Solar Documentation

Release 0.0.1

OpenStack Foundation

June 14, 2016

Contents

1 Installation **3**
- 1.1 Local environment . . . 3

2 Usage **5**
- 2.1 Examples . . . 5
- 2.2 CLI API . . . 5

3 List of Solar tutorials **7**
- 3.1 Wordpress tutorial . . . 7

4 Developement environment **13**
- 4.1 Vagrant environment . . . 13
- 4.2 Contribution . . . 15

5 Solar Glossary **17**
- 5.1 Resource . . . 17
- 5.2 Handler . . . 17
- 5.3 Transport . . . 18
- 5.4 Event . . . 18
- 5.5 Composer . . . 18
- 5.6 System log component . . . 19
- 5.7 Orchestration component . . . 19

6 Solar Internal Architecture **21**

7 Resource **23**
- 7.1 Basic resource structure . . . 23
- 7.2 Handler . . . 23
- 7.3 Input . . . 24
- 7.4 Computable Inputs . . . 25
- 7.5 Action . . . 25
- 7.6 Tag . . . 25

8 Resource Repository **27**
- 8.1 Resource Repository spec . . . 27
- 8.2 Resource Repository import . . . 27
- 8.3 Resource Repository update . . . 27

9 Orchestration **29**

9.1 Entities . . . 29
9.2 Configuration . . . 30
9.3 Daemonizing solar-worker . . . 32
9.4 Deployment operations . . . 32

10 Transports **35**
10.1 How it works . . . 35
10.2 Sync transport . . . 35
10.3 Ssh host key checking . . . 35
10.4 Run transport . . . 36
10.5 BAT transport . . . 36

11 Examples **37**
11.1 Create resource for the puppet handler . . . 37

12 Preparing deployment plan **41**
12.1 Required information . . . 41
12.2 Changes . . . 41
12.3 Staged changes . . . 41
12.4 Events usage . . . 42
12.5 Deployment plan construction . . . 42

13 FAQ **43**
13.1 Why nodes/transports have hardcoded keys, ips and other inputs ? . . . 43
13.2 I want to use different SSH keys . . . 43
13.3 I want to use passwords not keys . . . 43
13.4 How can I run solar worker ? . . . 43
13.5 How can I configure solar ? . . . 43
13.6 What database can I use with solar ? . . . 44
13.7 Where can I find solar examples ? . . . 44

14 Indices and tables **45**

Solar provides flexible orchestration and resource management framework for deploying distributed systems. It leverages abstraction layer over commonly used configuration management systems like puppet, ansible etc. to enable complex, multi node orchestration.

Solar can be used as separate tool for quick prototyping deployment topology, but as a framework it can be also integrated with existing tools used to configure and deploy distributed systems including OpenStack clouds. Solar also provides control over resulting changes by introducing changes log and history for deployment entities. This enables more control over lifecycle management of infrastructure.

Solar can deploy and manage any distributed system, focusing on OpenStack ecosystem e.g. OpenStack itself, Ceph, etc. There are also other examples like Riak.

Contents:

CHAPTER 1

Installation

Please note that currently Solar is in a beta stage and it shouldn't be used in production environments.

We also recommend testing Solar using a vagrant where fully working development environment will be created.

If you want to try Solar outside Vagrant jump to *Local environment*

1.1 Local environment

If you want to test Solar locally you may install it via pip:

```
pip install solar
```

Create solar configuration *solar_config* file and paste following data:

```
solar_db: sqlite:////tmp/solar.db
```

and set path to this configuration:

```
export SOLAR_CONFIG_OVERRIDE=<full/path/solar_config>
```

For more information about configuration see our FAQ questions: *here*.

You also need to download Solar resources and add them to a Solar repository.

```
git clone https://github.com/Mirantis/solar-resources

sudo mkdir -p /var/lib/solar/repositories
sudo chown -R <your_user_name> /var/lib/solar/

solar repo import -l solar-resources/resources/
solar repo import -l solar-resources/templates/
```

Next step is to start Solar orchestration worker.

```
solar-worker
```

CHAPTER 2

Usage

To understand a workflow you should start with our *Wordpress tutorial*.

Solar can be used in three ways. Using CLI Api, python API and Composer files. The last one is showed in *Wordpress tutorial*.

2.1 Examples

Note: You need to have nodes resources created before running example. You can add them by calling *solar resource create nodes templates/nodes count=X* where *X* is required nodes number

Each API is used in different examples:

2.1.1 Python API

- 3 node cluster riak
- hosts files
- 2 node OpenStack Cluster

2.1.2 Composer files

- Wordpress site
- 3 node cluster riak

2.2 CLI API

1. Create some resources (look at *solar-resources/examples/openstack/openstack.py*) and connect them between each other, and place them on nodes.
2. Run *solar changes stage* (this stages the changes)
3. Run *solar changes process* (this prepares orchestrator graph, returning change UUID)
4. Run *solar orch run-once <change-uuid>* (or *solar orch run-once last* to run the lastly created graph)

5. Observe progress of orch with *watch 'solar orch report <change-uuid>'* (or *watch 'solar orch report last'*).

Some very simple cluster setup:

```
solar resource create nodes templates/nodes count=1
solar resource create mariadb_service resources/mariadb_service '{"image": "mariadb:5.6", "root_passw
solar resource create keystone_db resources/mariadb_db/ '{"db_name": "keystone_db", "login_user": "ro
solar resource create keystone_db_user resources/mariadb_user/ user_name=keystone user_password=keyst

solar connect node1 mariadb_service # it will mark mariadb_service to run on node1
solar connect node1 keystone_db
solar connect mariadb_service keystone_db '{"root_password": "login_password", "port": "login_port",
solar connect keystone_db keystone_db_user

solar changes stage
solar changes process
solar orch run-once last # or solar orch run-once last
solar orch report last -w 1000 # or solar orch report last
```

You can fiddle with the above configuration like this:

```
solar resource update keystone_db_user '{"user_password": "new_keystone_password"}'
solar resource update keystone_db_user user_password=new_keystone_password   # another valid format

solar changes stage
solar changes process
solar orch run-once last
```

To get data for the resource *bar* (raw and pretty-JSON):

```
solar resource show --tag 'resources/bar'
solar resource show --as_json --tag 'resources/bar' | jq .
solar resource show --name 'resource_name'
solar resource show --name 'resource_name' --json | jq .
```

To clear all resources/connections:

```
solar resource clear_all
```

Show the connections/graph:

```
solar connections show
solar connections graph
```

You can also limit graph to show only specific resources:

```
solar connections graph --start-with mariadb_service --end-with keystone_db
```

You can make sure that all input values are correct and mapped without duplicating your values with this command:

```
solar resource validate
```

Disconnect

```
solar disconnect mariadb_service node1
```

Tag a resource:

```
solar resource tag node1 test-tags # Remove tags
solar resource tag node1 test-tag --delete
```

CHAPTER 3

List of Solar tutorials

Contents:

3.1 Wordpress tutorial

3.1.1 1. Introduction

In this tutorial we will create Worpdress site using docker containers. We will create one container with Mysql database, then we will create database and user for it. After that we will create Wordpress container which is running on Apache.

In this tutorial we will use our vagrant environment. We need two virtual machines. One where Solar database and Orchestrator will run and one where we will install Wordpress and all components:

3.1.2 2. Solar installation

```
git clone https://github.com/openstack/solar.git
cd solar
vagrant up solar-dev solar-dev1
vagrant ssh solar-dev
```

3.1.3 3. Config resource

First we need to create Solar Resource definition where global configuration will be stored. This will be a *data container* only, so it will not have any handler nor actions. Let's create base structure:

```
mkdir -p wp_repo/wp_config/1.0.0
touch wp_repo/wp_config/1.0.0/meta.yaml
```

Open meta file *wp_repo/wp_config/1.0.0/meta.yaml* with your favorite text editor and paste the following data:

```
handler: none
version: 1.0.0
input:
  db_root_pass:
    schema: str!
    value:
  db_port:
```

```
    schema: int!
    value:
  wp_db_name:
    schema: str!
    value:
  wp_db_user:
    schema: str!
    value:
  wp_db_pass:
    schema: str!
    value:
```

Let's go through this document line by line. *handler: none* says that this resource has no handler and no actions. In next line we define version. The most important part starts from line 3. We define there the inputs for this resource. It will be possible to configure following inputs:

- *db_root_pass* - Mysql root password
- *db_port* - Mysql port
- *wp_db_name* - database name for Wordpress
- *wp_db_user* - database user name for Wordpress
- *wp_db_pass* - database user password for Wordpress

In schema it's defined if input will be string or integer, *!* at the end means that the input is mandatory and value cannot be empty.

3.1.4 4. Composer file

All other required resources are already available in solar repositores: *resources* and *templates*. We will use four more resources:

- resources/docker - it installs docker
- resources/docker_container - it manages docker container
- resources/mariadb_db - it creates database in MariaDB and Mysql
- resources/mariadb_user - it creates user in MariaDB and Mysql

There are three ways to create resources in Solar: Python API, CLI and Composer files. We will use the last option. Composer file is just a simple yaml file where we define all needed resources and connections. Run:

```
mkdir -p wp_repo/docker/1.0.0
```

Create new file *wp_repo/docker/1.0.0/docker.yaml*, open it and past the following data:

```
resources:
  - id: docker
    from: resources/docker
    location: node1

  - id: config
    from: wp_repo/wp_config
    location: node1
    input:
      db_root_pass: 'r00tme'
      db_port: 3306
      wp_db_name: 'wp'
```

```
    wp_db_user: 'wp'
    wp_db_pass: 'h4ack'

- id: mysql
  from: resources/docker_container
  location: node1
  input:
    ip: node1::ip
    image: mysql:5.6
    ports:
      - config::db_port
    env:
      MYSQL_ROOT_PASSWORD: config::db_root_pass
    wait_cmd:
      computable:
        func: "mysql -p{{env['MYSQL_ROOT_PASSWORD']}} -uroot -e 'SELECT 1'"
        connections:
          - mysql::env::NO_EVENTS

- id: wp_db
  from: resources/mariadb_db
  location: node1
  input:
    db_name: config::wp_db_name
    db_host: mysql::ip
    login_user: 'root'
    login_password: config::db_root_pass
    login_port: config::db_port

- id: wp_user
  from: resources/mariadb_user
  location: node1
  input:
    user_password: config::wp_db_pass
    user_name: config::wp_db_user
    db_name: wp_db::db_name
    db_host: mysql::ip
    login_user: 'root'
    login_password: config::db_root_pass
    login_port: config::db_port

- id: wordpress
  from: resources/docker_container
  location: node1
  input:
    ip: node1::ip
    image: wordpress:latest
    env:
      WORDPRESS_DB_HOST: mysql::ip
      WORDPRESS_DB_USER: wp_user::user_name
      WORDPRESS_DB_PASSWORD: wp_user::user_password
      WORDPRESS_DB_NAME: wp_db::db_name
```

In block *resources* we define... resources. Each section is one resource. Each resource definition has a following structure:

- id - resource name
- from - path to resource dir

- location - node where resource will be run
- values: initialization of a Resource Inputs

In *location* we define *node1*. It's name of our virtual machine resource. It's not created yet, we will do it shortly.

In our configuration there are two formats which we use to assign values to inputs. First:

```
db_port: 3306
```

It just means that input *db_port* will be set to *3306*

Another format is:

```
login_port: config::db_port
```

This means that input *login_port* will have the same value as input *db_port* from resource *config*. In Solar we call it Connection. When value of *db_port* changes, value of *login_port* will also change.

wait_cmd is special, it's *computable input*. In *wait_cmd* input we define command which will be used to check if docker container is ready. In this case it's

```
`mysql -pr00tme -uroot -e 'SELECT 1`
```

Password for mysql is defined in config resource and can change at any time. Instead of hard-coding it, computable input is used making this resource more maintainable.

When all files are ready we need add created resources to solar repository:

```
solar repo import wp_repo
```

This command created new solar resource repository. To list resources in this repository run:

```
solar repo show -r wp_repo
```

3.1.5 5. Deploying

Now it's time to deploy our configuration. When running *vagrant up solar-dev solar-dev1* you started two virtual machines. We will deploy Wordpress on solar-dev1. To do it we need to create a resource for it. We already have in repo composer file which is doing it. Just run:

```
solar resource create nodes templates/nodes count=1
```

It will create all required resources to run actions on solar-dev1. You can analyze content of *templates/nodes/1.0.0/nodes.yaml* later (that's the source for *templates/nodes*). Now we create resources defined in *docker*

```
solar resource create wp_docker wp_repo/docker
```

Command *create* requires name, but it's not used by Composer.

Now you can deploy all changes with:

```
solar changes stage
solar changes process
solar orch run-once
```

To see deployment progress run:

```
solar orch report
```

Wait until all task will return status *SUCCESS*. When it's done you should be able to open Wordpress site at http://10.0.0.3

If it fails, before reporting a bug, please try to retry deployment:

```
solar orch retry last
```

3.1.6 6. Update

Now change password for Wordpress database user

```
solar resource update config wp_db_pass=new_hacky_pass
```

and deploy new changes

```
solar changes stage
solar changes process
solar orch run-once
```

Using *report* command wait until all tasks finish. Wordpress should still working and new password should be used.

CHAPTER 4

Developement environment

4.1 Vagrant environment

Currently for development we are using Vagrant.

4.1.1 Additional software

VirtualBox 5.x, or Libvirt Vagrant 1.7.4 or higher

Note: Make sure that Vagrant VirtualBox Guest plugin is installed

```
vagrant plugin install vagrant-vbguest
```

Note: If you are using VirtualBox 5.0 on Linux system, it's worth uncommenting paravirtprovider setting in *vagrant-settings.yaml* for speed improvements:

```
paravirtprovider: kvm
```

For details see *Customizing vagrant-settings.yaml* section.

4.1.2 Setup development env

Setup environment:

```
git clone https://github.com/openstack/solar
cd solar
vagrant up
```

Login into vm, the code is available in /vagrant directory

```
vagrant ssh
solar --help
```

Get ssh details for running slave nodes (vagrant/vagrant):

```
vagrant ssh-config
```

You can make/restore snapshots of boxes (this is way faster than reprovisioning them) with the *snapshotter.py* script:

```
./snapshotter.py take -n my-snapshot
./snapshotter.py show
./snapshotter.py restore -n my-snapshot
```

snapshoter.py to run requires python module *click*.

- On debian based systems you can install it via

  sudo aptitude install python-click-cli,
- On fedora 22 you can install it via *sudo dnf install python-click*,
- If you use virtualenv or similar tool then you can install it just with

  pip install click,
- If you don't have virtualenv and your operating system does not provide

  package for it then *sudo pip install click*.
- If you don't have *pip* then

  [install it](https://pip.pypa.io/en/stable/installing/) and then execute command step 4.

4.1.3 Customizing vagrant-settings.yaml

Solar is shipped with sane defaults in *vagrant-setting.yaml_defaults*. If you need to adjust them for your needs, e.g. changing resource allocation for VirtualBox machines, you should just copy the file to *vagrant-setting.yaml* and make your modifications.

4.1.4 Image based provisioning with Solar

- In *vagrant-setting.yaml_defaults* or *vagrant-settings.yaml* file uncomment *preprovisioned: false* line.
- Run *vagrant up*, it will take some time because it builds image for bootstrap and IBP images.
- Now you can run provisioning */vagrant/solar-resources/examples/provisioning/provision.sh*

To develop Solar we use Vagrant

4.1.5 Using Libvirt instead of Virtualbox

Virtualbox is a default provider for Vagrant, but it's also possible to use another providers. It should be possible to use any of Vagrant providers. As for today we support Libvirt provider. It can be used only on Linux systems.

To use Libvirt with vagrant just run:

```
vagrant up --provider libvirt
```

This will download libvirt image for vagrant.

In nodes definition we have hardcoded ssh keys paths, where we assume that Virtualbox is used. You need to copy keys to vagrant libvirt dir:

```
cp /vagrant/.vagrant/machines/solar-dev1/libvirt/private_key /vagrant/.vagrant/machines/solar-dev1/v
```

Or you can change path in node transport as described in *FAQ*.

do it for each solar-dev* machine.

Note: Libvirt by default is using KVM. You cannot run KVM and Virtualbox at the same time.

4.2 Contribution

To track development process we are using Launchpad. To see on what we are currently working check Series and milestones.

4.2.1 Submiting patches

We are using OpenStack infrastructure to track code changes which is using Gerrit. To see all proposed changes go to Solar panel

4.2.2 Reporting bugs

To trach bugs we are using Launchpad. You can see all Solar bugs here

CHAPTER 5

Solar Glossary

5.1 Resource

Resource is an abstraction of item in system managed by Solar. It's a basic building block used to assemble your system. Almost every entity in Solar is a resource.

You can learn more about it in *resource details*

5.1.1 Input

Resource configuration that will be used in actions, handlers and orchestration. All known inputs for a resource should be defined in meta.yaml

5.1.2 Connection

Allows to build hierarchy between inputs of several resources, parent value will be always used in child while connection is created. If connection is removed - original value of child will be preserved.

5.1.3 Action

Solar wraps deployment code into actions with specific names. Actions are executed from the resource.

5.1.4 Tag

Used to create arbitrary groups of resources, later this groups will be used for different user operations.

5.1.5 Resource Repository

It is a named location where different *Resource* are located.

5.2 Handler

Layer responsible for action execution and tracking results.

5.3 Transport

Used in handlers to communicate with hosts managed by Solar.

See also:

More details about transports

5.3.1 location_id

Used in transport layer to find ip address of a node.

```
'location_id': '96bc779540d832284680785ecd948a2d'
```

5.3.2 transports_id

Used to find transports array that will be used for transport selection.

```
'transports_id': '3889e1790e68b80b4f255cf0e13494b1'
```

5.3.3 BAT transport

According to preferences solar will choose best available transport for file uploading and command execution.

5.4 Event

Used in solar to describe all possible transitions between resources changes. Each event allows to specify two points of transitions, condition of this transition and type of event.

Right now we are supporting 2 types of events:

1. Dependency - inserts edge between 2 changes into the deployment plan.
2. Reaction - inserts change specified in reaction and makes edge between parent and child.

Example

```
type: depends_on
parent: nova-db
parent_action: run
child: nova-api
child_action: run
state: success // condition
```

5.5 Composer

Composition layer that allows user to:

- group resources
- specify connections between inputs
- add list of events

5.6 System log component

Component responsible for tracking changes and keeping ordered history of them.

5.6.1 Staged log

Based on user changes - solar will create log of staged changes. This log will be used later to build deployment plan.

5.6.2 History

After action that is related to change will be executed - it will be moved to history with same uuid.

5.6.3 Committed resource data

After each successful change committed copy of resource data will be updated with diff of that change.

5.7 Orchestration component

5.7.1 Deployment plan

Based on changes tracked by system log and configured events - solar build deployment plan. In general deployment plan is built with

```
solar ch process
```

And can be viewed with

```
solar or dg last
```

5.7.2 Deployment plan operations

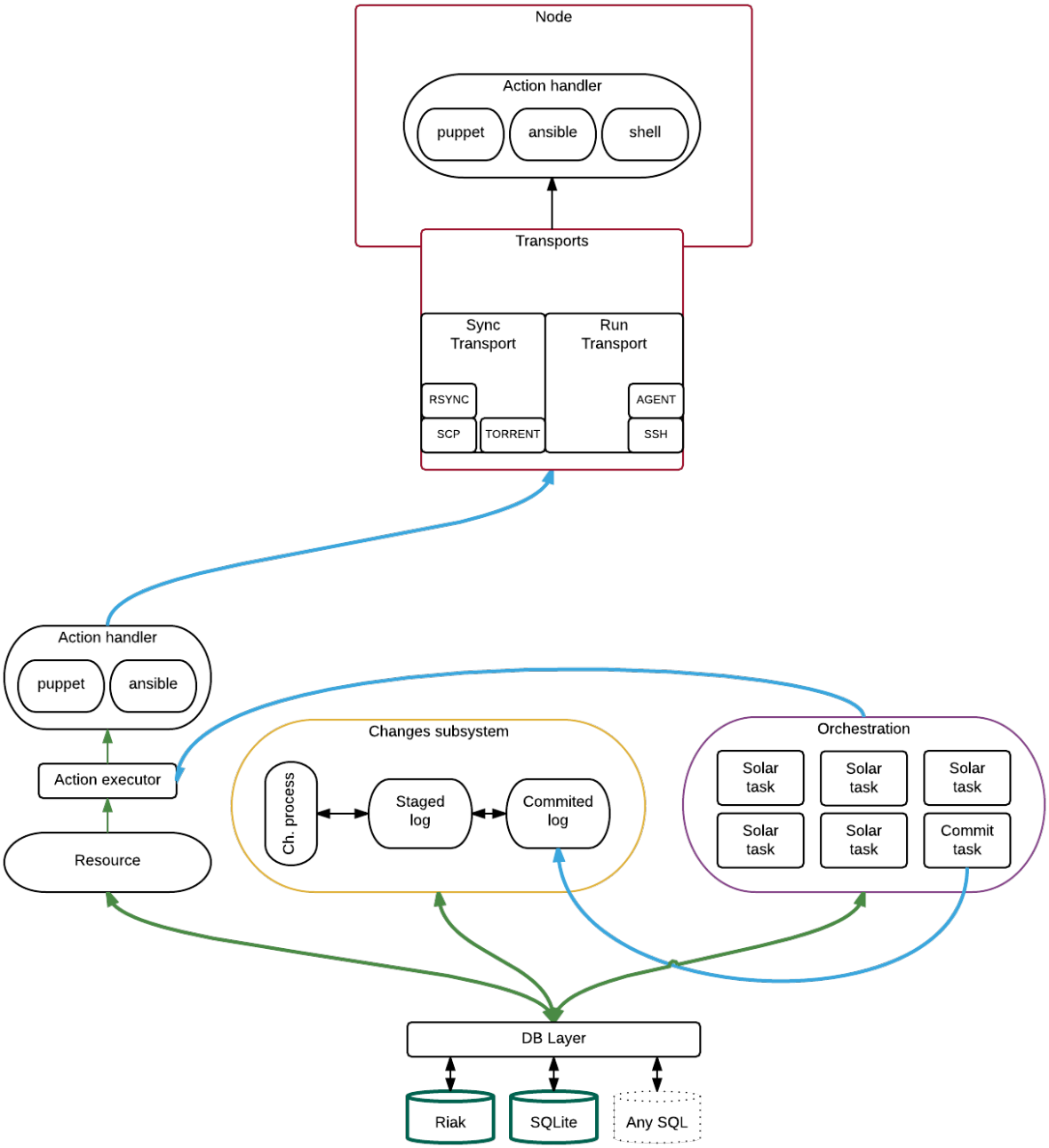
Solar cli provides several commands to work with deployment plan.

- run-once
- report
- stop
- resume/restart/retry

See also *Orchestration*

CHAPTER 6

Solar Internal Architecture



CHAPTER 7

Resource

Resource is one of the key Solar components, almost every entity in Solar is a resource. Examples are:

- packages
- services

Resources are defined in `meta.yaml` file. This file is responsible for basic configuration of given resource. Below is an explanation what constitutes typical resource.

Note: You can find example resources https://github.com/Mirantis/solar-resources

You can easily use this resource in your system, from CLI you just need to call `solar resource create` with correct options. During that operation solar will remember the version of this resource.

See also:

Resource Repository

7.1 Basic resource structure

```
-- actions
|   -- remove.pp
|   -- run.pp
|   -- update.pp
-- meta.yaml
```

7.2 Handler

Pluggable layer that is responsible for executing an action on resource. You need to specify handler per every resource. Handler is defined in `meta.yaml` as below

```
handler: puppet
```

Solar currently supports following handlers:

- puppet - first version of puppet handler (legacy, will be deprecated soon)
- puppetv2 - second, improved version of puppet, supporting hiera integration
- ansible_playbook - handler that supports more or less standard ansible playbooks

- ansible_template - handler that first generates ansible playbook using jinja template first (it's named `ansible`)

Handlers are pluggable, so you can write your own easily to extend functionality of Solar. Interesting examples might be Chef, SaltStack, CFEngine etc. Using handlers allows Solar to be quickly implemented in various environments and integrate with already used configuration management tools.

7.3 Input

Inputs are essentially values that given resource can accept. Exact usage depends on handler and actions implementation. If your handler is puppet, inputs are basically parameters that can be accepted by puppet manifest underneath.

All needed inputs should be defined in `meta.yaml` for example:

```
input:
    keystone_password:
      schema: str!
      value: 'keystone'
    keystone_enabled:
      schema: bool
      value: true
    keystone_tenant:
      schema: str
      value: 'services'
    keystone_user:
      schema: str
      value: 'cinder'
```

7.3.1 Input schema

Input definition contains basic schema validation that allows to validate if all values are correct. `!` at the end of a type means that it is required (`null` value is not valid).

- string type `str`, `str!`
- integer type `int`, `int!`
- boolean type `bool`, `bool!`
- complex types:
  - list of strings `[str!]`
  - hash with values `{a: str!}`
  - list with hashes `[{a: str!}]`
  - list with lists `[[]]`

7.3.2 Input manipulation

There is possibility to add and remove inputs from given resource. To do so you can use `solar input add` or `solar input remove` in Solar CLI.

7.4 Computable Inputs

Computable input is special input type, it shares all logic that standard input has (connections etc), but you can set a function that will return final input value.

Note: Remember, that you need to connect inputs to have it accessible in Computable Inputs logic.

Currently you can write the functions using:

- Pure Python
- Jinja2 template
- LUA

Besides that there are 2 types of Computable Inputs:

- `values`
  - all connected inputs are passed by value as `D` variable
- `full`
  - all connected inputs are passed as array (python dict type) as `R` variable, so you have full information about input.

In addition for `jinja` all connected inputs for current resource are accessible as first level variables.

7.4.1 Change computable input

You can change Computable Input properties by calling `solar input change_computable` in Solar CLI.

7.5 Action

Solar wraps deployment code into actions with specific names. Actions are executed by *Handler*

Several actions of resource are mandatory:

- run
- remove
- update

You can just put files into `actions` subdir in your resource and solar will detect them automatically based on their names, or you can also customize action file names in `meta.yaml`

```
actions:
  run: run.pp
  update: run.pp
```

7.6 Tag

Tags are used for flexible grouping of resources. You can attach as many tags to resource as you want, later you can use those tags for grouping etc

```
tags: [resource=hosts_file, tag_name=tag_value, just_some_label]
```

CHAPTER 8

Resource Repository

Resource Repository takes care about *Resource* definitions and it supports versioning.

Solar CLI supports following options:

```
add       Adds new resource to repository
contains  Checks if `spec` is in Solar repositories
destroy   Destroys repository
import    Imports repository to Solar
remove    Removes `spec` from Solar repositories
show      Shows all added repositories, or content of repository when `-r`
          given
update    Updates existing repository with new content
```

8.1 Resource Repository spec

spec is in format *{repository_name}/{resource_name}:{version_info}*, *version_info* is optional if omitted, latest (highest) will be used. Versions are in *Semantic Versioning* <*http://semver.org/*> format. You can also use >, >=, ==, <, <= operators to specify matches.

8.2 Resource Repository import

Command *solar repository import* it allows you to import existing repository or directory with resources into your system. It will traverse *source* path copy all resources definitions into repository and obviously proper structure will be automatically created.

Note: You may also check *–link* option to this command. It will just link repository contents so to import you need to have proper structure before.

8.3 Resource Repository update

Command *solar repository update* will update repository content with new data. With *–overwrite* flag it will overwrite conflicting resources definitions.

CHAPTER 9

Orchestration

Contents:

9.1 Entities

9.1.1 Worker

Worker encapsulates logic related to certain area in solar. Current we have next workers:

Scheduler Correctly initates execution plans and updates state of tasks.

Tasks Execute tasks scheduled by Scheduler worker

System log Updates system log e.g. commits and moves log item from staged log to history, or in case of error updates log item as erred

9.1.2 Executors

Each executor module should provide:

Executor Executor responsible for processing events and handle them via given worker. Concurrency policies is up to the executor implementation.

Client Client communicates with executor

In current version of Solar we are using executor based on Push/Pull zeromq sockets, and gevent pool for concurrent processing of events.

9.1.3 Subscriptions

Each public method of worker is subscribable, in current version 4 events are available to subscribers.

on_success Called in the case of successful execution, provides context, result and event arguments

on_error Called in the case of error, prorives context, error type, event arguments

before Called before method execution, provides only context

after Called after method executuon, provides only context

To subscribe use:

```
worker.method.on_sucess(callable)
```

Additionally each worker provides *for_all* descriptor which allows to subscribe to all public methods:

```
worker.for_all.before(callable)
```

9.2 Configuration

Orchestration is configured using two different methods.

1. *Config options*
2. *Entrypoints*

9.2.1 Config options

system_log_address

Passed to executor which will run system log worker

tasks_address

Passed to executor which will run tasks worker

scheduler_address

Passed to executor which will run scheduler worker

executor

Driver name should be registered in entrypoints, see *Executor namespace*

tasks_driver

Driver name should be registered in appropriate entrypoints (see *Worker driver namespaces*)

scheduler_driver

Driver name should be registered in appropriate entrypoints (see *Worker driver namespaces*)

system_log_driver

Driver name should be registered in appropriate entrypoints (see *Worker driver namespaces*)

runner

Driver name should be registered in entrypoints (see *Runner namespace*)

9.2.2 Entrypoints

Executor namespace

Note: solar.orchestration.executors

One specified in configuration will be used.

Extensions namespace

Note: solar.orchestration.extensions

Using driver namespaces for each worker - loads all workers.

Worker driver namespaces

Note:

solar.orchestration.drivers.tasks
solar.orchestration.drivers.scheduler
solar.orchestration.drivers.system_log

Only one driver can be selected from each namespace, see driver options in config.

Constructor namespace

Note: solar.orchestration.constructors

Loads callables from this namespace and executes hooks connected to those namespaces.

Constructor hooks namespaces

Note:

solar.orchestration.hooks.tasks.construct
solar.orchestration.hooks.system_log.construct
solar.orchestration.hooks.scheduler.construct

All callables in each hook will be loaded and executed before spawning executor with instance of worker. Currently all subscriptions are done in this hooks.

Runner namespace

Note: solar.orchestration.runners

Runner should be selected in solar config. Runner will be executed as a last step in solar-worker main function.

9.3 Daemonizing solar-worker

9.3.1 Upstart

To daemonize solar-worker on debian or ubuntu upstart script should be used, in script and pre-script stanzas - */etc/default/solar-worker* will be sourced, and following variables used:

```
SOLAR_UID=solar
SOLAR_GID=solar
SOLAR_PIDFILE=/var/opt/solar/solar-worker.pid
```

Warning: SOLAR_UID and SOLAR_GID should be present in the system.

9.4 Deployment operations

9.4.1 Stage changes

After user created all required resource - it is possible to automatically detect which resource requires changes with

```
solar changes stage
```

9.4.2 History

After changes are staged - they will be used to populate history which can be previewed with command (*n* option used to limit number of items, -1 will return all changes)

```
solar changes history -n 5
```

9.4.3 Prepare deployment plan

User is able to generate deployment scenario based on changes found by system log.

```
solar changes process
```

This command will prepare deployment graph, and return uid of deployment graph to work with.

All commands that are able to manipulate deployment graph located in *orch* namespace.

Tip: Solar writes returned deployment graph uid into special file (*.solar_cli_uids*), it allows you to use *last* instead of full returned uid: *solar orch report <uid>* becomes *solar orch report last*

9.4.4 Report

Report will print all deployment tasks in topological order, with status, and error if status of task is *ERROR*

```
solar orch report <uid>
```

9.4.5 Graphviz graph

To see picture of deployment dependencies one can use following command

```
solar orch dg <uid>
```

Keep in mind that it is not representation of all edges that are kept in graph, we are using trasitive reduction to leave only edges that are important for the order of traversal.

9.4.6 Run deployment

Execute deployment

```
solar orch run-once <uid>
```

9.4.7 Stop deployment

Gracefully stop deployment, after all already scheduled tasks are finished

```
solar orch stop <uid>
```

9.4.8 Resume deployment

Reset SKIPPED tasks to PENDING and continue deployment

```
solar orch resume <uid>
```

9.4.9 Restart deployment

All tasks will be returned to PENDING state, and deployment will be restarted

```
solar orch restart <uid>
```

9.4.10 Retry deployment

Orchestrator will reset all ERROR tasks to PENDING state and restart deployment

```
solar orch retry <uid>
```

CHAPTER 10

Transports

Transports are used by Solar to communicate with managed nodes. Transports are also resources, so they have all resources features and flexibility. Transports should be added to a node, but if you need you can add different transports for different resources.

10.1 How it works

Each resource in solar has a random *transports_id* generated, when resources are connected to each other. Solar will ensure that correct *transport_id* is used. Then using this *transport_id* a correct real value is fetched. Changing transports contents will not cause *resource.update* action for related resources.

10.2 Sync transport

This transport uploads required information to target node.

Currently there are following sync transports available:

- ssh
- rsync
- solar_agent
- torrent

10.3 Ssh host key checking

Solar wont disable strict host key checking by default, so before working with solar ensure that strict host key checking is disabled, or all target hosts added to .ssh/known_hosts file.

Example of .ssh/config

```
Host 10.0.0.*
  StrictHostKeyChecking no
```

10.4 Run transport

This transport is responsible for running commands on remote host.

Currently there are following run transports available:

- ssh
- solar_agent

10.5 BAT transport

A transport that will automatically select best available transport (BAT) that is available for a given resource. Currently it's default transport in the system, so when you add more transports, everything should configure automatically.

CHAPTER 11

Examples

11.1 Create resource for the puppet handler

Let's create an example *Resource* for the puppet *Handler* version 1 [1]. The resource should install and configure OpenStack Nova API service.

11.1.1 Step 1: Find an appropriate puppet module

The Puppet OpenStack module for Nova provides all of the required functionality.

11.1.2 Step 2: Define granularity level for a resource

One may want to implement resources as atomic entities doing their only single task, like running one and only puppet manifest [2]. Other option might be single entity doing all required tasks instead. In order to configure and run the Nova API service at least two manifests should be executed: init.pp and api.pp [3].

Assuming the atomic tasks approach, the example resource for Nova API service should only use the *api.pp* manifest. Note that the puppet handler is normally executed in its own isolated puppet catalog with its specific hiera data only. This assumes every puppet manifest called by every action to be executed as a separate puppet run and shares nothing with other tasks.

11.1.3 Step 3: Define resource inputs

Once the granularity level of the resource is clearly defined, one should define the resource's *Input* data. The puppet class *nova::api* contains lots of parameters. It looks reasonable to use them as the resource inputs as is.

Note: There is a helper script to convert a puppet class parameters into the format expected by the *meta.yaml* inputs file.

---

[1] There is also puppet handler version 2 but it is out of the scope of this example.
[2] Puppet manifests may contain references to externally defined classess or services in the catalog. Keep that in mind then designing the resource.
[3] This assumes configuring DB and messaging entities like user, password database, vhost, access rights are left out of the scope of this example.

11.1.4 Step 4: Implement basic action run

Each resource should have all of the mandatory actions defined. In this example we define only the ref-action-term *run*. With the example of Nova API resource, the action run should:

- fetch the resource inputs from the hiera [4]

```
$resource = hiera($::resource_name)
$ensure_package = $resource['input']['ensure_package']
$auth_strategy = $resource['input']['auth_strategy']
```

- call the *class { 'nova::api': }* with the required parameters
- implement workarounds for externally referenced entities, like

```
exec { 'post-nova_config':
  command     => '/bin/echo "Nova config has changed"',
}

include nova::params

package { 'nova-common':
  name   => $nova::params::common_package_name,
  ensure => $ensure_package,
}
```

Note: Otherwise, called class would assume the package and exec are already included in the catalog by the *init.pp*. And would fail as there is no *class { 'nova': }* call expected for the Nova API resource action run. In order to implement the resource without such workarounds, one should rethink the granularity scope for the resource. And make sure the resource contains required inputs for the main *nova* and *nova::api* classes and call them both in the resource action run.

11.1.5 Step 5: Think of the rest of the resource actions

One should also design other actions for the resource. Mandatory are only *run*, *update* and *remove*. There might be additional ones like *on-fail*, *on-retry* or whichever are actually required to implement expected behavior. For the given API resource there are no specific actions expected and there is nothing to do for the action remove. For the action update, it is likely the same steps should be done as for the action run.

11.1.6 Step 6: Design the high level functional test

TODO(bogdando) provide details about test.py and writing tests for Nova API in order to verify if it works on the app level.

11.1.7 Step 7: Think of the deployment composition

The deployment composition is which resources should be used and in which order it should be executed to achieve the expected result, which is a successful *Deployment plan*. For the given example, the composition may be as following:

- Install and configure MySQL DB [5]

[4] The syntax is the puppet handler v1 specific. The v2 allows to query the hiera directly, like *$public_vip = hiera('public_vip')*

[5] Omitted host related steps like OS provisioning, disks and network configuration.

- Install and configure RabbitMQ node
- Install and configure dependency components like OpenStack Keystone
- Create all of the required user/tenant/db/vhost entities and assign rights
- Install and configure Nova main components, like packages, db sync, configs.
- Install and configure Nova API. BINGO! A job for our resource, at last!

Besides the execution plan, there is also data *Connection* to be considered. For example, one might want to have all of the OpenStack services to use the common RabbitMQ virtualhost and user. Or have them separated instead. Or use the clustered RabbitMQ nodes. These decisions will directly impact how resources' inputs should be connected.

CHAPTER 12

Preparing deployment plan

Solar allows you to make transitions between different versions of infrastructure based on changes found by solar control plane and events configured between this changes.

12.1 Required information

- *Input*
- *Orchestration*

12.2 Changes

By changes in solar we understand everything that is explicitly made by user (human/program). Examples of changes are: - create resource - remove resource - update value manually - update value using hierarchy

12.3 Staged changes

On demand solar runs procedure that will find all resources changed from last deployment and will determine list of actions using transitions from solar state machine .

This procedure is performed by

```
solar changes stage -d
```

It prints information like

```
log task=openrc_file.run uid=e852455d-49d9-41f1-b49c-4640e2e19944
    ++ ip: 10.0.0.3
    ++ location_id: 694b35afa622da857f95e14a21599d81
    ++ keystone_port: 35357
    ++ transports_id: abc7745f2ad63709b5845cecfa759ff5
    ++ keystone_host: 10.0.0.3
    ++ password: admin
    ++ user_name: admin
    ++ tenant: admin
log task=neutron_db.run uid=95cac02b-01d0-4e2f-adb9-4205a2cf6dfb
    ++ login_port: 3306
    ++ encoding: utf8
```

```
    ++ login_user: root
    ++ login_password: mariadb
    ++ transports_id: abc7745f2ad63709b5845cecfa759ff5
    ++ db_name: neutron_db
    ++ db_host: 10.0.0.3
    ++ ip: 10.0.0.3
    ++ collation: utf8_general_ci
    ++ location_id: 694b35afa622da857f95e14a21599d81
```

At this point information is stored as a list, and user doesn't know anything about dependencies between found changes.

12.4 Events usage

For events definition see *Event*.

Events are used during this procedure to insert dependencies between found changes, and to add new actions that are reactions for changes.

Example of dependency between changes would be *nova service* that depends on successful creation of *database*.

For removal we might add dependencies that will allow reverse order, e.g. when removing *nova service* and *database*, *database* will be removed only after successful *nova service* removal.

This can be specified as

```
Dependency database1.run -> nova1.run
Dependency nova1.remove -> database1.remove
```

Reaction allows to construct more advanced workflows that will take into account not only changes, but also arbitrary actions for resources in solar.

Good example of usage is provisioning procedure, where reboot must be done only after node is provisioned, and dnsmasq configuration changes to reflect that that node is now using statically allocated address. We can specify such ordering as

```
React node1.run -> node1.reboot
React node1.run -> dnsmasq1.change_ip
Dependency dnsmasq1.change_ip -> node1.reboot
```

12.5 Deployment plan construction

Using list of staged changes and graph events we can proceed with construction of deployment plan for current version of infrastructure

```
solar changes process
```

After this deployment command plan can be viewed by

```
# graphviz representation
solar orch dg last

# text report
solar orch report last
```

CHAPTER 13

FAQ

13.1 Why nodes/transports have hardcoded keys, ips and other inputs ?

This is temporary situation, we will improve it in near future.

13.2 I want to use different SSH keys

Just update resource for example:

```
solar resource update ssh_transport1 '{"key": "/path/to/some/key"}'
```

13.3 I want to use passwords not keys

Just update resource:

```
solar resource update rsync1 '{"password": "vagrant", "key": null}'
```

Note: You need to change it for all transport resources (ssh and rsync by default).

13.4 How can I run solar worker ?

- If you use *vagrant* then you can just *sudo start solar-worker*

as *vagrant* user.

13.5 How can I configure solar ?

There are several places where we search for config values:

1. *.config* file in CWD or in path from *SOLAR_CONFIG* env variable

2. if env *SOLAR_CONFIG_OVERRIDE* contains valid path then it override previous values 3. *.config.override* in CWD 4. You can also set upper-cased env variable which matches one of those in config

13.6 What database can I use with solar ?

By default for simplicity we use *sqlite*. On our vagrant environment we use single node *riak*. You can also use multiple nodes *riak*, with some strong consistent buckets.

13.7 Where can I find solar examples ?

Example resources, composer templates and examples itself are located: https://github.com/Mirantis/solar-resources

CHAPTER 14

Indices and tables

- search