
...         'GET': ({'Status': 200}, [{'id': 1, 'name': 'Dive into Python', 'resource_url': 'htt
...         'POST': ({'Status': 201, 'Location': 'http://localhost/api/book/2'}, ''),
...     },
...     'book/1': {'GET': ({'Status': 200}, {'id': 1, 'name': 'Dive into Python', 'author': 'htt
...     'author/': {'GET': ({'Status': 200}, [{'id': 1, 'name': 'Mark Pilgrim', 'resource_url':
...     'author/1': {'GET': FileResponse({'Status': 200}, 'response.json')}
... }, root_uri='http://localhost/api/')
...
>>> response = mock_api_client.get('http://localhost/api/book/1')
>>> response.content
```

```
{'id': 1, 'name': 'Dive into Python', 'author': 'http://localhost/api/author/1'}
>>> response.status_code
200
```

get_response_from_request(*request*)
You may override this method to implement your own response logic based on given request. You can even modify the `self.responses` based on some POST, PUT or DELETE request.

This is the only method that looks at `self.responses`. Therefore, overriding this method also allows you to create a custom format for this container variable or even mutate the `responses` variable based on the request.

create_server(*ip_address*, *port*, *handler=None*)
Creates a server instance and returns it. The server instance has access to this mock to provide the responses.

```
>>> from restorm.clients.mockclient import MockApiClient, StringResponse
>>> mock_api_client = MockApiClient(responses={'/': {'GET': ({'Status': 200}, 'My homepage')
>>> server = mock_api_client.create_server('127.0.0.1', 8000)
>>> server.serve_forever()
```

Example: Library API

An extensive example is given in the `restorm.examples.mock` module that extends the mock API from the *Tutorial*.

Note: This example is also used in internal unit tests. The `MockApiClient` and related classes are specifically made for the purpose of unit testing, or "playground" testing.

The example here is a JSON webservice. You can instantiate it and perform requests from the console:

```
>>> from restorm.examples.mock.api import LibraryApiClient
>>> client = LibraryApiClient()
>>> response = client.get('author/1')
>>> response.raw_content
'{"books": [{"resource_url": "http://localhost/api/book/978-1441413024", "isbn": "978-1441413024", "t
```

You can also start it as a server and connect to it with your browser, or let your application connect to it:

```
$ python -m restorm.examples.mock.serv 127.0.0.1:8000
```

Shut it down with CTRL-C. The above Python script basically does:

```
>>> server = client.create_server()
>>> server.serve_forever()
```

3.4.3 Expanding on what's there

Both the `MockClient` and `MockApiClient` constist of a base class and a mixin that handles the specifics. The `MockClient` class is defined as:

```
from restorm.clients.base import ClientMixin
from restorm.clients.mockclient import BaseMockClient

class MockClient(BaseMockClient, ClientMixin):
    pass
```

You can easily use these classes for your own use. Actually, creating a mock client that serves JSON is nothing more than:

```
from restorm.clients.jsonclient import JSONClientMixin

class MockJSONClient(BaseMockClient, JSONClientMixin):
    pass
```

With the `MockApiClient` you can also use these mixins:

```
from restorm.clients.jsonclient import JSONClientMixin, json
from restorm.clients.mockclient import BaseMockApiClient

class MockApiJSONClient(BaseMockApiClient, JSONClientMixin):
    pass
```

CHAPTER

FOUR

INSTALLATION

RestORM is on PyPI, so you can simply use:

```
$ pip install restorm
```

If you want the latest development version, get the code from Github:

```
$ pip install -e git+git://github.com/joeribekker/restorm.git#egg=restorm
```

CHAPTER

FIVE

CONTRIBUTE

1. Get the code from Github:

   ```
   $ git clone git://github.com/joeribekker/restorm.git
   ```

2. Create and activate a virtual environment:

   ```
   $ cd restorm
   $ virtualenv .
   $ source bin/activate
   ```

3. Setup the project for development:

   ```
   $ python setup.py develop
   ```

4. Start hacking!

CHAPTER

SIX

TESTING

RestORM has a whooping 90% test coverage. Although reaching 100% is not a goal by itself, I consider unit testing to be essential during development.

Performing the unit tests yourself:

```
pip install nose
python setup.py nosetest
```

Note: Until a version 1.0 release, backwards incompatible changes may be introduced in future 0.x versions.

PYTHON MODULE INDEX

r