Armonic

Release 0.1a

aeiche

October 15, 2014

Contents

Armonic is a state machine system oriented for deployment written in python.

With Armonic you can express the different states of an application or a service and make relations with other services. States can also provide management methods to interact with the service.

Each *Lifecycle* (a state machine) representing an application or a service is written in python. If you know already python it's really easy to get started (See *Lifecycle anatomy*).

Client tools are provided to interact with your collection of Lifecycles.

CHAPTER 1

Architecture

Using CLI or Web frontends you can interact with the smart lib to manage the deployment process. When a deployment is asked lib smart resolve the deployment path across all concerned servers. Then the lib translate the requirements to achieve the deployment to the frontend. The frontend fills manually or automatically the requirements and then ask lib smart to orchestrate the deployment.

CHAPTER 2

Quickstart

To enjoy the full potential of Armonic an XMPP server is needed. Multiple XMPP servers are supported like Ejabberd or Prosody. The setup of Prosody is described in this Quickstart.

2.1 Checking out the project

Clone armonic from the github repository on your system. Create a python virtualenv and install all the dependencies of Armonic without polluting your system with the followind commands:

```
git clone https://github.com/armonic/armonic.git
cd armonic
virtualenv --system-site-packages venv
. venv/bin/activate
python setup.py install
```

2.2 Prosody setup

On your favorite distribution install the *prosody* package. Then configure a virtual host for Armonic in */etc/prosody/prosody.cfg.lua*:

```
VirtualHost "armonic.example.com"
```

In production you need to configure add an entry in your DNS server for armonic.example.com with the IP of the prosody server, but you can also add a line in your */etc/hosts* file to test Armonic on your local machine.

To enable Armonic logs propagation between agents and clients the MUC module must be enabled. Just add the following directive in the configuration:

```
Component "logs.armonic.example.com" "muc"
```

Like the XMPP domain, configure your DNS server or /etc/hosts file.

This is all you need to use the armonic cli clients (*smartonic* and *armocli*). To use the webinterface *warmonic* you also have to enable the bosh and/or websocket modules. The Prosody websocket module can be found in the prosody-modules project at https://code.google.com/p/prosody-modules/.

For the bosh and websocket modules use the following options:

```
cross_domain_bosh = true;
bosh_max_inactivity = 600;

cross_domain_websocket = true;
consider_websocket_secure = true;
```

For testing purposes you can enable the automatic creation of accounts so that the agents running on the servers will register an account on the XMPP server automatically:

```
allow_registration = true;
```

Restart the Prosody service and you are good to go.

2.3 Setting up an admin account

On the XMPP server we need to create an account for the administrator. This account will be used by the clients orchestrate the deployments:

```
prosodyctl adduser master@armonic.example.com
```

2.4 Running the agents

If you have enable the automatic registration of accounts you can run the agent directly. If not, create an account on the XMPP server for the agent.

Run the agent with:

```
armonic-agent-xmpp --jid server_account@armonic.example.com --password server_password -v
```

The agent should be running and be connected to the XMPP server.

2.5 Using armocli

armocli is the low level client for armonic. It allows you to call armonic API methods directly. With this tool we can easily check any agent for its status or available *Lifecyles* (Armonic modules) for example.

For exampe, run:

```
armocli --jid master@armonic.example.com --password master_master list
armocli --jid master@armonic.example.com --password master_master info -J server_account@armonic.exar
armocli --jid master@armonic.example.com --password master_master lifecycle -J server_account@armonic
```

2.6 Running deployments with smartonic

smartonic is a client using lib smart allowing to contact multiple agents to orchestrate a deployment. smartonic is used to call Armonic Provides aka *States* methods. When calling a *Provide* smartonic will build the deployment tree and will resolve any depedencies to complete the deployment. If some some *Requires* are needed smartonic will prompt the user to fill some values.

Two tests modules are available to show how this works. They don't execute any operation on the machine. Run the following command:

```
smartonic --jid master@armonic.example.com --password master_password --manage Website//start
```

The Website module emulates the configuration of a web application. This module has a dependency on the Webserver module which emulates a webserver.

The Webserver dependency is first resolved and some configuration is needed to setup the Webserver. *smartonic* will ask on wich port the Webserver will listen, and where the document root will be setup. Some values are suggested and validated when submitted. Then, the Webserver is deployed. After that some configuration is needed on the Website module and finally the Website is deployed.

CHAPTER 3

Howto

3.1 Write a simple service Lifecycle

We can say that a standard service has usually 4 states:

- Not installed: the packages of the service are not installed
- Installed: the packages of the service are installed
- Configured: the service is configured with configuration files, DB values
- Active: the service is running and is accepting requests

Using already provided States we can express a service Lifecycle easily. For example we create the Lifecycle of the sshd service on a Debian like system:

```
from armonic import Lifecycle, State, Transition
from armonic.states import InitialState, InstallPackagesApt, ActiveWithSystemV


class NotInstalled(InitialState):
    pass


class Installed(InstallPackagesApt):
    packages = ['openssh-server']


class Configured(State):

    def enter(self):
        with open('/etc/ssh/sshd_config', 'a') as f:
            f.write('AllowUsers admin')


class Active(ActiveWithSystemV):
    services = ['sshd']


class SSHServer(Lifecycle):
    initial_state = NotInstalled()
    transitions = [
        Transition(NotInstalled(), Installed()),
        Transition(Installed(), Configured()),
```

```
        Transition(Configured(), Active())
    ]
```

Note: To try this example, see section *Load a new lifecycle*.

In this example we define 4 States for the service. The service Lifecycle defines the possible transitions between the States. In this case the Lifecycle transitions are very simple. From the `NotInstalled` state can go to `Installed` state then to the `Configured` state and finally to the `Active` state.

The `Configured` state defines an enter method. This method is trigered when entering the State (ie: when going from state `Installed` to state `Configured`). In this example it will add a line to the `/etc/ssh/sshd_config` file to restrict ssh connections to the `admin` user.

States like `Installed` or `Active` uses already provided States by Armonic with standard python inheritance. The `armonic.states.InstallPackagesApt` state will make sure the package `openssh-server` is installed using Debian package management tools. The `armonic.states.ActiveWithSystemV` state will verify that the `sshd` service is running using the classic SystemV init system.

3.1.1 Requires

What if we wanted to provide manually configuration values to the service ? Using the `armonic.require.Require` decorator you can define variables that needs to be provided to enter a State. Lets rewrite the `Configured` state to take a users list to be configured in the `AllowUsers` directive:

```
from armonic import State, Require
from armonic.variable import VList, VString


class Configured(State):

    @Require('allowed_users', [VList('users', VString, default=["admin"], required=True)])
    def enter(self, requires):
        users = " ".join(requires.allowed_users.variables().users.values)
        with open('/etc/ssh/sshd_config', 'a') as f:
            f.write('AllowUsers %s' % users)
```

We define that to enter in the `Configured` state we need to provide a list of users in the `allowed_users` `armonic.require.Require`. The list is named `users` and is composed of strings. This `armonic.require.Require` cannot be empty (`required=True`) and has a default value (`default=["admin"]`).

Note: Since the `enter` method has now a require you need add `requires` to the `enter` arguments.

A `armonic.require.Require` can be composed of multiple variables. In our case it is only composed of a `armonic.variable.VList`.

Check the complete documentation about *Requires*.

3.1.2 Variables

Variables of the `armonic.require.Require` are also python classes provided by Armonic. This allows to create our own variables with custom validation. For example we could verify that each user provided in the list actually exist on the system. We can do that by simply inherit the `armonic.variable.VString` class and override the validate method:

```
from armonic.variable import VString
from armonic.common import ValidationError
from armonic.utils import grep


class SystemUser(VString):

    def validate(self, value):
        if not grep('/etc/passwd', value):
            raise ValidationError("The user %s doesn't exists on the system" % value)
        return True
```

Then it would be sufficient to change the `armonic.require.Require` declaration to have a custom validation on the user list:

```
@Require('allowed_users', [VList('users', SystemUser, default=["admin"], required=True)])
```

Armonic provides the following base Variable classes: `armonic.variable.VString`, `armonic.variable.VInt`, `armonic.variable.VFloat`, `armonic.variable.VBool`, `armonic.variable.VList`.

Check the complete documentation about *Variables*.

3.1.3 Load a new lifecycle

Armonic loads lifecycles at start time. To load a lifecycle, you have to put your lifecycle source code file in a folder in `armonic/modules` directory and restart the agent.

3.2 Write modules ready for orchestration

3.2.1 Replication

Some modules allow replication which improves performance or brings fault tolerance features. Different kinds of replications can be encountered such as master/slave or master/master.

In the master/master case, there is no difference between instances at runtime. Thus, they expose same API (require and provide). However, they could require differents parameters at set up time.

For instance, the first instance of a Galera cluster must create the cluster, all other will just connect to it. Once the cluster is built, all instances are equivalent.

The problem that appears is that instances are different at set up time and equivalent at runtime. Because they are equivalent at runtime, we don't want to use different state or lifecycle to build them.

This can be solved by using `armonic.variable.ArmonicHosts` and `armonic.variable.ArmonicHost` variables.

We suppose that replicated instances are always load balanced or managed by a common entity. For instance, Galera is load balanced by HaProxy. Thus, to build a Galera cluster, we first have to build a HaProxy load balancer. HaProxy will call X times the creation of Galera nodes.

ArmonicHost and ArmonicHosts

When building the deployment tree lib smart construct a list of all instances in the case of a replication and it also know which is the current instance.

The setup method of each instance can require the instances list by declaring the variable `armonic.variable.ArmonicHosts` in its *Requires*. You can also require the variable `armonic.variable.ArmonicHost` to get the address of the current instance.

Example:

```
@Require('nodes', [
    ArmonicHost("current", label="Current instance"),
    ArmonicHosts("list", label="List of instances")
])
def my_instance_setup_method(self, requires):
    nodes = requires.nodes.variables().list.value
    # eg: nodes = ["192.168.1.1", "192.168.1.2"]
    node = requires.nodes.variables().current.value
    # eg: node = "192.168.1.1"
```

With this complete list of instances and the current instance address the setup method can fully configure the current node.

CHAPTER 4

Complete documentation

4.1 Lifecycle anatomy

`Lifecycle` is an automaton with a set of `State` and transitions between these states.

The state machine represents differents steps (called states) for deploying a service. For example the installation, the configuration, the activation of the service.

When a state is reached, it is added in a stack managed by `Lifecycle`. This permits to know which states have been applied in order to be able to unapplied them. This stack in internally managed and thus not exposed.

4.1.1 State

A `State` describes the actions to do in a particular state of a `Lifecycle`. It can be actions when entering in the state or actions available when the state is applied (in the stack).

States are pure python classes and actions are the methods of the class. The actions are called provides. Methods must be decorated to be used as provides.

To create a state you just need to subclass `State`:

```
from armonic import State

class MyState(State):
    pass
```

More about *State*.

4.1.2 Provide

A `Provide` describes a `State` method of a `Lifecycle`. The `Provide` defines the list of requires to be provided in order to call a state method like it was arguments of the method.

For example a state provide (method) with no requires (arguments) can be declared like this:

```
from armonic import State

class MyState(State):

    @Provide()
    def my_provide(self):
```

```
        # actions here
        pass
```

Each state has at least the `Provide` `enter`. This method is called when entering the state.

To other methods are also reserved: `leave` and `cross`. You can override `leave` if actions must be done when leaving the state. `leave` cannot take any arguments. `cross` is used when the state is traversed.

Flags

Flags can be defined on a `Provide`. These flags are propagated to upper states when the `Provide` is called.

When a provide is called the state that contains the provide is applied (in the `Lifecycle` stack). It can be a state that is not the current state (ie the last state applied). In that case the provide flags will be propagated to the `cross` methods of each state that was applied after the provide's state.

See *Flags usage*.

4.1.3 Require

`Require` describes the arguments needed to call a `Provide`. A require is a group of `Variable` with some context (name, extra information...)

Different of types of requires can be used:

- `Require` defines arguments that should be provided.
- `RequireLocal` defines that another `Provide` must be called on *the same host* before. The result of this call can be used if needed.
- `RequireExternal` defines that another `Provide` must be called on *a different host* before. The result of this call can be used if needed.

More about *Requires*.

4.1.4 Variable

`Variable` describes a `Provide` argument with some context (name, default value, optional, validation and more).

`Variables` are grouped in `Require`.

More about *Variables*.

4.2 In depth documentation

4.2.1 Predefined States

Using a predefined state

Usually you just need to subclass the state and adjust some class attributes. Example with the `InstallPackagesApt` class:

```
from armonic.states import InstallPackagesApt

class InstallOpenLDAP(InstallPackagesApt):
    packages = ["openldap-server"]
```

Include the `InstallOpenLDAP` state in your Lifecycle transitions and you are done.

List of predefined states

Service activation

Package installation

4.2.2 State

4.2.3 Requires

A *require* allows user to specify values required to apply a state or to call a provide. Basically, a *require* is used to specify a tuple of values. These values are described by a list of `armonic.variable.Variable`.

Require describes parameter values and service dependencies

The configuration parameter of a service is often bound to the configuration parameter of a another services that the first one needs. A typical example is the *database_name* parameter of the Wordpress that has to be equal to the *database_name* parameter used to create the database by Mysql.

That means parameter values and dependencies can be handled together. Thus, we introduce `armonic.require.RequireLocal` and `armonic.require.RequireExternal` to describe service dependencies in Armonic.

Arity of parameters and dependencies

The parameter *nargs* of a *require* is used to specify how many times a require can be specified. For instance, a load balancer requires several backend where each backend consists of a host and port variable. The nargs parameters of the backend require of a load balancer can be set to '*'.

Require

Example with `armonic.require.Require`:

```
class ConfigureApache(State):

    @Require('ports', [VInt('http', default=80), VInt('https', default=443)])
    def set_ports(self, requires):
        print requires.ports.variables().http
        print requires.ports.variables().https
```

Note: When calling the `set_ports` provide you can define port values for http and https. If no values are provided default values are used.

RequireLocal

Example with armonic.require.RequireLocal:

```
class ConfigureWordpress(State):

    @RequireLocal("web_server", "//vhost",
                  provide_ret=[VString("directory", default="/var/www/wordpress")])
    def enter(self, requires):
        print requires.web_server.variables().directory
```

The local require describes the dependency between Wordpress and a web server such as Apache2. The second parameter is a xpath that will be specialized at runtime by the user. The default *directory* is suggested by Wordpress and could be used by Apache2 to configure a *vhosts*.

RequireExternal

```
class ConfigureWordpress(State):

    @RequireLocal("dbinfo", "//Mysql//add_database",
                  provide_ret=[VString("dbname"), VString("dbuser"), VString("dbpassword")])
    def enter(self, requires):
        print requires.dbinfos.variables().dbname
        print requires.dbinfos.variables().dbuser
        print requires.dbinfos.variables().dbpassword
```

Note: When entering the ConfigureWordpress state the add_database provide will be called on the same system and the result of this call is saved in the requires argument passed to the enter method. The provide to call is written with an xpath string.

---

armonic.require.RequireExternal has the same usage as armonic.require.RequireLocal. The only difference is that a host must be provided to armonic.require.RequireExternal.

4.2.4 Variables

Variables types

Armonic provides base classes for defining your own variables. They all inherit from armonic.variable.Variable. No dict like variable is provided since all variables have names.

class armonic.variable.**VString**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)

Variable of type string

pattern = None

Validate the value again a regexp

pattern_error = None

Error message if the value doesn't match the regexp

class armonic.variable.**VInt**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)

Variable of type int.

max_val = None

Maximum value

min_val = None
Minimum value

class armonic.variable.**VFloat**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Variable of type float.

class armonic.variable.**VBool**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Variable of type boolean.

class armonic.variable.**VList**(*name*, *inner*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
VList provide a list container for Variable instances.

Running the validation on VList will recursively run the validation for all contained instances.

Parameters

- **name** (*str*) – variable name
- **inner** (all instances of Variable) – the type of variable used in the list
- **default** (*list*) – default value
- **required** (*bool*) – required variable
- ****extra** – extra variable fields

Predefined variables

class armonic.variable.**Host**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
Variable for hosts.

Validate that the value is an IP or a hostname

class armonic.variable.**Hostname**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
Variable for hostnames.

Validate that the value is a hostname

class armonic.variable.**Port**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Variable for port numbers.

Validate that the value is between 0 and 65535

Custom validation

To specifiy a custom validation method subclass the variable type of your choice and implement a validate function. If the validation fails you must raise armonic.common.ValidationError with an error message:

```
from armonic.common import ValidationError
from armonic.variable import VString

class CustomVar(VString):

    def validate(self):
        if not self.value in ('foo', 'bar'):
            raise ValidationError('%s value should be foo or bar' % self.name)
        return True
```

Extra variable info

If you wish to provide additional context to the variable you can define extra arguments in the variable constructor:

```
>>> from armonic.variable import VString
>>> var = VString('username', label="Username", help="Type your username")
>>> print var.to_primitive()
{'default': None,
 'error': None,
 'extra': {'help': 'Type your username', 'label': 'Username'},
 'from_xpath': None,
 'name': 'username',
 'required': True,
 'type': 'str',
 'value': None,
 'xpath': None}
```

These extra infos can be used in clients that consume the `armonic.lifecycle.Lifecycle` API.

4.2.5 Flags usage

Given the following `armonic.lifecycle.Lifecycle`:

```
from armonic import Lifecycle, Transition, State, Flags

class StateA(State):

    @Flags(reload=True)
    def provide1(self):
        # do stuff
        pass

    @Flags(restart=True)
    def provide2(self):
        # do other stuff
        pass

class StateB(State):

    def cross(self, reload=False, restart=False):
        if reload:
            # do stuff
            pass
        if restart:
            # do stuff
            pass

class LifecycleA(Lifecycle):
    initial_state = StateA()
    transitions = [Transition(StateA(), StateB())]

lf = LifecycleA()
lf.state_goto('//StateB')
ret = lf.provide_call('//provide1')
# StateB.cross called with reload=True
ret = lf.provide_call('//provide2')
# StateB.cross called with restart=True
```

If the current state of `LifecycleA` is `StateB` and we call `provide1()`, `StateB.cross()` will be called with `reload=True` after that `provide1()` returns. If `provide2()` is called, `StateB.cross()` will be called with `restart=True`.

4.2.6 Armonic XPath

XPath (XML Path Language) is a query language for selecting nodes from an XML document. In Armonic, we use it to select resources, ie. such as lifecycles, states, provides, requires and variables. These resources are modelized by a tree such as:

```
Location (where lifecycles are loaded)
|
+--Webserver
|  |
|  +--NotInstalled (State)
|  |  |
|  |  +--enter (Provide)
|  |
|  +--Installed
|  |  |
|  |  +--enter
|  |
|  +--Configured
|  |  |
|  |  +--enter
|  |  |
|  |  +--create_document_root
|  |     |
|  |     +--document_root (Variable of type str)
|  |
|  +--Active
|     |
|     +--enter
|     |
|     +--start
|
+--WebSite
.  |
.  .
.  .
.  .
```

XPath as selectors

XPath permits to address one or several resources in a generic and standardized way. For instance, we can list all states of the lifecycle *Webserver* with the command:

```
$ armocli state "//WebServer/*"
```

The XPath *"//WebServer/*"* matches all states of resource WebServer which is a lifecycle. To get states of all lifecycles:

```
$ armocli state "/*/*"
```

To get description of provide start of state Active:

```
$ armocli provide "/*/WebServer/Active/start" -l
```

Absolute XPath

A absolute XPath is a XPath that contains locations also called LifecycleManager. They start with a */*. To query the Armonic API, Absolute XPath MUST be used. To avoid locations specification (which can be redundant with the agent address), you can use a prefix such as *//* or */*/*.

Relative XPath

A relative XPath is a XPath that doesn't start with a */*. They are used to describe Armonic resources independently of the location where they are loaded. Relative XPath are only used by the internal API and they don't concern end users.

Armonic resource URI

Since Armonic resources are modelized by a tree, we use the path to provide unique resource identifier. Some methods of the API needs of resource URI to avoid potential conflict. For instance, to reach a state, we have to use a URI instead of a more generic XPath (*state-goto* needs a XPath URI).

A XPath URI is just a XPath that matches only one resource. To be sure to provide an URI, use full specialized XPath, ie. don't use wildcards, *//*, etc.

4.3 Smart

4.4 API documentation

4.4.1 Lifecycle

class `armonic.lifecycle.`**`Lifecycle`**

The Lifecycle of a service or application is represented by transitions between `State` classes. The transitions list is specified in the class attribute `Lifecycle.transition`.

Main operations on a Lifecycle are:

- `Lifecycle.state_list()` to list available states,
- `Lifecycle.state_current()` to know the current state,
- `Lifecycle.state_goto()` to go from current state to another state.
- `Lifecycle.provide_call()` to call a provide.

States applied are recorded in a stack to be able to unapply them. The State stack does not contain the same State twice.

`abstract` = **False**

If the Lifecycle is abstract it won't be loaded in the LifecycleManager and in the XML registery.

`doc`()

Return docstring of this lifecycle.

`init`(*state*, *requires*=[])

If it is not already initialized, push state in stack.

initial_state = None
The initial state for this Lifecycle

os_type = <OsType(Ubuntu - 14.04)>
To specify the current OS type. By default, OS type is automatically discovered but it is possible to override this attribute to manually specify one.

provide_call (*state*, *provide_name*, *requires=[]*, *path_idx=0*)
Go to provide state and call provide.

Parameters

- **state** (state_name | `State`) – the target state
- **provide_name** (*str*) – name of the provide
- **requires** (*tuple of variable values and deployment info*) – variable values to fill the requires

```
([
    ("//xpath/to/variable", {0: value}),
    ("//xpath/to/variable", {0: value})
 ], {'source' : xpath, 'id': uuid})
```

Return type provide result

provide_call_args (*state_name*, *provide_name*)
From a provide_name, returns its needed arguments.

provide_call_path (*state*)
Get paths to call a provide in state.

Parameters **state** (state_name | `State`) – the target state

provide_call_requires (*state*, *path_idx=0*)
Get requires to call provide in state.

Parameters

- **state** (state_name | `State`) – the target state
- **path_idx** (*int*) – the path to use when there is multiple paths to go to the target State

provide_list (*reachable=False*)
Get all available provides

Parameters **reachable** (*bool*) – list only reachable provides from the current state

Return type [(`State`, [`Provide`])]

state_by_name (*name*)
Get state from its name

Parameters **name** (*str*) – the name of a state

Return type `State`

state_current ()
Get current state.

Return type `State`

state_goto (*state*, *requires=[]*, *path_idx=0*)
Go to state.

Parameters

- **state** (state_name | `State`) – the target state
- **requires** (*tuple of variable values and deployment info*) – variable values to fill the requires

```
([
    ("//xpath/to/variable", {0: value}),
    ("//xpath/to/variable", {0: value})
 ], {'source' : xpath, 'id': uuid})
```

- **path_idx** (*int*) – the path to use when there is multiple paths to go to the target State

Return type None

state_goto_path(*state*, *func=None*, *path_idx=0*)
Get one path to go to State.

Parameters

- **state** (state_name | `State`) – the target state
- **func** (*function*) – function to apply on all States of the path
- **path_idx** (*int*) – the path to use when there is multiple paths to go to the target State

Return type [(`State`, method), (`State`, method), ...]

state_goto_path_list(*state*)
Get the list of paths to go to State.

Parameters **state** (state_name | `State`) – the target state

Return type [[(`State`, method), (`State`, method), ...], ...]

state_goto_requires(*state*, *path_idx=0*)
Get Requires to go to State.

Parameters

- **state** (state_name | `State`) – the target state
- **path_idx** (*int*) – the path to use when there is multiple paths to go to the target State

Return type [`Provide`]

state_list(*reachable=False*)
To get all available states.

Parameters **reachable** (*bool*) – list only reachable states from the current state

Return type [`State`]

to_dot(*cross=False*, *enter_doc=False*, *leave_doc=False*, *reachable=False*)
Return a dot string of lifecycle.

class `armonic.lifecycle.`**LifecycleManager**(*os_type=None*, *autoload=True*, *public_ip='localhost'*)
The `LifecyleManager` is used to manage `Lifecyle` objects. It permits to interact with lifecycles by provinding xpaths.

The full path to a variable is:

```
/hostname/lifecycle_name/state_name/provide_name/require_name/variable_name
```

The xpath to get all states of the Mysql `Lifecyle` would be:

`//Mysql/*`

To get the `add_database` provide in the Mysql `Lifecyle`:

`//Mysql//add_database`

All methods of `LifecyleManager` returns python objects.

Parameters

- **os_type** – to specify which kind of os has to be used. If it is not specified, the os type is automatically discovered.
- **public_ip** – the public ip of the agent. This is used by clients to know how to contact services deployed by this agent.

from_xpath (*xpath*, *ret='lifecycle'*)
From a xpath try to get the object of type `ret`

Parameters

- **xpath** (*str*) – xpath to a ressource
- **ret** (*str*) – object type to return (lifecycle, state, provide, require, variable)

Return type `Lifecycle|State|Provide|Require|Variable`

info ()
Get info of armonic agent

Return type dict

lifecycle (*lifecycle_xpath*)
List loaded lifecycle objects

Parameters **lifecycle_xpath** (*str*) – xpath that matches lifecycles

Returns list of `Lifecycle`

Return type [`Lifecycle`]

load (*lf_name*)
Load a `Lifecycle` in the manager and register it in the XML register.

Parameters **lf_name** (*str*) – the `Lifecycle` name to load

Raises **LifecycleNotExist** if the `Lifecycle` isn't found

Returns the loaded `Lifecycle`

Return type `Lifecycle`

provide (*provide_xpath*)
Return provides that match provide_xpath and that can be reached (OS_TYPE).

Parameters **provide_xpath** (*str*) – xpath to provide

Returns list of provides that match provide_xpath

Return type [`Provide`]

provide_call (*provide_xpath_uri*, *requires=[ ]*, *path_idx=0*)
Call a provide of a lifecycle and go to provider state if needed

Parameters

- **xpath** (*str*) – xpath of the provide to call

- **requires** (*tuple of variable values and deployment info*) – variable values to fill the requires

```
([
    ("//xpath/to/variable", {0: value}),
    ("//xpath/to/variable", {0: value})
 ], {'source' : xpath, 'id': uuid})
```

Returns provide_xpath_uri call result

provide_call_path (*provide_xpath*)
Paths for provides that matches provide_xpath.

Parameters **provide_xpath** (*str*) – xpath to provide

Returns list of paths to call provides that match provide_xpath

Return type [(`Provide`, [path, ...])]

provide_call_requires (*provide_xpath_uri*, *path_idx=0*)
Requires for the provide.

Parameters

- **provide_xpath_uri** (*str*) – unique xpath to provide
- **path_idx** (*int*) – path to use when there is multiple paths to go to the provide

Returns list of provides to call it order to call provide_xpath_uri

Return type [`Provide`]

provide_call_validate (*provide_xpath_uri*, *requires=[ ]*, *path_idx=0*)
Validate requires to call the provide

Parameters

- **xpath** (*str*) – unique xpath of the provide to call
- **requires** (*tuple of variable values and deployment info*) – variable values to fill the requires

```
([
    ("//xpath/to/variable", {0: value}),
    ("//xpath/to/variable", {0: value})
 ], {'source' : xpath, 'id': uuid})
```

Returns list of validated provides to call in order to call provide_xpath_uri

Return type {'errors': bool, 'xpath': xpath, 'requires': [`Provide`]}

register()
Register the manager in the XMLRegistery.

state (*state_xpath*)
Return a list of states that matches state_xpath.

Parameters **state_xpath** (*str*) – xpath that can match multiple states

Returns list of `State`

Return type [`State`]

state_current (*lifecycle_xpath*)
Get the current state name of matched lifecycles.

Parameters **lifecyle_xpath** – xpath that can match multiple `Lifecycle`

Return type [State]

state_goto (*state_xpath_uri*, *requires=[]*, *path_idx=0*)
From the current state go to state.

Parameters

- **xpath** (*str*) – unique xpath of a state
- **requires** (*tuple of variable values and deployment info*) – variable values to fill the requires

```
([
    ("//xpath/to/variable", {0: value}),
    ("//xpath/to/variable", {0: value})
 ], {'source' : xpath, 'id': uuid})
```

Return type None

state_goto_path (*state_xpath*)
From the current state, returns all paths to goto states that match state_xpath.

Parameters **state_xpath** (*str*) – xpath that can match multiple states

Returns list of paths for every state matched by state_xpath

Return type [(State, [path])]

state_goto_requires (*state_xpath_uri*, *path_idx=0*)
Return the list a special provide required to go from the current state to the state that match state_xpath_uri.

Parameters

- **state_xpath_uri** (*str*) – unique state xpath
- **path_idx** (*int*) – path to use when there is multiple paths to go to the provide

Return type [Provide]

to_dot (*lf_name*, *reachable=False*)
Return the dot string of a lifecyle object

Parameters **lf_name** (*str*) – name of the lifecycle object

Return type dot file string

to_primitive (*lf_name*, *reachable=False*)
Return a serialized Lifecycle object

Parameters **lf_name** (*str*) – name of the Lifecycle object

Returns serialized Lifecycle object

Return type dict

to_xml (*xpath=None*)
Return the xml representation of the LifecyleManager.

uri (*xpath='//'*, *relative=False*, *resource=None*)
Return the list of xpath_uris that match this xpath.

Parameters

- **xpath** (*str*) – an xpath string
- **relative** (*bool*) – If true, returns relative xpath
- **resource** (*str*) – Returns only xpath that describe this resource type

Returns list of xpaths

Return type [xpath_uri]

class armonic.lifecycle.**MetaState**
Set by state.__new__ to add implementation of this metastate.

4.4.2 Provide

class armonic.provide.**Flags**(***flags*)
Decorator to define flags on a state method.

class armonic.provide.**Provide**(*name=None*, *requires=[]*, *flags={}*, ***extra*)
Basically, this describes the method of a armonic.lifecycle.State.

It contains the list of armonic.require.Require needed to call the method.

Parameters

- **name** – name of the method
- **requires** – list of requires
- **flags** – flags to be propagated
- **tags** (*list*) – a list of tags where tags are strings
- **label** – a human readable short description
- **help** – a long help message

fill(*requires=[]*)
Fill the provide with variables values.

Parameters **variables_values** – list of tuple (variable_xpath, variable_values):

```
("//xpath/to/variable", {0: value}),
("//xpath/to/variable", {0: value})
```

require_by_name(*require_name*)

Parameters **require_name** (*str*) – require name

Return type armonic.require.Require

to_primitive()
Serialize the provide to a python dict.

validate()
Validate the provide.

Raises ValidationError when validation fails

class armonic.provide.**ProvideHistory**(*initial_history=[]*)
Record provide calls.

4.4.3 Require

A Require permits to a module developper to specify what type of value it must provide to go to a state. They are specified in armonic.lifecycle.State.

Two subclasses of `Require` can be used if value can be provided by a lifecycle provider, namely `RequireLocal` and `RequireExternal`. These requires permit to specify the name of a provide and what variables it needs and returns. Moreover, it is sometime intersting to be able to call several time this provide and then, to use several values returned by this provide (see `armonic.varnish` for instance).

- `RequireLocal` specify a provide call on the same agent instance.
- `RequireExternal` specify a provide call on an other agent instance.

To provide values to a require, `Require.fill()` method has to be used. Note that this method is automatically called when a state is reached. `Require.fill()` take a dict (or a list) of primitive types to fill values of a require.

class `armonic.require.`**Require**(*name*, *variables*, *nargs='1'*, ***extra*)

Basically, a require is a set of `armonic.variable.Variable`. They are defined in a state and are used to specify, verify and store values needed to enter in this state.

To submit variable values of a require, `fill()` method must be used. Then, method `validate()` can be used to validate that values respect constraints defined by the require.

Parameters

- **name** – name of the require
- **variables** – list of variables
- **nargs** – variables occurences (1 or more, '*', '?')

factory_variable()

Return an Itercontainer of variables based on variables_skel

Return type IterContainer of `Variable`

fill(*variables_values*)

Fill the require with a list of variables values

Parameters **variables_values** – list of tuple (variable_xpath, variable_values) variable_xpath is a full xpath variable_values is dict of index=value

generate_args(*dct={}*)

Return a tuple. First element of tuple a dict of argName:value where value is the default value. Second is a list of argName without default value.

Parameters **dct** – To specify a argName and its value.

validate(*values=[]*)

Validate Require values. If values is specified, they are used to validate the require variables. Otherwise, you must already have fill it because filled values will be used.

Return type boolean

validate_one_set(*iterContainer*, *values={}*)

Validate Require values on one variables set. If values is specified, they are used to validate the require variables. Otherwise, you must already have fill it because filled values will be used.

Return type boolean

variable_by_name(*variable_name*, *index=0*)

From a variable name return the corresponding instance

Parameters

- **variable_name** (*str*) – variable name
- **index** (*int*) – variable set index

Return type `Variable`

variables(*index=0*, *all=False*)
Return variables of given index.

TODO: Check if index respect nargs. :param index: index of a variable set. :param all: if true returns all variables :rtype: iterContainer or ([iterContainer] if all == True)

exception armonic.require.**RequireDefinitionError**
This is raised when the definition of a require is not correct.

class armonic.require.**RequireExternal**(*name*, *xpath*, *provide_args=[]*, *provide_ret=[]*, *nargs='1'*, ***extra*)
To specify a configuration variable which can be provided by a *provide* of a external module. A 'host' variable is automatically added to the args list. It MUST be provided.

class armonic.require.**RequireLocal**(*name*, *xpath*, *provide_args=[]*, *provide_ret=[]*, *nargs='1'*, ***extra*)
To specify a configuration variable which can be provided by a *provide_name* of a local Lifecycle object.

nargs parameters permits to specify how many time you can call a provide. It can be '1', '?', '*' times. Then, variables is a list which will contains many values for each variables.

Parameters

- **name** – name of the require
- **xpath** – the path of the provide to call
- **provide_args** – default values for the provide
- **provide_ret** – provide return value
- **nargs** – provide occurences (1 or more, '*') or is optional ('?')

generate_args(*dct={}*)
Return a tuple. First element of tuple a dict of argName:value where value is the default value. Second is a list of argName without default value.

Parameters **dct** – To specify a argName and its value.

exception armonic.require.**RequireNotFilled**(*require_name*, *variable_name*)
Raise if the value of variable is None.

4.4.4 Variables

class armonic.variable.**ArmonicFirstInstance**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
This variable must be used to specify if an instance is the first one or not. This will be used by the lifecycle to realize some special initial stuff.

This special variable type allows smartlib to specify first instance and other. This is useful for replicated instances such as Galera.

class armonic.variable.**ArmonicHost**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
Internal variable that contains the host of an RequireExternal

class armonic.variable.**ArmonicHosts**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Internal variable to store the list of hosts when deploying multiple instances.

class armonic.variable.**ArmonicThisHost**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
This variable describe the host where the current provide is executed.

class armonic.variable.**Host**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
Variable for hosts.

Validate that the value is an IP or a hostname

class armonic.variable.**Hostname**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
Variable for hostnames.

Validate that the value is a hostname

class armonic.variable.**Port**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Variable for port numbers.

Validate that the value is between 0 and 65535

class armonic.variable.**VBool**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Variable of type boolean.

class armonic.variable.**VFloat**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Variable of type float.

class armonic.variable.**VInt**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Variable of type int.

max_val = None
Maximum value

min_val = None
Minimum value

class armonic.variable.**VList**(*name*, *inner*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
VList provide a list container for Variable instances.

Running the validation on VList will recursively run the validation for all contained instances.

Parameters

- **name** (*str*) – variable name
- **inner** (all instances of Variable) – the type of variable used in the list
- **default** (*list*) – default value
- **required** (*bool*) – required variable
- ****extra** – extra variable fields

class armonic.variable.**VString**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
Variable of type string

pattern = None
Validate the value again a regexp

pattern_error = None
Error message if the value doesn't match the regexp

class armonic.variable.**VUrl**(*name*, *default=None*, *required=True*, *from_xpath=None*, *modifier='%s'*, ***extra*)
Open an url, download the remote object to a local file and return the local path of this object.

This should be renamed.

get_file()

Return type A local file name which contain uri object datas.

class armonic.variable.**Variable**(*name*, *default=None*, *required=True*, *from_xpath=None*, ***extra*)
Describes a value used in a state provide.

Only name is required.

The type of a variable is validated (with `_validate_type()`) when the value is set. The value of a variable can be validated by hand with the `_validate()` method.

Parameters

- **name** (*str*) – variable name
- **default** – default value
- **required** (*bool*) – required variable
- **from_xpath** (*str*) – use the xpath value for this variable
- ****extra** – extra variable fields

validate(*value=None*)
Run the variable validation

Validate value or self.value if value is not set. If values is specified, they are used to validate the require variables. Otherwise, you must already have fill it because filled values will be used.

Set self.error when ValidationError is raised.

Raises ValidationError

validation(*value*)
Override for custom validation

4.4.5 Utils

Process

class armonic.process.**ProcessThread**(*type*, *status*, *module*, *command*, *cwd=None*, *callback=None*, *shell=None*, *env=None*)
Base class for running tasks

catch_output()
get command context

launch()
Thread is started and joined. This is a blocking method.

Return type True if process execution success

run()
run command

stop()
stop current process if exists

armonic.process.**run**(*executable*, *args=[]*, *cwd=None*, *env=None*)
Launch a executable and wait it. Return True if command succeed (ie. if executable return 0).

Parameters

- **executable** – Absolute path of executable
- **args** – List of arguments

- **cwd** – The working directory
- **env** – A optionnal dict containing environnement variable name and its value

Augeas

4.4.6 Clients

Smart

Smart module offers a high level way to call a provide. Function `smart_call()` generates steps to help the user to

- define LifecycleManager,
- specialize xpath provide,
- specify variable value,
- ...

To use this module, you have to create a `armonic.client.smart.Provide`, and call `armonic.client.smart.smart_call()`. In the following, the classical code to use this library.

First, we define by inheritance global behavior of provides. In this example, we want to 'manage' all provides:

```
from armonic.client.smart import Provide, smart_call

class MyProvide(Provide):
    def on_manage(self, data):
        return True
```

Then, we can build a provide from this classe and call smart_call on it which returns a generator. We use this generator to walk on provides:

```
my_provide = MyProvide("//a/xpath)
generator = smart_call(my_provide)
data = None
while True:
    provide, step, args = generator.send(data)
    data = None

    if step == "manage":
        print "Provide %s is managed!" % provide.generic_xpath
    elif step == "specialize":
        # Do others stuffs on specialize step
        # ...
    elif step == ....
```

Some tips about how it works...

About provide_ret validation: Since provide_ret variables's values are known at runtime, we need to do special thing to pass agent validation before deployement. Smart ignore validation errors returned by agents if error occurs on variables that belongs to provide_ret.

class `armonic.client.smart.`**`Provide`**(*generic_xpath*, *requirer=None*, *child_num=None*, *require=None*)

This class describe a provide and its requires and remotes requires contains provide. Thus, this object can describe a tree. To build the tree, the function `smart_call()` must be used.

To adapt the behavior of this class, redefine methods on_step and do_step, where step is manage, lfm, specialize, etc. If method do_step returns True, this step is 'yielded'. Method on_step takes as input the sent data.

Parameters

- **child_number** – if this Provide is a dependencies, this is the number of this child.
- **requirer** – the provide that need this require
- **require** – the remote require of the requirer that leads to this provide.

build_child(*generic_xpath*, *child_num*, *require*)
Build and return a new provide by using the same class.

do_lfm()
The step lfm is applied if it returns True.

Currently, do_lfm is already called, even if the provide is local. We may only call it when the provide is external

do_specialize()
Specialization can not be avoided. If the provide matches only 1 xpath, yield doesn't occurs if this method returns False.

Thus, by returning True, specialization always yields.

has_requirer()
To know if it is the root provide.

matches()
Return the list of provides that matched the generic_xpath

on_call(*call*)

on_specialize(*xpath*)
Actions after the provide has been specialized.

require = None
Contains the `Require` that requires this provide.

requirer = None
Contains the `Provide` that requires this current provide.

reset_lfm()
Reset all data set at the lfm step

update_scope_provide_ret(*provide_ret*)
When the provide call returns value, we habve to update the scope of the require in order to be able to use these value to fill depending provides.

validate(*values*, *static=False*)
Validate all variables using values from data. Moreover, variable value is set with values coming from data.

The static validation is used to validate variables before deployment is running. In this case, we don't handle error on provide_ret's variables since we don't know value returned by porvide calls.

Parameters **static** – If True, run a static validation.

Return type bool

variables()

Return type [`Variable`]

variables_scope()
Return the variable scope of this provide.

Return type [Variable]

variables_serialized()
Get variables in the format for provide_call

armonic.client.smart.**smart_call**(*root_provide*, *values={}*)
Generator which 'yields' a 3-uple (provide, step, optionnal_args).

Socket

class armonic.client.sock.**ClientSocket**(*host='127.0.0.1'*, *port=8000*, *handlers=[]*)
A simple socket client for armonic agent.

Logs emit by agent are forwarded to this client. To use them, add a logging handler with add_logging_handler() or they can be specified as arguments at init time.

Parameters **handlers** (*[logging.Handler]*) – To set handlers to forward agent logs

add_logging_handler(*handler*)
Set a handler. You can use handler defined by the standard logging module, for instance logging.StreamHandler

call(*method*, **args*, ***kwargs*)
Make a call to the agent. See armonic.lifecycle.LifecycleManager to know which methods can be called.

to_xml(*xpath=None*)
Return the xml representation of agent.

4.5 Running modes

Several modes are available to use Armonic.

4.5.1 Simulation

The SIMULATION flag inhibits body of provide method to be executed but requires are validated and states are applied.

This is really useful to develop interaction between module without applying any modification on the host.

4.5.2 Non validation on call

With the simulation mode, a problem occurs with provide ret values. Since we don't execute provides, Armonic provide methods don't generate return values which are required to fill provide ret values.

We can set the flag armonic.common.DONT_VALIDATE_ON_CALL to avoid these validation to occur on provide calls.

4.5.3 Don't call

This mode is just useful for client since it inbits the client to realize the provide call on the agent.

4.6 Indices and tables

- *genindex*
- *modindex*
- *search*

a

R

S

T

U

V