xal Documentation

Release 0.1dev

Benoît Bryon

March 04, 2013

CONTENTS

"xal" is a contextual execution framework for Python.

> **Warning:** This project is experimental. Current goal is to implement a proof-of-concept, so that the project can be shown and discussed with community and teams from other projects such as Fabric, Salt or zc.buildout.
> **Proposals focus about API and usage**, implementation comes next.
> At this early stage, even the name "xal" may be changed.

Xal helps you create scripts to manage resources or perform actions on a system, independantly from the execution context:

```
>>> def home_directory_exists(session):
...     """Return True if home directory of session's user exists.
...
...     ``session`` argument is the execution context.
...
...     """
...     return session.dir.exists(session.dir.home)
```

Then create an execution session:

> **Warning:** Notice that most of the configuration stuff below is supposed to be done via auto-configuration tools, or via pre-configured sessions. In future releases required code should be reduced to something like this:
>
> ```
> import xal
> session = xal.local()
> ```

```
>>> # Create a session with its registry.
>>> from xal.session import LocalSession
>>> from xal.registry import Registry
>>> session = LocalSession(registry=Registry())
>>> # The registry contains a mapping of providers attached to interfaces.
>>> # So, let's import some providers...
>>> from xal.sys.local import LocalSysProvider
>>> from xal.client.local import LocalClient
>>> from xal.dir.local import LocalDirProvider
>>> # ... and map the providers to interfaces.
>>> session.registry.register(sys=LocalSysProvider(),
...                           client=LocalClient(),
...                           dir=LocalDirProvider())
```

Finally run your scripts in the session:

```
>>> home_directory_exists(session)
True
```

Execution sessions can be local, remote, use Fabric, Salt... The main motivation of this library is to provide a common set of tools for sysadmin scripts.

Main use case are:

- deployment, provisioning;
- remote execution;
- portable execution.

CHAPTER

ONE

RESSOURCES

- code repository: https://github.com/benoitbryon/xal
- bugtracker: https://github.com/benoitbryon/xal/issues

CHAPTER

TWO

CONTENTS

2.1 Proof of concept

Here are proof-of-concept examples of contextual execution.

> **Warning:** The following features are not currently available! This section only illustrates possibilities.

2.1.1 Hello world!

Hello world:

```
from xal.session import Session
from xal.client import LocalClient
from xal.cmd.local import LocalCmdProvider

session = Session(client=LocalClient)
session.registry.register('cmd', LocalCmdProvider())

session.cmd.echo(u'Hello world!')
```

Defaults...

```
from xal import Session

context = Session()
context.dir.create(context.file.join(context.dir.home(), 'foo', 'bar'))
context.package.install('postgresql')
print context.postgresql.version
context.postgresql.start()
context.service.stop(postgresql)
```

Basic customization...

```
from xal import Session

context = Session(sudoer=True)  # Will execute commands with "sudo" or equivalent.
context.package.install('postgresql')

remote = Session('foo@example.com', identity_file=context.file.join(context.ssh.cfg_dir, 'id_foo'))
assert(remote.user.is_sudoer)
remote.package.install('postgresql', version='8.2')
```

Using Fabric:

```
from xal import FabricSession
from fabric import api as fab_api

@task
def install_postgresql():
    """Install PostgreSQL server."""
    session = FabricSession(env=fab_api.env)
    session.package.install('postgresql')
```

In a buildout recipe:

```
from xal import BuildoutSession

class PackageRecipe(object):
    def __init__(self, buildout, name, options):
        self.session = BuildoutSession(buildout)
        options['version'] = options['version'] or session.package.VERSION_LATEST
        options['provider'] = session.package.get(options['provider'], None)
        self.name, self.options = name, options

    def install(self):
        self.session.package.install(self.options['package'],
                                     version=self.options['version'],
                                     provider=self.options['provider'])
```

2.1.2 Resources

Use a session to handle resources:

```
from xal.session import LocalSession
session = LocalSession()

dir = session.dir  # Shortcut to the provider.
foobar = dir.create(dir.join(dir.HOME, 'foo', 'bar', recursive=True))
assert(foobar.exists)
assert(foobar.status('chmod') is 0755)
foobar.delete()
```

Call to a provider is a resource factory:

```
from xal.session import LocalSession
session = LocalSession()

foobar = session.dir(path=session.dir.HOME)
assert(foobar.exists)
assert(foobar.status('exists'))
foobar.chmod('0755')
```

Create a generic resource then use with any session.

```
# Create a generic resource: home_directory.
from xal.resources import Dir
home_directory = Dir(path=Dir.HOME, mode="755")

from xal.session import LocalSession
session = LocalSession()
```

```
session_home = session.dir.create(home_directory)
# session_home is a resource that belongs to the session.
assert(session_home is not home_directory)
assert(session_home._session is session)
assert(home_directory._session is None)
```

2.1.3 Providers

Available providers:

- directory: filesystem directories
- file: filesystem files
- package: system package
- process
- service
- virtualenv
- buildout
- ...

Providers implement an API.

Automatically plug your providers with entry-points. Or do it manually via session's registry:

```
from xal import LocalSession
session = LocalSession()


class HelloWorldProvider(object):
    def echo(self, who):
        return self._context.command.echo(who)

provider = session.registry.register('hello', HelloWorldProvider())
session.hello.echo('world!')
provider.echo('world!')


def hello(context, who):
     return context.command.echo(who)

session.registry.register('hello', hello)
session.hello('world!')
```

Use a specific provider:

```
pkg = session.package('python-dev', provider=Aptitude)
pkg.install()

session.registry.use(package='foo.bar.providers.Aptitude')
session.package.install('curl')
```

2.1.4 Monitoring

```
from xal import Session

context = Session(sudoer=True)
context.package.status('postgresql')
context.service.status('postgresql')
```

2.2 Installation

This code is open-source. See *License* for details.

If you want to contribute to the code, you should go to *Contributing to the project* documentation.

Install the package with your favorite Python installer. As an example, with pip:

```
# Not released yet on PyPI, so use the source...
pip install https://github.com/benoitbryon/xal.git#egg=xal
```

See *xal* for a detailed usage documentation.

2.3 xal

2.3.1 xal Package

xal Package

provider Module

registry Module

resource Module

session Module

Subpackages

client Package

local Module

provider Module

cmd Package

local Module

provider Module

`resource` Module

dir Package

`generic` Module

`local` Module

`provider` Module

`resource` Module

sys Package

`local` Module

`provider` Module

user Package

`provider` Module

`resource` Module

2.4 About xal

This section is about the project itself.

2.4.1 Vision

XAL is a contextual execution framework for Python. Through its contextual execution system, it makes a resource abstraction layer possible.

Contextual execution

In Python, there are several tools to perform actions on the local system. As an example, the *subprocess* module makes it possible to run arbitrary shell commands. So, let's suppose you wrote a script that echoes "Hello world!" using the shell:

```
import subprocess
subprocess.call(['echo', u'Hello world!'])
```

Then what if you want to run commands on a remote machine? You can write a fabfile:

```
from fabric import api as fab_api

@fab_api.task
def hello():
    fab_api.run(['echo', 'Hello world!'])
```

Then what if you want to run the command as an admin? With Fabric:

```
from fabric import api as fab_api

@fab_api.task
def hello():
    fab_api.sudo(['echo', 'Hello world!'])
```

Then what if you want to run the command as a sudoer, on the local machine? Fabric doesn't provide a wrapper for that:

```
from fabric import api as fab_api

@fab_api.task
def hello():
    fab_api.local(['sudo', 'echo', 'Hello world!'])
```

Then what if you want to run it on a Windows machine? You'll have to adapt the code.

Then what if you migrate to another deployment tool, such as zc.buildout or Salt? You'll have to change the code.

Even with Fabric, we had to write 3 distinct scripts to be able to face all situations. As developers, we'd like to write only one function, then pass it parameters:

- run on local machine or on remote client;
- run as current user, as admin/root, or maybe as another user.

That's why xal were created: write portable high-level system scripts.

A framework

There are so many commands and so many systems... it would be impossible to support them all. And even if it was, it would be made of tons of code and dependencies. That's the first reason why XAL is a framework:

- focus on the API;
- allow plugins;
- keep framework lightweight.

Fully configurable, no global states

XAL is not designed to use global states. XAL registry is entirely configurable: you can change providers' mapping, configure providers or write custom ones.

However, for convenience, XAL offers some configuration helpers, so that common use cases are covered easily.

2.4.2 Alternatives and related projects

This document presents other projects that provide similar or complementary functionalities. It focuses on differences with xal.

Deployment utilities

- Puppet
- Chef
- salt
- zc.buildout
- fabric
- fabtools
- collective.hostout

Fabric

Fabric is great for performing simple tasks. When you want to perform complex tasks, or when you want to reuse your tasks in several situations, you come to reinvent provisioning tools. You'd better use fabric to run buildout recipes (and write buildout recipes instead of fabric scripts), or use salt, or use monitoring...

Fabtools

Fabtools is a provisioning library for Fabric. One strength is its simplicity. But it's also a drawback: it's limited to Fabric (which itself is limited), there are not so many "recipes".

Salt

Salt is about remote execution, and via remote execution it can perform provisioning.

Salt looks great, but as Chef of Puppet, it's a complete software environment: it uses zeromq, requires a server (master) and clients (minions). I mean, for simple needs, it's overkill.

As a developer, I like my development environment to keep as simple as possible. And I like to isolate my projects from my personal system. I mean I'd better install and run salt server on a VM than on my personal computer. But in the same time, I can't reproduce a complete production environment, i.e. run one VM for salt master, one for the database server, one for the web front-end, one for the shared filesystem... Cloud-computing is not the definitive solution for me, because I often work offline (and I like it).

So... I'd like to have an alternative to Salt for simple architectures... Fabric looks like one. But I currently can't write scripts for both!

I'd like Salt modules (those who execute commands) to be packaged as third-party libraries.

I'd like Salt to have a tiny Python client I could install on my personal computer and use it as a remote-control for the master (kind of Chef's knife, but lighter).

zc.buildout and recipes

Recipes for zc.buildout allow you to configure script execution. A recipe have install(), update() and uninstall() methods. It's truly powerful on the local machine. One strength of zc.buildout is isolation. One limit is that it is not really meant to be run as a sudoer. You can, but it introduces some problems. Running 2 buildouts, one as a sudoer, and another as a normal user, could solve the problem, but then you have to protect yourself against running only one of the two.

I'd like to invoke buildout as a sudoer, then, inside buildout configuration, switch from one "context" to another, i.e. tell execute this recipe as sudoer, this recipe as user "postgres", this one as "myself"...

Another strength of zc.buildout is that it automatically discovers and installs some dependencies, such as extensions and recipes. I guess we can't have an execution manager that implements all resources or providers, and that there would be several candidates for some resources (such as "package"). So it would be great if those dependencies where at least automatically discovered. And whenever possible, automatically installed. I suppose that discovery could be a feature bundled in the project, and installation would be implemented by consumers (i.e. buildout, pip, salt...).

Subprocess and wrappers

- http://pypi.python.org/pypi/commandwrapper
- http://pypi.python.org/pypi/EasyProcess
- http://pypi.python.org/pypi/extcmd
- http://pypi.python.org/pypi/sarge

subprocess

When you want to perform simple things, subprocess is a bit complicated, and you'd like to have a simplified wrapper.

When you want to perform complex or repetitive tasks, you'd better write wrappers for code readability and reusability.

So, imho, in any cases, sharing common wrappers would be useful. That's for the execution part.

About the contextual part, suprocess executes commands on the local system with the current user, current environment... I guess one would appreciate to use the same execution API whatever the target system, user, environment...

Other

os and os.path

os and os.path are really useful in daily use, especially when you are dealing with deployment or sysadmin scripts.

os and os.path provide operating system interfaces. As interfaces, the implementation could vary depending on the environment. It currently depends on local operating system. I guess one would appreciate if it depends on contextual execution environment.

I'm not talking about rewriting os module. I'm talking about a third-party provisioner which provides a higher-level interface, but with respect to contextual execution environment. The implementation for local system should use os.

2.4.3 Why "xal" name?

"xal" stands for "eXecution Abstraction Layer".

2.4.4 License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of wardrobe nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

2.4.5 Authors & contributors

- Benoît Bryon <benoit@marmelune.net>

2.4.6 Changelog

0.1 (unreleased)

- Introduced contextual execution architecture: session, registry, providers, resources, client.
- Proof of concept implementation of Dir resource, with DirProvider interface and LocalDirProvider implementation based on Python standard library.

2.5 Presentations

Here are some presentations (slides) about XAL.

2.5.1 XAL - execution abstration layer

Presentation of XAL proof-of-concept, by Benoit Bryon.

Python for sysadmins

Python is great:

- shell, scripts, provisioners, frameworks...
- runs on almost any system

But...

It's hard to write and share portable scripts

- environment vary: users, packages...
- provisioners are overkill: just want a shell or a simple script
- libraries are divided: fabric, buildout, salt...

Develop to XAL session

Write a script which takes a XAL session as argument:

```
def write_greetings(session):
    """Write 'Hello world!' in 'greetings.txt' file relative to user's home."""
    home = session.user.home
    file_path = session.file.join(home, 'greetings.txt')
    file_resource = session.file(file_path)
    if not file_resource.exists():
        file_resource.write('Hello world!')
```

Fabric

Use it in a fabfile:

```
from fabric.api import task
import write_greetings
import xal

@task
def hello_fabric():
    session = xal.fabric(sudoer=True)
    write_hello_world(session)
```

---

zc.buildout

In a buildout recipe:

```
import write_greetings
import xal

class HelloBuildout(object):
    def __init__(self, buildout, name, options):
        self.session = xal.buildout(buildout, name, options)

    def install(self):
        write_hello_world(self.session)
```

```
    def update(self)
        pass
```

Salt

As a salt module:

```
import write_greetings
import xal

def hello_salt():
    session = xal.salt(__salt__)
    write_greetings(session)
```

Shell

In an interactive shell:

```
import write_greetings
import xal
session = xal.local()
write_greeting(session)
```

Resources

XAL session is a proxy to resources:

- files, directories,
- users,
- processes,
- packages,
- your own customized resources...

Share and reuse scripts!

- Reduced cost of change
- Enhanced collaboration between projects related to deployment

=> What about scripts fabric, buildout, salt... can share on PyPI?

XAL is a proof of concept

- https://github.com/benoitbryon/xal
- feedback is welcome!
- are popular projects interested in?

2.6 Contributing to the project

This document provides guidelines for people who want to contribute to the project.

2.6.1 Create tickets

Please use the bugtracker [1] **before** starting some work:

- check if the bug or feature request has already been filed. It may have been answered too!
- else create a new ticket.
- if you plan to contribute, tell us, so that we are given an opportunity to give feedback as soon as possible.
- Then, in your commit messages, reference the ticket with some `refs #TICKET-ID` syntax.

2.6.2 Fork and branch

- Work in forks and branches.
- Prefix your branch with the ticket ID corresponding to the issue. As an example, if you are working on ticket #23 which is about contribute documentation, name your branch like `23-contribute-doc`.

2.6.3 Setup a development environment

System requirements:

- Python [2] version 2.6 or 2.7, available as `python` command.

  Note: You may use Virtualenv [3] to make sure the active `python` is the right one.

- make and wget to use the provided `Makefile`.

Execute:

```
git clone git@github.com/benoitbryon/xal.git
cd xal/
make develop
```

If you cannot execute the Makefile, read it and adapt the few commands it contains to your needs.

[1] https://github.com/benoitbryon/xal/issues
[2] http://python.org
[3] http://virtualenv.org

2.6.4 The Makefile

A `Makefile` is provided to ease development. Use it to:

- setup the development environment: `make develop`
- update it, as an example, after a pull: `make update`
- run tests: `make test`
- build documentation: `make documentation`

The `Makefile` is intended to be a live reference for the development environment.

2.6.5 Documentation

Follow style guide for Sphinx-based documentations [4] when editing the documentation.

2.6.6 Test and build

Use the Makefile.

2.6.7 References

[4] http://documentation-style-guide-sphinx.readthedocs.org/