txYoga Documentation

Release 0

The txyoga authors

Sep 26, 2017

Contents

This is the documentation for txyoga, a REST toolkit for Twisted.

Contents:

CHAPTER 1

Tutorial

The tutorial consists of a bunch of progressively more complex examples, showcasing more and more features as you go along. You should have a decent working knowledge of Python and some idea of how REST works. Some understanding of Twisted, particularly `twisted.web` and the way it does object publishing probably, wouldn't hurt either; but that's mostly important for more advanced use cases.

The tutorial examples themselves are `rpy` files. These are basically Python files that expose a resource. This way, we can run the tutorials with minimal boilerplate. You might want to do something more sophisticated in a real application, but that's outside the scope of this tutorial.

Each example starts with a short summary, followed by walking you through the documented example code. Finally, you can try the example out interactively.

Serving the tutorial examples

In order for the interactive examples to work, Twisted needs to be serving the tutorial. There are two ways of doing that: using the helper script, or invoking `twistd` manually. The former neatly daemonizes `twistd` and cleans up the log file, but the latter makes it a bit easier to see what's going on under the hood in terms of HTTP requests.

The helper script will only work on *nix-like environments. Windows users should run `twistd` manually.

Using the helper script

From the `docs` directory:

```
./serveTutorial start
```

When you're done:

```
./serveTutorial stop
```

Running `twistd` manually

Run following command from the txyoga base directory:

```
twistd -n web --path doc/tutorial
```

The `-n` flag makes `twistd` stay in the foreground instead of daemonizing. The other arguments should be fairly self-explanatory. You should see something similar to this:

```
2011-04-17 21:29:28+0200 [-] Log opened.
2011-04-17 21:29:28+0200 [-] twistd 11.0.0+r31557 (/usr/bin/python 2.7.1) starting
↪up.
2011-04-17 21:29:28+0200 [-] reactor class: twisted.internet.selectreactor.
↪SelectReactor.
2011-04-17 21:29:28+0200 [-] twisted.web.server.Site starting on 8080
2011-04-17 21:29:28+0200 [-] Starting factory <twisted.web.server.Site instance at
↪0x2e7f3b0>
```

As you can see, Twisted is listening for connections on port 8080.

List of examples

Accessing collections and elements

To introduce txyoga, we're going to start with a simple example of a bunch of employees who work at a startup (because blogs with articles are just cliché).

Example code

```
# -*- mode: python; coding: utf-8 -*-
from twisted.web.resource import IResource

from txyoga import Collection, Element


class Company(Collection):
    """
    A company.
    """
    exposedElementAttributes = "name",



class Employee(Element):
    """
    An employee at a company.
    """
    exposedAttributes = "name", "title"

    def __init__(self, name, title, salary):
        self.name = name
        self.title = title
        self.salary = salary



startup = Company()

startup.add(Employee(u"lvh", u"CEO", 10000))
startup.add(Employee(u"asook", u"Junior intern", 1))

resource = IResource(startup)
```

First, the code defines a collection class called `Company`. Collections are containers for elements. The collection class has an attribute called `exposedElementAttributes`. These are the attribute the collection will expose about its elements. In this case, we want to show the names of our employees when the collection is queried. Obviously, the elements should have such a `name` attribute.

Note the comma at the end: the attribute is an iterable of the collection names (in this case, a tuple with one element). If you left it out, txyoga would just see a string, which is *also* an iterable of strings, but it's quite likely that you don't want to expose the attributes `'n', 'a', 'm', 'e'`...

Once the collection is defined, the module defines an element class to go in the collection. In this example, that's `Employee`. Its instances go in instances of the previously defined Company collection. The element class has an attribute called `exposedAttributes`. These are the attributes returned when the element itself is requested. Our fictuous employees tell people their names and are quite proud of their titles, but they're shy when it comes to their salary.

Two sample employees are created and added to the collection, so there's some data to play with when you try this code out.

Finally, it builds a resource from the collection, which allows it to be served. Resources are the things that Twisted serves – they're roughly equivalent to the concept of a web page or a view. Twisted formally specifies what a resource *is* in an interface called `IResource`. Although the collection isn't a resource itself, it can be turned into one, in a process called "adaptation". That wraps the collection with an object (called the adapter) which behaves like a resource. That allows for a clean separation of concerns: being a collection on one side, and serving that collection on the other.

Both Collections and Elements are adaptable to resources. This means they're easy to integrate with existing Twisted Web object hierarchies.

Trying it out

You can access the entire collection by sending an HTTP GET request to the root resource. txyoga uses JSON as the default serialization format. The examples use the `json` and `httplib`[1] modules from the standard library.

In the interest of readability, the examples use some helpers to make requests. These will build the appropriate URL and make the HTTP request. JSON decoding is done manually, so the response headers can still be verified. If you want to use these yourself, they are available in `doc/tutorial/util.py`.

```
# -*- coding: utf-8 -*-
"""
Utilties for making HTTP requests to txyoga tutorial examples.
"""
import functools
import httplib
import urllib


def buildPath(*parts):
    return "/" + "/".join(urllib.quote(part) for part in parts)


class Example(object):
    """
    A txyoga tutorial example.
    """
    def __init__(self, exampleName, host="localhost:8080"):
        self._makeConnection = functools.partial(httplib.HTTPConnection, host)
        self._buildPath = functools.partial(buildPath, exampleName + ".rpy")
```

[1] The examples use `httplib` and not `urllib` or `urllib2`, because it's less of an abstraction over HTTP. It supports for HTTP PUT and DELETE, which are obviously important. Although `urllib2` can be hacked at so that it does support those verbs, it's not very pretty. `urllib` itself is still used because it provides `quote`.

```
    def _makeRequest(self, method, body, headers, *parts):
        """
        Makes an HTTP request to the tutorial example.
        """
        path = self._buildPath(*parts)
        connection = self._makeConnection()
        connection.request(method, path, body, headers)
        return connection.getresponse()


    def get(self, *parts):
        """
        Gets a particular collection or element.
        """
        return self._makeRequest("GET", "", {"Accept": "application/json"}, *parts)


    def delete(self, *parts):
        """
        Deletes a particular element.
        """
        return self._makeRequest("DELETE", "", {}, *parts)


    def put(self, body, headers, *parts):
        """
        Puts a particular element in a collection.
        """
        return self._makeRequest("PUT", body, headers, *parts)
```

Although it has other methods, we'll only be using `get` in this example.

The `buildPath` function creates a URL path, roughly as you'd expect:

```
>>> buildPath("test.rpy")
'/test.rpy'
>>> buildPath("test.rpy", "lvh", "minions")
'/test.rpy/lvh/minions'
```

Create an Example object for this tutorial example:

```
>>> example = Example("accessing")
```

Accessing the collection

```
>>> response = example.get()
>>> json.load(response)
{u'prev': None, u'results': [{u'name': u'lvh'}, {u'name': u'asook'}], u'next':␣
↪None}
```

The important key here is `results`. As you can see, has two entries, one for every employee. Each entry is a dictionary, containing the name of that employee. This is because `Company.exposedElementAttributes` only has `name` in it.

The two remaining keys, `prev` and `next`, are there for supporting pagination. In this case, they're both `None`, indicating that there is neither a previous nor a next page. A later tutorial example will demonstrate how paginating collections works.

txyoga is being a good HTTP citizen behind the scenes, telling you the date, the web server serving the request, and content length and type behind the scenes.

```
>>> headers = response.getheaders()
>>> headerKeys = [k for k, v in headers]
>>> expectedKeys = ["date", "content-type", "transfer-encoding", "server"]
>>> assert all(k in headerKeys for k in expectedKeys)
```

txyoga will never return responses with missing content types. A later tutorial example on content type support in txyoga will elaborate on this.

```
>>> next(v for k, v in headers if k == "content-type")
'application/json'
```

Admittedly, it really is Twisted Web telling you it's serving the request and the time it served it, not txyoga, but it's still nice:

```
>>> serverName = next(v for k, v in headers if k == "server")
>>> assert "TwistedWeb" in serverName
```

Accessing an element

Let's look at this `lvh` employee from up close next.

```
>>> response = example.get("lvh")
>>> json.load(response)
{u'name': u'lvh', u'title': u'CEO'}
```

As expected, you get a dictionary back with the `name` and `title` attributes of the employee, because those are the keys in `Employee.exposedAttributes`.

Although `exposedElementAttributes` and `exposedAttributes` are typically defined on the class they are looked up on the instance for each object. You could have a particular company that does expose the titles of its employees when you query it, or a particular employee that will divulge his salary.

As before, txyoga will serve the content type correctly:

```
>>> next(v for k, v in response.getheaders() if k == "content-type")
'application/json'
```

Review

In this example, we've:

1. shown what basic txyoga code looks like
2. shown how the tutorial example helpers work
3. accessed collections and their elements over HTTP

Creating and deleting elements

In computing, there is a common term called CRUD: creating, reading (accessing), updating and deleting. In the *previous example*, you learned how to retrieve collections and elements over HTTP. In this example, we'll show how creating and deleting elements in collections works.

Example code

```
# -*- mode: python; coding: utf-8 -*-
cache()

from twisted.web.resource import IResource

from txyoga import Collection, Element


class Employee(Element):
    """
    An employee at a company.
    """
    exposedAttributes = "name", "title"
    updatableAttributes = "salary", "title"

    def __init__(self, name, title, salary):
        self.name = name
        self.title = title
        self.salary = salary



class Company(Collection):
    """
    A company.
    """
    defaultElementClass = Employee
    exposedElementAttributes = "name",



startup = Company()

startup.add(Employee(u"lvh", u"CEO", 10000))
startup.add(Employee(u"asook", u"Junior intern", 1))

resource = IResource(startup)
```

The example code is very similar, but not identical to that in the previous example.

First of all, there is the `updatableAttributes` attribute on the element class. This is an iterable of the attribute names for which updates are allowed. In this case, it's `salary` and `title` – we can give people new positions and give them raises, but not change their names.

Then there is the `elementClass` attribute on the collection class. By default, this class will be instantiated when new elements are added. In this case, new elements will be instances of the `Employee` class.

Trying it out

Like before, start by creating the helper object for this example:

```
>>> example = Example("crud")
```

Creating an element

First, we'll hire a new employee in the company. In REST, there are two usual ways of creating elements:

> 1. If you just want to add an element to a Collection and you don't care what URL it will be available under, POST to the collection. 2. If you want to make an element accessible at a particular path, PUT it there.

Just like everywhere else, when faced with doubt, txyoga refuses the temptation to guess. In this case, when you provide data through POST or PUT, specifying the encoding is mandatory.

```
>>> data = {"name": u"alice", "salary": 100, "title": "engineer"}
>>> headers = {"content-type": "application/json"}
>>> response = example.put(json.dumps(data), headers, "alice")
```

As usual, we get the appropriate response:

```
>>> response.status, response.reason
(201, 'Created')
```

If accepted but not yet created, a txyoga server may optionally return 202 (Accepted).

Updating an element

The company's success has really gotten to lvh's head. He's not happy unless we give him a ridiculous new title.

To do that, we update his record. In REST, updates are typically done using a PUT request.

```
>>> def getTitle(employee="lvh"):
...     response = example.get(employee)
...     return json.load(response)["title"]
>>> assert getTitle() == u'CEO'
>>> headers = {"content-type": "application/json"}
>>> data = {"title": u"Supreme Commander"}
>>> response = example.put(json.dumps(data), headers, "lvh")
>>> response.status, response.reason
(200, 'OK')
>>> assert getTitle() == u'Supreme Commander'
```

Deleting an element

Next, we'll remove poor Asook from the workforce. As you might expect from a REST toolkit, you do that with the DELETE verb, or, with our helper abstraction layer, the `delete` method.

```
>>> response = example.delete("asook")
```

The server will respond with the appropriate response code:

```
>>> response.status, response.reason
(204, 'No Content')
```

When you access the collection again, Asook is missing, as expected:

```
>>> employees = json.load(example.get())["results"]
>>> assert u"asook" not in employees
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search