
pyxs Documentation

Release 0.3.1

Sergei Lebedev, Fedor Gogolev

January 12, 2016

1	pyxs	1
2	Tutorial	3
2.1	Basics	3
2.2	Transactions	3
2.3	Events	4
2.4	Y U SO LAZY DAWG	4
3	API reference	5
3.1	pyxs.client	5
3.2	pyxs.connection	8
3.3	pyxs.helpers	9
3.4	pyxs.exceptions	9
3.5	pyxs._internal	10
4	pyxs Changelog	11
4.1	Version 0.3.1	11
4.2	Version 0.3	11
4.3	Version 0.2	11
4.4	Version 0.1	12
5	Indices and tables	13
	Python Module Index	15

Pure Python bindings for communicating with XenStore. Currently two backend options are available:

- over a Unix socket with *UnixSocketConnection*;
- over *XenBus* with *XenBusConnection*.

Which backend is used is determined by the arguments used for *Client* initialization, for example the following code creates a *Client* instance, working over a Unix socket:

```
>>> Client(unix_socket_path="/var/run/xenstored/socket")
<pyxs.client.Client object at 0xb74103cc>
>>> Client()
<pyxs.client.Client object at 0xb74109cc>
```

Use `xen_bus_path` argument to initialize a *Client*, communicating with XenStore over *XenBus*:

```
>>> Client(xen_bus_path="/proc/xen/xenbus")
<pyxs.client.Client object at 0xb7410d2c>
```

copyright

3. 2011 by Selectel, see AUTHORS for more details.

license LGPL, see LICENSE for more details.

Contents:

2.1 Basics

Using *pyx*s is easy! the only class you need to import is *Client* (unless you're up to some spooky stuff) – it provides a simple straightforward API to XenStore content with a bit of Python's syntactic sugar here and there. Generally, if you just need to fetch or update some XenStore items you can do:

```
>>> from pyxs import Client
>>> with Client() as c:
...     c["/local/domain/0/name"] = "Domain0"
...     c["/local/domain/0/name"]
'Domain0'
```

Note: Even though *Client* does support `dict()`-like lookups, they have nothing else in common – for instance there's no `Client.get()` currently.

2.2 Transactions

If you're already familiar with XenStore features, you probably know that it has basic transaction support. Transactions allow you to operate on a separate, isolated copy of XenStore tree and merge your changes back atomically on commit. Keep in mind, however, that **changes made inside a transaction aren't available to other XenStore clients unless you commit them**. Here's an example:

```
>>> c = Client()
>>> with c.transaction() as t:
...     t["/foo/bar"] = "baz"
...     t.transaction_end(commit=True)
...
>>> c["/foo/bar"]
'baz'
```

The second line inside `with` statement is completely optional, since the default behaviour is to commit everything on context manager exit. You can also abort the current transaction by calling `transaction_end()` with `commit=False`.

2.3 Events

When a new path is created or an existing path is modified, XenStore fires an event, notifying all watchers that a change has been made. To watch a path, you have to call `watch()` with a path you want to watch and a token, unique for that path within the active transaction. After that, incoming events can be fetched by calling `wait()`:

```
>>> with Client() as c:
...     c.watch("/foo/bar", "a unique token")
...     c.wait()
Event("/foo/bar", "a unique token")
```

XenStore also has a notion of *special* paths, which are reserved for special occasions:

Path	Description
@intro- duceDo- main	Fired when a new domain is introduced to XenStore – you can also introduce domains yourself with a <code>introduce_domain()</code> call, but in most of the cases, <code>xenstored</code> will do that for you.
@release- Domain	Fired when XenStore is no longer communicating with a domain, see <code>release_domain()</code> .

Events for both *special* and ordinary paths are simple two element tuples, where the first element is always *event target* – a path which triggered the event and second is a token, you’ve passed to `watch()`. A nasty consequence of this is that you can’t get *domid* of the domain, which triggered `@introduceDomain` or `@releaseDomain` from the received event.

2.4 Y U SO LAZY DAWG

`pyxs` also provides a compatibility interface, which copies the ones of `xen.lowlevel.xs` – so you don’t have to change **anything** in the code to switch to `pyxs`:

```
>>> from pyxs import xs
>>> xs = xs()
>>> xs.read(0, "/local/domain/0/name")
'Domain0'
```

API reference

3.1 pyxs.client

This module implements XenStore client, which uses multiple connection options for communication: `connection.UnixSocketConnection` and `connection.XenBusConnection`. Note however, that the latter one can be a bit buggy, when dealing with `WATCH_EVENT` packets, so using `connection.UnixSocketConnection` is preferable.

copyright

3. 2011 by Selectel, see AUTHORS for more details.

license LGPL, see LICENSE for more details.

class `pyxs.client.Client` (`unix_socket_path=None`, `socket_timeout=None`, `xen_bus_path=None`, `connection=None`, `transaction=None`)
 XenStore client – <useful comment>.

Parameters

- **xen_bus_path** (*str*) – path to XenBus device, implies that `XenBusConnection` is used as a backend.
- **unix_socket_path** (*str*) – path to XenStore Unix domain socket, usually something like `/var/run/xenstored/socket` – implies that `UnixSocketConnection` is used as a backend.
- **socket_timeout** (*float*) – see `settimeout()` for details.
- **transaction** (*bool*) – if `True` `transaction_start()` will be issued right after connection is established.

Note: `UnixSocketConnection` is used as a fallback value, if backend cannot be determined from arguments given.

Here's a quick example:

```
>>> with Client() as c:
...     c.write("/foo/bar", "baz")
...     c.read("/foo/bar")
'OK'
'baz'
```

read (*path*, *default=None*)
 Reads data from a given path.

Parameters

- **path** (*str*) – a path to read from.
- **default** (*str*) – default value, to be used if *path* doesn't exist.

write (*path*, *value*)

Writes data to a given path.

Parameters

- **value** – data to write (can be of any type, but will be coerced to `bytes()` eventually).
- **path** (*str*) – a path to write to.

mkdir (*path*)Ensures that a given path exists, by creating it and any missing parents with empty values. If *path* or any parent already exist, its value is left unchanged.**Parameters** **path** (*str*) – path to directory to create.**rm** (*path*)Ensures that a given does not exist, by deleting it and all of its children. It is not an error if *path* doesn't exist, but it is an error if *path*'s immediate parent does not exist either.**Parameters** **path** (*str*) – path to directory to remove.**ls** (*path*)Returns a list of names of the immediate children of *path*.**Parameters** **path** (*str*) – path to list.**get_permissions** (*path*)Returns a list of permissions for a given *path*, see [InvalidPermission](#) for details on permission format.**Parameters** **path** (*str*) – path to get permissions for.**set_permissions** (*path*, *perms*)Sets a access permissions for a given *path*, see [InvalidPermission](#) for details on permission format.**Parameters**

- **path** (*str*) – path to set permissions for.
- **perms** (*list*) – a list of permissions to set.

walk (*top*, *topdown=True*)Walk XenStore, yielding 3-tuples (*path*, *value*, *children*) for each node in the tree, rooted at node *top*.**Parameters**

- **top** (*str*) – node to start from.
- **topdown** (*bool*) – see `os.walk()` for details.

get_domain_path (*domid*)Returns the domain's base path, as is used for relative transactions: ex: `"/local/domain/<domid>"`. If a given *domid* doesn't exists the answer is undefined.**Parameters** **domid** (*int*) – domain to get base path for.**is_domain_introduced** (*domid*)Returns `True` if `xenstored` is in communication with the domain; that is when `INTRODUCE` for the domain has not yet been followed by domain destruction or explicit `RELEASE`; and `False` otherwise.

Parameters `domid` (*int*) – domain to check status for.

introduce_domain (*domid*, *mfn*, *eventchn*)

Tells `xenstored` to communicate with this domain.

Parameters

- `domid` (*int*) – a real domain id, (0 is forbidden).
- `mfn` (*long*) – address of xenstore page in *domid*.
- `eventchn` (*int*) – an unbound event channel in *domid*.

release_domain (*domid*)

Manually requests `xenstored` to disconnect from the domain.

Parameters `domid` (*int*) – domain to disconnect.

Note: `xenstored` will in any case detect domain destruction and disconnect by itself.

resume_domain (*domid*)

Tells `xenstored` to clear its shutdown flag for a domain. This ensures that a subsequent shutdown will fire the appropriate watches.

Parameters `domid` (*int*) – domain to resume.

set_target (*domid*, *target*)

Tells `xenstored` that a domain is targetting another one, so it should let it tinker with it. This grants domain *domid* full access to paths owned by *target*. Domain *domid* also inherits all permissions granted to *target* on all other paths.

Parameters

- `domid` (*int*) – domain to set target for.
- `target` (*int*) – target domain (yours truly, Captain).

transaction_start ()

Starts a new transaction and returns transaction handle, which is simply an int.

Warning: Currently `xenstored` has a bug that after 2^{32} transactions it will allocate id 0 for an actual transaction.

transaction_end (*commit=True*)

End a transaction currently in progress; if no transaction is running no command is sent to XenStore.

monitor ()

Returns a new *Monitor* instance, which is currently *the only way* of doing PUBSUB.

transaction ()

Returns a new *Client* instance, operating within a new transaction; can only be used only when no transaction is running. Here's an example:

```
>>> with Client().transaction() as t:
...     t.do_something()
...     t.transaction_end(commit=True)
```

However, the last line is completely optional, since the default behaviour is to commit everything on context manager exit.

Raises `pyxs.exceptions.PyXSError` if this client is linked to and active transaction.

class `pyxs.client.Monitor` (*connection*)

XenStore monitor – allows minimal PUBSUB-like functionality on top of XenStore.

```
>>> m = Client().monitor()
>>> m.watch("foo/bar")
>>> m.wait()
Event(...)
```

watch (*wpath*, *token*)

Adds a watch.

When a *path* is modified (including path creation, removal, contents change or permissions change) this generates an event on the changed *path*. Changes made in transactions cause an event only if and when committed.

Parameters

- **wpath** (*str*) – path to watch.
- **token** (*str*) – watch token, returned in watch notification.

unwatch (*wpath*, *token*)

Removes a previously added watch.

Parameters

- **wpath** (*str*) – path to unwatch.
- **token** (*str*) – watch token, passed to `watch()`.

wait (*sleep=None*)

Waits for any of the watched paths to generate an event, which is a (*path*, *token*) pair, where the first element is event path, i.e. the actual path that was modified and second element is a token, passed to the `watch()`.

Parameters **sleep** (*float*) – number of seconds to sleep between event checks.

3.2 pyxs.connection

This module implements two connection backends for `Client`.

copyright

3. 2011 by Selectel, see AUTHORS for more details.

license LGPL, see LICENSE for more details.

class `pyxs.connection.UnixSocketConnection` (*path=None*, *socket_timeout=None*)

XenStore connection through Unix domain socket.

Parameters

- **path** (*str*) – path to XenStore unix domain socket, if not provided explicitly is restored from process environment – similar to what `libxs` does.
- **socket_timeout** (*float*) – see `settimeout()` for details.

class `pyxs.connection.XenBusConnection` (*path=None*)

XenStore connection through XenBus.

Parameters **path** (*str*) – path to XenBus block device; a predefined OS-specific constant is used, if a value isn't provided explicitly.

3.3 pyxs.helpers

Implements various helpers.

copyright

3. 2011 by Selectel, see AUTHORS for more details.

license LGPL, see LICENSE for more details.

`pyxs.helpers.error(smith)`

Returns a `PyXSError` matching a given `errno` or error name.

```
>>> error(22)
pyxs.exceptions.PyXSError: (22, 'Invalid argument')
>>> error("EINVAL")
pyxs.exceptions.PyXSError: (22, 'Invalid argument')
```

`pyxs.helpers.validate_path(path)`

Checks if a given path is valid, see [InvalidPath](#) for details.

Parameters `path` (*str*) – path to check.

Raises `pyxs.exceptions.InvalidPath` when path fails to validate.

`pyxs.helpers.validate_watch_path(wpath)`

Checks if a given watch path is valid – it should either be a valid path or a special, starting with @ character.

Parameters `wpath` (*str*) – watch path to check.

Raises `pyxs.exceptions.InvalidPath` when path fails to validate.

`pyxs.helpers.validate_perms(perms)`

Checks if a given list of permission follows the format described in [get_permissions\(\)](#).

Parameters `perms` (*list*) – permissions to check.

Raises `pyxs.exceptions.InvalidPermissions` when any of the permissions fail to validate.

3.4 pyxs.exceptions

This module implements a number of Python exceptions used by `pyxs` classes.

copyright

3. 2011 by Selectel, see AUTHORS for more details.

license LGPL, see LICENSE for more details.

exception `pyxs.exceptions.InvalidOperation`

Exception raised when `Packet` is passed an operation, which isn't listed in [Op](#).

Parameters `operation` (*int*) – invalid operation value.

exception `pyxs.exceptions.InvalidPayload`

Exception raised when `Packet` is initialized with payload, which exceeds 4096 bytes restriction or contains a trailing NULL.

Parameters `operation` (*bytes*) – invalid payload value.

exception `pyxs.exceptions.InvalidPath`

Exception raised when a path processed by a comand doesn't match the following constraints:

- its length should not exceed 3072 or 2048 for absolute and relative path respectively.
- it should only consist of ASCII alphanumerics and the four punctuation characters `-/_@` – *hyphen, slash, underscore* and *atsign*.
- it shouldn't have a trailing `/`, except for the root path.

Parameters `path` (*bytes*) – invalid path value.

exception `pyxs.exceptions.InvalidPermission`

Exception raised for permission which don't match the following format:

<code>w<domid></code>	write only
<code>r<domid></code>	read only
<code>b<domid></code>	both read and write
<code>n<domid></code>	no access

Parameters `perm` (*bytes*) – invalid permission value.

exception `pyxs.exceptions.ConnectionError`

Exception raised for failures during socket operations.

exception `pyxs.exceptions.UnexpectedPacket`

Exception raised when recieved packet header doesn't match the header of the packet sent, for example if outgoing packet has `op = Op.READ` the incoming packet is expected to have `op = Op.READ` as well.

3.5 pyxs._internal

A place for secret stuff, not available in the public API.

copyright

3. 2011 by Selectel, see AUTHORS for more details.

license LGPL, see LICENSE for more details.

`pyxs._internal.Op = Operations(DEBUG=0, DIRECTORY=1, READ=2, GET_PERMS=3, WATCH=4, UNWATCH=5, TR`

Operations supported by XenStore.

class `pyxs._internal.Packet`

A single message to or from XenStore.

Parameters

- `op` (*int*) – an item from `Op`, representing operation, performed by this packet.
- `payload` (*bytes*) – packet payload, should be a valid ASCII-string with characters between `[0x20; 0x7f]`.
- `rq_id` (*int*) – request id – hopefully a **unique** identifier for this packet, XenStore simply echoes this value back in reponse.
- `tx_id` (*int*) – transaction id, defaults to 0 – which means no transaction is running.

pyxs Changelog

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

4.1 Version 0.3.1

Released on November 29th 2012

- Added *default* argument to *Client.read()*, which acts similar to *dict.get()*.
- Fixed a lot of minor quirks so *pyxs* can be Debianized.

4.2 Version 0.3

Released on September 12th 2011

- Moved all PUBSUB functionality into a separate *Monitor* class, which uses a *separate* connection. That way, we'll never have to worry about mixing incoming XenStore events and command replies.
- Fixed a couple of nasty bugs in concurrent use of *Client.wait()* with other commands (see above).

4.3 Version 0.2

Released on August 18th 2011

- Completely refactored validation – no more *@spec* magic, everything is checked explicitly inside *Client.execute_command()*.
- Added a compatibility interface, which mimics *xen.lowlevel.xs* behaviour, using *pyxs* as a backend, see *pyxs/_compat.py*.
- Restricted *SET_TARGET*, *INTRODUCE* and *RELEASE* operations to Dom0 only – */proc/xen/capabilities* is used to check domain role.
- Fixed a bug in *Client.wait()* – queued watch events weren't wrapped in *pyxs._internal.Event* class, unlike the received ones.
- Added *Client.walk()* method for walking XenStore tree – similar to *os.walk()*

4.4 Version 0.1

Initial release, released on July 16th 2011

- Added a complete implementation of XenStore protocol, including transactions and path watching, see *pyxs.Client* for details.
- Added generic validation helper – *@spec*, which forces arguments to match the scheme from the wire protocol specification.
- Added two connection backends – *XenBusConnection* for connecting from DomU through a block device and *UnixSocketConnection*, communicating with `xenstored` via a Unix domain socket.

Indices and tables

- `genindex`
- `modindex`
- `search`

p

- `pyx`, 1
- `pyx.__internal`, 10
- `pyx.client`, 5
- `pyx.connection`, 8
- `pyx.exceptions`, 9
- `pyx.helpers`, 8

C

Client (class in pyxs.client), 5

ConnectionError, 10

E

error() (in module pyxs.helpers), 9

G

get_domain_path() (pyxs.client.Client method), 6

get_permissions() (pyxs.client.Client method), 6

I

introduce_domain() (pyxs.client.Client method), 7

InvalidOperation, 9

InvalidPath, 9

InvalidPayload, 9

InvalidPermission, 10

is_domain_introduced() (pyxs.client.Client method), 6

L

ls() (pyxs.client.Client method), 6

M

mkdir() (pyxs.client.Client method), 6

Monitor (class in pyxs.client), 7

monitor() (pyxs.client.Client method), 7

O

Op (in module pyxs._internal), 10

P

Packet (class in pyxs._internal), 10

pyxs (module), 1

pyxs._internal (module), 10

pyxs.client (module), 5

pyxs.connection (module), 8

pyxs.exceptions (module), 9

pyxs.helpers (module), 8

R

read() (pyxs.client.Client method), 5

release_domain() (pyxs.client.Client method), 7

resume_domain() (pyxs.client.Client method), 7

rm() (pyxs.client.Client method), 6

S

set_permissions() (pyxs.client.Client method), 6

set_target() (pyxs.client.Client method), 7

T

transaction() (pyxs.client.Client method), 7

transaction_end() (pyxs.client.Client method), 7

transaction_start() (pyxs.client.Client method), 7

U

UnexpectedPacket, 10

UnixSocketConnection (class in pyxs.connection), 8

unwatch() (pyxs.client.Monitor method), 8

V

validate_path() (in module pyxs.helpers), 9

validate_perms() (in module pyxs.helpers), 9

validate_watch_path() (in module pyxs.helpers), 9

W

wait() (pyxs.client.Monitor method), 8

walk() (pyxs.client.Client method), 6

watch() (pyxs.client.Monitor method), 8

write() (pyxs.client.Client method), 6

X

XenBusConnection (class in pyxs.connection), 8