



JSONStore

robertodealmeida

[Clone](#) [Fork](#) [Compare](#)

Overview

Source

Commits

Branches

Pull requests

Issues 1

Wiki

Downloads

Bitbucket is a code hosting site with unlimited public and private repositories. We're also free for small teams!

[Sign up for free](#)HTTPS 

1

Branch



1

Tag



2

Forks



18

Watchers

Owner	Roberto De Almeida
Access level	Public
Type	Mercurial
Language	Python
Last updated	2013-05-08
Created	2011-04-10
Size	145.6 KB (download)

JSONStore

JSONStore is a lightweight database for JSON documents exposed through HTTP. It can be used to create, manage and search documents through a REST API (as a "MongoDB lite", as it's been called) or using the Python API (as Pickle on steroids). The REST interface has support for GET, POST, PUT, DELETE and PATCH (using [jsonpatch](#)) methods, e-tags, webhooks, conditional PUT, JSONP, [Rison](#) and [JSON pointer](#).

Getting started with HTTP API

You can get started really quick:

```
$ pip install jsonstore
$ jsonstore
* Running on http://127.0.0.1:31415/
```

JSONStore is now happily listening on the loopback interface. You can specify a different host and port combination; check `jsonstore -h` for help. Also, since JSONStore is a simple WSGI application it can be deployed in a thousand different ways; the Flask website has a nice list of [deployment options for WSGI apps](#) if you want to run it in production.

Adding some data

Now that we have a database running, how do we add some data? Well, JSONStore speaks HTTP all the way down. In order to interact with the database we need a client that can handle HTTP and its different verbs: GET, POST, PUT, PATCH and DELETE. In these examples we'll be using the Unix utility `curl` from the command line.

In order to add a document to the database we need to POST a JSON object:

```
$ curl -v http://127.0.0.1:31415/ -d '{"name":"Roberto","age":34}'
...
< Location: http://127.0.0.1:31415/da6360da-a627-461f-870d-08593569c21e
...
{
  "age": 34,
  "name": "Roberto",
  "__updated__": "2012-05-09T16:17:15.587406+00:00",
  "__id__": "da6360da-a627-461f-870d-08593569c21e"
}
```

Note here that our document received two additional attributes: an `__id__` and an `__updated__` timestamp. These are automatically added if missing in the document, and we can override them with any values that we want. The server also responds with an HTTP header containing the location of our new document, which is simply the base URL plus the `__id__`. We can introspect our document at that location:

```
$ curl http://127.0.0.1:31415/da6360da-a627-461f-870d-08593569c21e
{
  "age": 34,
  "name": "Roberto",
  "__updated__": "2012-05-09T16:17:15.587406+00:00",
  "__id__": "da6360da-a627-461f-870d-08593569c21e"
}
```

One of the advantages of JSONStore is that it's schema-free: instead of being defined in the database layer schemas are defined in the application logic (this may be a disadvantage, depending on your application). So we can add different types of documents at any time; here's a document representing a pet:

```
$ curl http://127.0.0.1:31415/ -d '{"name":"Minhoca","type":"pet","age":4}'
{
  "age": 4,
  "type": "pet",
  "name": "Minhoca",
  "__updated__": "2012-05-09T16:21:33.382131+00:00",
  "__id__": "0f917800-74c7-420f-a030-72e74a21ceda"
}
```

One important thing is that, even though in these examples we're creating simple flat documents, JSONStore has support for arbitrary JSON objects. Your documents can have many levels of nesting.

Updating data

How do we update a document? Updates are performed by PUTting a new document to its canonical URL:

```
$ curl -X PUT http://127.0.0.1:31415/da6360da-a627-461f-870d-08593569c21e -d '{"name":
{
  "age": 34,
  "type": "human",
  "name": "Roberto",
  "__updated__": "2012-05-09T16:28:23.084198+00:00",
  "__id__": "da6360da-a627-461f-870d-08593569c21e"
}
```

This will simply overwrite the old document at that location with a new one. Note that `__updated__` has been, well, updated. PUTting the whole document can be a bit tedious, so we can simply [PATCH](#) it:

```
$ curl -X PATCH http://127.0.0.1:31415/da6360da-a627-461f-870d-08593569c21e -d '[{"pat
{
  "age": 34,
  "type": "human",
  "name": "Roberto De Almeida",
  "__updated__": "2012-05-09T16:28:23.084198+00:00",
  "__id__": "da6360da-a627-461f-870d-08593569c21e"
}
```

In this case we submitted a JSON fragment that the server applied to the document.

Conditional PUT

Imagine that we have multiple clients accessing the store at the same time. Bad things could happen if one of them tried to update a document, and in the meantime another one updated it first. This can be easily fixed using conditional updating.

Let's create a counter in the store:

```
$ curl http://127.0.0.1:31415/ -d '{"count":0}'
{
  "count": 0,
  "__id__": "1e3be179-d444-4e2b-8b24-222de28f205f",
  "__updated__": "2012-05-09T16:38:01.379461+00:00"
}
```

Now, remember when I said it was HTTP all the way down? The canonical way in HTTP to perform an action unless a resource has changed is using its etag and issuing an `If-Match` header. The way this works is, the client sends a hash of the original document (the etag) and an HTTP header that says "do this action as long as the hash of the document is still this".

The etag of a document can be found when we access its URL. Let's look at the headers when we access our counter:

```
$ curl -v http://127.0.0.1:31415/1e3be179-d444-4e2b-8b24-222de28f205f
* About to connect() to 127.0.0.1 port 31415 (#0)
* Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 31415 (#0)
```

```
> GET /1e3be179-d444-4e2b-8b24-222de28f205f HTTP/1.1
> User-Agent: curl/7.21.0 (x86_64-pc-linux-gnu) libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2
> Host: 127.0.0.1:31415
> Accept: */*
>
127.0.0.1 - - [09/May/2012 16:39:46] "GET /1e3be179-d444-4e2b-8b24-222de28f205f HTTP/1
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< X-ITEMS: 1
< etag: "cc7b6ec0caa876e33338f1840aabacf8dc3949ce"
< Content-Length: 127
< Server: Werkzeug/0.8.3 Python/2.6.6
< Date: Wed, 09 May 2012 16:39:46 GMT
<
{
  "count": 0,
  "__id__": "1e3be179-d444-4e2b-8b24-222de28f205f",
  "__updated__": "2012-05-09T16:38:01.379461+00:00"
}
* Closing connection #0
```

In this example our counter is set to 0, and its etag is

`cc7b6ec0caa876e33338f1840aabacf8dc3949ce`. So this is how we update the counter as long as it has not been changed:

```
$ curl -X PUT -H 'If-Match: "cc7b6ec0caa876e33338f1840aabacf8dc3949ce"' \
> http://127.0.0.1:31415/1e3be179-d444-4e2b-8b24-222de28f205f -d '{"count":1}'
{
  "count": 1,
  "__id__": "1e3be179-d444-4e2b-8b24-222de28f205f",
  "__updated__": "2012-05-09T16:48:23.522425+00:00"
}
```

Our counter has now been increased by one, to the new value 1. If another client tried to increase the counter in the meantime, setting its value (wrongly) to 1, this is what would happen:

```
$ curl -v -X PUT -H 'If-Match: "cc7b6ec0caa876e33338f1840aabacf8dc3949ce"' \
> http://127.0.0.1:31415/1e3be179-d444-4e2b-8b24-222de28f205f -d '{"count":1}'
< ...
< HTTP/1.0 412 Precondition Failed
```

So the server returns the status code `412 Precondition Failed` which means that the precondition specified by the client is not valid. In this case the client should request the new version of the document and try again.

Deleting

Entries can be deleted by performing a DELETE request to their URL:

```
curl -X DELETE http://127.0.0.1:31415/1e3be179-d444-4e2b-8b24-222de28f205f
```

The server will return an empty response if everything went ok, and the status code will be `204 No Content`.

Searching

Ok, we know how to put data *in*, now how do we get data *out*? JSONStore implements a very simple search algorithm based on *fragment matching* for performing searches. (I'm not sure if that term really exists.) Here's how we would query the database for all documents representing pets:

```
$ curl -g 'http://127.0.0.1:31415/{"type":"pet"}'
[
  {
    "age": 4,
    "type": "pet",
    "name": "Minhoca",
    "__updated__": "2012-05-09T16:21:33.382131+00:00",
    "__id__": "0f917800-74c7-420f-a030-72e74a21ceda"
  }
]
```

(Note that the `-g` option is required for `curl` so it doesn't try to interpret the braces as a list.)

What happened here? In this example we passed a JSON document to the root URL, instead of

an id. By doing this the store will return a list of all documents that match that fragment. Here the list has a single document, representing my pet Minhoca. Now, I agree that this is quite limiting, so we can also use operators in the match to make more interesting queries:

```
$ curl -g 'http://127.0.0.1:31415/{\"age\": \"GreaterThan(10)\"}'  
[  
  {  
    \"age\": 34,  
    \"type\": \"human\",  
    \"name\": \"Roberto De Almeida\",  
    \"__updated__\": \"2012-05-09T16:28:23.084198+00:00\",  
    \"__id__\": \"da6360da-a627-461f-870d-08593569c21e\"  
  }  
]
```

The list of available operators includes:

```
>>> from jsonstore import operators  
>>> print operators.__all__  
['Operator', 'Equal', 'NotEqual', 'GreaterThan', 'LessThan', 'GreaterEqual', 'LessEqual']
```

Operators are way cooler when used in the Python API, because there they are actual objects and not strings. When encoding them as JSON we need to have them inside a string, unfortunately.

Note again that in the same way as documents can include more complex structures, so can the search pattern.

JSONp

Sometimes we will need to access JSONStore from Javascript in a different domain, and we will hit the [same domain policy](#). A quick solution for this problem is using JSONp, or "JSON with padding". If we specify a query parameter called `jsonp` or `callback` with the name of a function, the store will return Javascript code that can be inserted dynamically into the `<head>` of the document. Here's an example:

```
$ curl -g 'http://127.0.0.1:31415/{\"age\": \"GreaterThan(10)\"}?jsonp=myfunc'  
myfunc([  
  {  
    \"age\": 34,  
    \"type\": \"human\",  
    \"name\": \"Roberto De Almeida\",  
    \"__updated__\": \"2012-05-09T16:28:23.084198+00:00\",  
    \"__id__\": \"da6360da-a627-461f-870d-08593569c21e\"  
  }  
)
```

Notice that instead of returning a JSON document the database now returns Javascript code that calls the function `myfunc`, passing the document to it. This is how the client would use it:

```
myfunc(entry) {  
  // do stuff with the document  
}  
  
var script = document.createElement('script');  
script.src = 'http://127.0.0.1:31415/{\"age\": \"GreaterThan(10)\"}?jsonp=myfunc';  
document.getElementsByTagName('head')[0].appendChild(script);
```

Webhooks

JSONStore has a simple mechanism based on webhooks for notifying a client when changes have occurred to a document. Suppose our client wants to monitor changes to documents that match a specific pattern, say, `{\"type\": \"component\"}`. All it has to do is create a document like this on the store:

```
{  
  \"type\": \"__webhook__\",  
  \"pattern\": {  
    \"type\": \"component\"  
  },  
  \"url\": \"http://client.example.com/\"  
}
```

With this document, the store will perform a GET on `http://client.example.com/` everytime a

document that matches that pattern is created, modified or deleted. Note that operators can also be used in the pattern.

The Python API

JSONStore also comes with a Python API that can be used to interact with a remote store via HTTP. The first step is defining an entry manager by passing the URL of the store:

```
>>> from jsonstore.client import EntryManager
>>> em = EntryManager('http://127.0.0.1:31415/')
```

We can now search the store by passing a JSON-like structure, keyword arguments, or a combination of both:

```
>>> em.search({"name": "Minhoca"})
[{u'age': 4, u'type': u'pet', u'name': u'Minhoca', u'__updated__': datetime.datetime(2014, 12, 1, 15, 15, 15)}]

>>> from jsonstore.operators import Like
>>> em.search(name=Like('Roberto%'))
[{u'age': 34, u'type': u'human', u'name': u'Roberto De Almeida', u'__updated__': datetime.datetime(2014, 12, 1, 15, 15, 15)}]
```

Note that like when we searched via `curl` the response is a list of documents that match the pattern we passed. Also note that the `__updated__` attribute is automatically converted to a `datetime` object.

In order to add a new document we use the `create` method:

```
>>> em.create(name='JSONStore', url='http://code.dealmeida.net/jsonstore')
{'url': 'http://code.dealmeida.net/jsonstore', 'name': 'JSONStore', '__updated__': datetime.datetime(2014, 12, 1, 15, 15, 15)}
```

The method will return the newly created entry, so we have access to the `__updated__` and `__id__` attributes.

When updating a new document, we should pass the old one if we want to do a conditional update:

```
>>> old = em.search(name=Like('Roberto%'))[0]
>>> new = old.copy()
>>> new['interests'] = ["Python"]
>>> em.update(new, old)
{'interests': ['Python'], 'name': 'Roberto De Almeida', 'age': 34, '__updated__': datetime.datetime(2014, 12, 1, 15, 15, 15)}
```

Finally, documents are deleted by their id, just like in HTTP:

```
>>> em.delete('da6360da-a627-461f-870d-08593569c21e')
```

The Local API

The same Python API can be used to interact with a local storage. This is useful if we need a small database for our Python application. For example:

```
>>> from jsonstore.store import EntryManager
>>> em = EntryManager('index.db')
>>> em.create(type='foo', bar='baz')
{'u'type': u'foo', u'bar': u'baz', u'__id__': u'239f5b84-ad1a-48c0-99be-cea216d0bd45', u'__updated__': datetime.datetime(2014, 12, 1, 15, 15, 15)}

>>> from jsonstore.operators import In
>>> em.search(bar=In('baz', 'bez', 'biz'))
[{u'type': u'foo', u'bar': u'baz', u'__id__': u'239f5b84-ad1a-48c0-99be-cea216d0bd45', u'__updated__': datetime.datetime(2014, 12, 1, 15, 15, 15)}]
```

The DSL

If you're using JSONStore from Python you can also play with its DSL:

```
>>> from jsonstore.dsl import Store
>>> from jsonstore.operators import *
>>> remote = Store("http://localhost:31415/")
>>> local = Store("backup.db")
```

Let's add some data:











```
>>> {"name": "John", "followers": 10} >> remote
>>> {"name": "Paul", "followers": 2} >> remote
```

```
>>> print remote
[{u'followers': 2, u'name': u'Paul', u'__updated__': datetime.datetime(2013, 5, 2, 4, 46
```

And move data from one store to the other, filtering it:

```
>>> remote | {"followers": GreaterThan(5)} | local
>>> print local
[{u'followers': 10, u'name': u'John', u'__updated__': datetime.datetime(2013, 5, 2, 4,
```

Recent activity

-  [Roberto De Almeida](#) pushed 1 commit to [robertodealmeida/JSONStore](#)
2013-05-08
[ef79f98](#) - Allow REGEXP searches
-  [Roberto De Almeida](#) pushed 1 commit to [robertodealmeida/JSONStore](#)
2013-05-02
[3195bfb](#) - Fix docs
-  [Roberto De Almeida](#) pushed 1 commit to [robertodealmeida/JSONStore](#)
2013-05-02
[3a5e4f2](#) - Update docs
-  [Roberto De Almeida](#) pushed 1 commit to [robertodealmeida/JSONStore](#)
2013-05-01
[1a9b7b9](#) - Implement JSON pointer in the URL
-  [Roberto De Almeida](#) reported [issue #2](#) to [robertodealmeida/JSONStore](#)
2013-05-01
Implement JSON pointer
-  [Roberto De Almeida](#) updated [issue #1](#) in [robertodealmeida/JSONStore](#)
2013-05-01
-  [Roberto De Almeida](#) pushed 1 commit to [robertodealmeida/JSONStore](#)
2013-05-01
[dbd4f22](#) - Updated PATCH, now uses jsonpatch
-  [nathants](#) began watching [robertodealmeida/JSONStore](#)
2013-04-15
-  [Simon Belluzzo](#) began watching [robertodealmeida/JSONStore](#)
2013-04-14
-  [Troy Huang](#) began watching [robertodealmeida/JSONStore](#)
2013-04-09