PROVISIONING THE ORACLE DATABASE IN THE CLOUD

Frits Hoogland, Accenture Enkitec Group

ABSTRACT

This presentation provides a walkthrough of provisioning a full Oracle database software install using Ansible. The presentation also touches on automating local virtual machine provisioning for test and development using vagrant. 'Full' not only means it is showing how to get Oracle installed, but also fully patched to the latest PSU. It goes beyond showing concepts, which means actual execution of the deployment is shown, step by step, via live demo's. In today's world, repetitive tasks like installing a database should be automated and not done manually using expensive administrator hours. Maybe the even most important aspect (more important than expensive hours being used) is that manual installations are guaranteed to have inconsistencies, which will make administration (and further automation) hard. This is a must see session for anyone who is serious about using Oracle databases at scale (which could be cloud)!

TARGET AUDIENCE

The target audience for this paper are architects, database administrators and developers who are interested in automating operations.

EXECUTIVE SUMMARY

Any operational task that needs to be performed at scale, for which typically the cloud is a good example, but also local datacenter operational tasks, down to even local laptop test installations, should be automated to guarantee a consistent result. A certain percentage of manual executed tasks will have deviations from the intended outcome. Deviations of any imposed standard makes administration and/or automation of administrative tasks on these inconsistent environment hard.

This presentation focusses on Ansible as an automation tool to create highly consistent provisioning results. It also uses Rundeck as the repository and execution engine on top of Ansible. The combination of Rundeck and Ansible provides:

- Accountability
- Traceability
- Delegation of tasks
- REST integration

To demonstrate the provisioning, a local deployment is demonstrated using Vagrant, Virtualbox and Ansible. Vagrant and Virtualbox are tools to provide a local running system image, which can be easily swapped with a cloud based image, not restricted to a cloud vendor.

Ansible allows administrators or developers to describe the intented state, and let the ansible module handle the complexities. This means Ansible playbooks are generally well readable.

VAGRANT

Vagrant integrates a repository of operating system images, called ‘boxes’, together with a virtualization provider, which is Virtualbox by default. In order to create a Vagrant managed virtual machine, simply create a Vagrantfile to indicate your settings, like:

```
Vagrant.configure("2") do |config|
  config.vm.box = "box-cutter/ol73"
  config.ssh.insert_key = false
  config.ssh.private_key_path = [ "~/.vagrant_ssh/id_dsa",
"~/.vagrant.d/insecure_private_key" ]
  config.vm.hostname = "rundeck.local"
  config.vm.network "private_network", ip: “192.168.66.100"
  config.vm.provider "virtualbox" do |vb|
   vb.name = “rundeck"
   vb.memory = "1024"
   vb.cpus = "1"
   # PERL L4 CACHE WORK AROUND
   if !File.exist?("u01_disk.vdi")
    vb.customize [ 'createhd', '--filename', "u01_disk.vdi",
'--size', 40960 ]
   end
   vb.customize [ 'storageattach', :id, '--storagectl', 'SATA
Controller', '--port', 2, '--device', 0, '--type', 'hdd', '--
medium', "u01_disk.vdi" ]
  end
```

```
end
```

A couple of noteworthy things to point out: 'config.vm.box' points to the virtual machine image that is pulled from the vagrant repository. 'config.ssh.insert_key' can inject a key to the virtual machine to override the default (publically known) one, I chosen to use the default, unsafe, one, because this is used for demonstrations. I also use 'config.ssh.private_key_path' to point vagrant to a non-default private key path, because ssh requires the private key to have mode 400, which my external HDD doesn't support. Other settings are a hostname (config.vm.hostname), a second network interface with an ip address (config.vm.network), and then the virtualization provider (config.vm.provider) specific settings like memory and number of CPUs. Additionally, an extra disk is added to the virtual machine, in order to have room for a bigger installation like an Oracle database install.

To startup the virtual machine, simply type 'vagrant up' in the directory with the Vagrantfile. The 'vagrant up' command will check if the virtual machine image for the given (virtualbox) provider is already present. If not, it will see if there's a local cached copy of the vagrant 'box' (the template of the virtual machine), and download that from the vagrant repository if it's not available. If the virtual machine didn't exist, it will create the virtual machine and apply all the settings for the provider selected. At this point it will start up the virtual machine, and apply the provisioning if the provisioning was not applied before.

AUTHENTICATION

Vagrant uses a default user account ('vagrant') to be able to log in to the virtual machine using ssh using certificate based authentication, after which sudo can be used to switch to another user or the superuser (root). This is the standard way for cloud machines to provide access too, for example, the Oracle cloud uses 'opc' as the default user account. The configuration for vagrant is done via /etc/sudoers.d/vagrant:

```
%vagrant ALL=(ALL) NOPASSWD: ALL
```

In fact, this let all users in the *group* vagrant have all access (the percent means 'group'). The Oracle public cloud uses /etc/sudoers.d/90-cloud-init-users:

```
opc ALL=(ALL) NOPASSWD:ALL
```

Because Vagrant is meant for local deployments, virtual machines from the Vagrant repository do allow password authentication too. Most, if not all, cloud based virtual machines do not allow password based authentication as a security precaution, and the require a key as part of virtual machine startup, which is 'injected' into the virtual

machine at startup. Injected means the public key is added to the authorized_keys file of the default user account.

YUM REPOSITORY

Whenever you update packages for groups of linux systems, I strongly recommend to first replicate the repository that is used for that to an in-house server, after which you apply the update to the first group of servers for testing any issues. Having a repository in-house is a guaranteed way to be sure none of the packages are changed while going through the cycles of deploying the update to the different tiers of test, development, acceptance, pre-production and production.

A yum repository is simply a directory structure with some meta-data, which is served by an HTTP server. This is an example configuration for the NGNIX webserver to serve the repositories in the directory /repo:

```
server {
  listen *:80;
  root /repo;
  location / { autoindex on; }
}
```

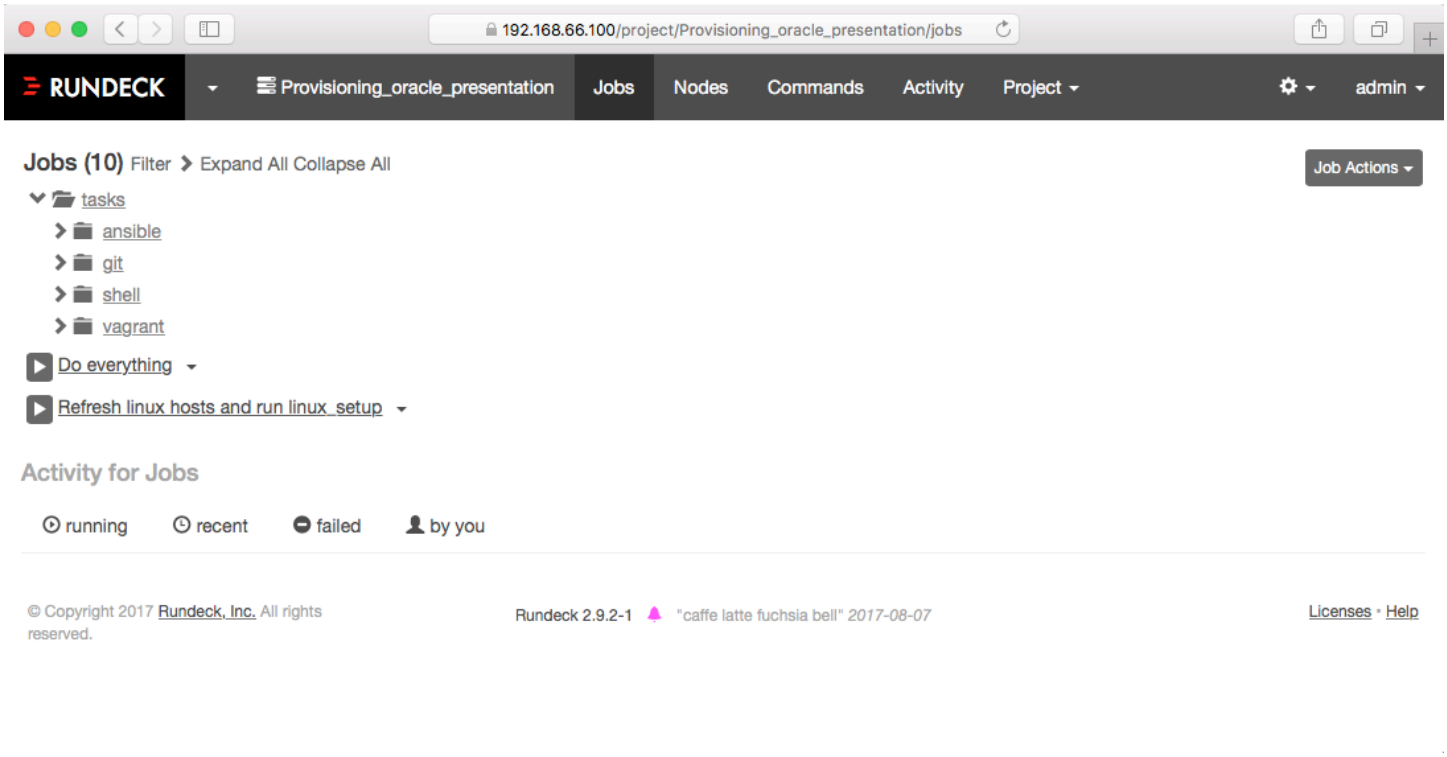
In order to create a yum repository, a few simple steps need to be done:

1. Install a few additional packages:
   ```
   yum install createrepo yum-utils
   ```
2. Create a directory to hold the repository:
   ```
   mkdir /repo/myrepo; cd $_
   ```
3. Download all packages from a repository ('yum_repo' is an accessible remote repository defined in /etc/yum.repos.d/*.repo):
   ```
   reposync –r yum_repo .
   ```
4. Generate the yum repository metadata
   ```
   createrepo --update .
   ```
5. Setup a selinux context to allow http serving (only if selinux is active):
   ```
   chcon –Rt httpd_sys_content_t /repo
   ```

Optionally, these steps can be executed automatically using the Ansible setup_repo.yml file in http://gitlab.com/FritsHoogland/setup_repo.git. Ansible, once installed, allows to execute against localhost without any keys and host inventories setup.

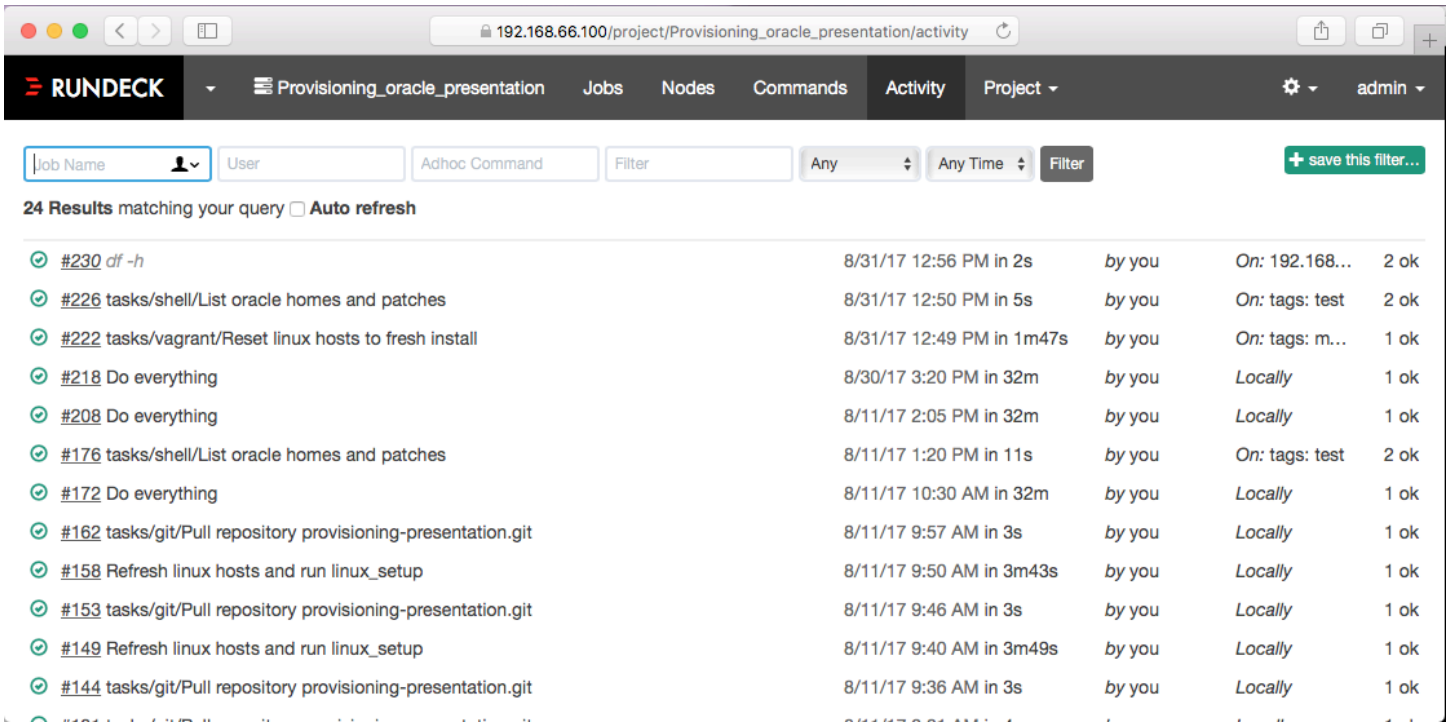
RUNDECK

Rundeck is an operations user interface for running tasks that can be Ansible playbooks, but also just simple shell commands or shell scripts.



Rundeck project interface, showing a tasks tree, and some tasks.

The biggest advantage of using a product like Rundeck is that any execution always is directly linked to a user executing it, and all output of the execution is saved, so any issues can easily be diagnosed or shared for diagnosis.



Rundesk activity overview, showing task logging.

A description on how to install a rundeck setup is available in the blogpost: https://fritshoogland.wordpress.com/2017/08/09/installation-of-rundeck-with-ansible-plugin-on-centos-7/

Rundeck is not the only choice. There is an independent open source Ansible server, called semaphore: https://fritshoogland.wordpress.com/2017/04/29/how-to-install-the-semaphore-ui-for-running-ansible/ however, that is limited strictly to running Ansible playbooks. Other alternatives that I found (also limited to running Ansible playbooks) are Redhat's Tower (https://www.ansible.com/tower) and the 'upstream' open source version of it, which means it's the development ground for Tower, which is called awx (https://github.com/ansible/awx).

ANSIBLE

In order to show the power of using Ansible, a few playbooks are shown here and explained.

1. Setup linux: validate and optionally install a specific kernel version, and them make sure the virtual machine runs that kernel version.

```
01: ---
02: - hosts: all
03:   become: true
04:   vars:
```

```
05:     kernel: kernel-uek-4.1.12-61.1.28.el7uek
06:   tasks:
07:
08:   - name: install kernel
09:     yum:
10:       name: "{{ kernel }}"
11:       state: present
12:
13:   - name: set the desired kernel as default kernel
14:     shell: >
15:       /sbin/grubby --set-default=/boot/vmlinuz-{{ kernel |
regex_replace('^kernel-(.*)$','\1') | regex_replace('^uek-(.*)$','\1') }}.x86_64
16:
17:   - name: is the running kernel the desired kernel?
18:     shell: >
19:       if [ $(uname -r) = $( /sbin/grubby --default-kernel | sed
"s/\/boot\/vmlinuz-//" ) ]; then echo "yes"; else echo "no"; fi
20:    register: remain_running
21:
22:   - name: reboot to activate kernel
23:     shell: >
24:       sleep 2; /sbin/shutdown -r now "Ansible activate new kernel"
25:     async: 1
26:     poll: 0
27:     when: "'no' in remain_running.stdout"
28:
29:   - name: wait for server to come down
30:     local_action: wait_for
31:     args:
32:       host: "{{ inventory_hostname }}"
33:       port: 22
34:       state: stopped
35:       timeout: 600
36:     become: false
37:     when: "'no' in remain_running.stdout"
38:
39:   - name: wait for server to come back up
40:     local_action: wait_for
41:     args:
42:       host: "{{ inventory_hostname }}"
43:       port: 22
44:       state: started
45:       timeout: 600
46:     become: false
47:     when: "'no' in remain_running.stdout"
```

1: Ansible works with Yaml document, which start with '---'.

3: Become: true means this is run as root, which is obviously needed when you want to install anything, and change kernel settings.
5: A variable is declared that is called ‘kernel’ and contains a kernel version.
8: This task is self-explanatory: this installs the kernel package using the ‘yum’ module. If the package is already installed, it will simply mark the task as ‘okay’, because the task is fulfilled without any change.
13: This task is not using native Ansible modules, but rather using the ‘shell’ module to execute a shell command. The command is ‘grubby’, which can be used to manipulate the kernel startup choice, which is what is executed here. In order to be able to use the variable used to install the kernel package, regular expressions are used to change the variable content.
17: This task too is simply is executing a shell command, which is doing a comparison of the running kernel (uname –r) versus the now set default startup kernel. This comparion tells if the running kernel is the same as the default kernel. If not ‘no’ is echo’ed to STDOUT, if it is the same, ‘yes’ is echo’ed to STDOUT. The execution data of this task is registered in variable ‘remain_running’.
22: This task is only executed when the ‘when’ line is true. The ‘when’ line reads ‘when: “’no’ in remain_running.stdout”’. This means that the task execution data variable remain_running is used, of which the .stdout property is the STDOUT of the task. So this means that if we determined that the kernels are not the same, and thus ‘no’ is echo’ed, this task should be executed, and otherwise is skipped. The task is again a shell execution ‘sleep 2; shutdown –r now “Ansible activate new kernel”. In other words: first the execution sleeps for 2 seconds, then a reboot (shutdown –r) is executed, which annotates ‘Ansible activate new kernel’ in the system log. Also, this task is executed asynchronously (async: 1), so the playbook execution continues.
29: This task too is only executed when remain_running.stdout indicates a reboot is required. Here a special action is executed: local_action. Local_action means the action is executed from the Ansible host, rather than on the destination host. This task keeps track of port 22 (22/tcp is the secure shell port) and waits until the port is not reachable anymore, which means the host has gone down.
39: This task too is only executed when remain_running.stdout indicates a reboot is required. Here also local_action is executed, and does the opposite of the previous task, which is monitor for for port 22 to be reachable again.

This playbook installs a kernel, sets it default, and then reboots any server that is not using the kernel specified, in order to have that specific kernel active.

2. Setup a filesystem in the logical volume manager. This playbook shows the high level in which the state can be specified, leaving the details to the Ansible modules:

```
01: ---
```

```
02: - hosts: all
03:   become: true
04:   tasks:
05:
06:   - name: create volume group vg_oracle using /dev/sdb
07:     lvg:
08:       vg: vg_oracle
09:       pvs: /dev/sdb
10:
11:   - name: create logical volume lv_oracle in vg_oracle
12:     lvol:
13:       vg: vg_oracle
14:       lv: lv_oracle
15:       size: 35g
16:
17:   - name: create filesystem in lv_oracle
18:     filesystem:
19:       fstype: xfs
20:       dev: /dev/vg_oracle/lv_oracle
21:
22:   - name: mount filesystem to /u01
23:     mount:
24:       name: /u01
25:       src: /dev/vg_oracle/lv_oracle
26:       state: mounted
27:       fstype: xfs
28:
29:   - name: set ownership correct for /u01 directory
30:     file:
31:       dest: /u01
32:       state: directory
33:       owner: oracle
34:       group: oinstall
```

6: the logical volume group module is used to create a physical volume from /dev/sdb and add it to the volume group vg_oracle.

11: the logical volume module is used to create the logical volume ‘lv_oracle’ sized of 35G from volume group ‘vg_oracle’.

17: the filesystem module is used to create an XFS filesystem in the logical volume ‘lv_oracle’.

22: the mount module is used to mount the logical volume ‘lv_oracle’ to the /u01 directory. Please mind this also makes the mount persistent in /etc/fstab.

29: the file module is used to change the default ownership of root for the /u01 directory/mountpoint to oracle.oinstall.

The descriptions of both Ansible runbooks should make clear that operational tasks can be executed using Ansible modules, which makes them understandable and easy to read.

The full runbooks of the presentation which shows linux configuration, installing and configuring Oracle database requirements, installing the database software and applying the general PSU as well as the JVM PSU and creating an archive of the installation, copying it locally, copying it back and installing the archive via the clone procedure are available at: https://gitlab.com/FritsHoogland/provisioning-presentation