IRONIC
A MODERN APPROACH TO MACHINE DEPLOYMENT
DEVANANDA VAN DER VEEN
@DEVANANDA

GITHUB.COM/DEVANANDA/TALKS
ABOUT

- Performance Consultant
- Engineer @ HP Cloud
- Likes interactive audiences
- Favorite whisky: Nikka
Observation:

Every company has their own PXE-based installation framework

Binary image copy improves repeatability and reduces entropy

So why not do this for bare metal, too?
Ironically, a physical machine behaves a lot like a VM or a container

*Based on the name, you might have expected that joke*
installer-based

User
Provisioning Service
DHCP service
TFTP service
Physical Host

request kickstart/preseed file

request installation media

unattended OS installer chooses drivers, apps, etc

request machine image

dd image to disk

request config data

hostname, SSH keys

write configdrive to disk

signals completion of installation

removes / changes config

reboot

optional, may include net info

cloud-init reads configdrive applies configuration
Build and customize your own images

$ pip install diskimage-builder

$ disk-image-create -o my-image -t qcow2 -a amd64 \  ubuntu vm serial-console cloud-init-datasources

"vm" element creates partition table and installs a bootloader

*Don't ask me why it's called "vm"*
You can build your own deploy ramdisk

$ pip install diskimage-builder

$ ramdisk-image-create -o my-ramdisk -a amd64 \ ubuntu deploy-ironic

Or download a kernel and ramdisk image from our build server

http://tarballs.openstack.org/ironic-python-agent/coreos/files
STANDARD PROTOCOLS

Power

- **IPMI**: intelligent platform management interface, for remote control of machine power state, boot device, serial console, etc.

- **SNMP**: simple network management protocol, often used with Power Distribution Units for remote control of power status.
STANDARD PROTOCOLS

Boot

- **DHCP**: dynamic host configuration protocol, used to locate the NBP on the network, and provide the host OS with IP address during init
- **TFTP**: trivial file transfer protocol, copies the NBP over the network
- **PXE**: pre-boot execution environment, allows host to boot from network
- **[g,i]PXE**: recent enhancements make PXE more flexible, supported on most hardware
IPMI HAS NOT SIGNIFICANTLY CHANGED IN THE LAST 10 YEARS

Meanwhile, vendors continue to add new (and different!) capabilities to their management controllers, each with different protocols.
A new standard is in the works (RedFish) but software will continue to change faster than hardware standards.
Vendor value is derived from quality of hardware, services, support, and integration not from proprietary solutions to common problems
ARCHITECTURE
SERVICE COMPONENTS
# DRIVER API

<table>
<thead>
<tr>
<th>Class</th>
<th>Method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerInterface</td>
<td>get_power_state(), set_power_state(), reboot()</td>
</tr>
<tr>
<td>ManagementInterface</td>
<td>get_supported_boot_devices(), get_boot_device(), set_boot_device(), get_sensors_data()</td>
</tr>
<tr>
<td>DeployInterface</td>
<td>prepare(), clean_up(), deploy(), tear_down(), take_over()</td>
</tr>
<tr>
<td>optional methods:</td>
<td>prepare_cleaning(), tear_down_cleaning()</td>
</tr>
<tr>
<td>ConsoleInterface</td>
<td>start_console(), stop_console(), get_console()</td>
</tr>
<tr>
<td>VendorInterface</td>
<td>validate(), driver_validate()</td>
</tr>
<tr>
<td>decorators:</td>
<td>@passthru, @driver_passthru</td>
</tr>
</tbody>
</table>
This *simplicity* gives driver authors a lot of *flexibility*.

While the REST API provides common abstraction for provisioning a pool of servers *repeatably* regardless of vendor.
REST API

Resource types: Node, Port, Driver (*)

docs.openstack.org/developer/ironic/webapi/v1.html

Documentation is continually built from source and packaged with each release

(*) There is a fourth resource type, "chassis". This is a remnant of early designs, and doesn't perform a meaningful function today
GET /v1/nodes/

```
{
    "nodes": [
    {
        "name": "nuc",
        "maintenance": false,
        "instance_uuid": null,
        "power_state": "power off",
        "uuid": "ba031dea-e7a8-4917-89f1-0f3ad31344ee",
        "provision_state": "available"
        "links": [
        {
            "href": "http://127.0.0.1:6385/v1/nodes/ba031dea-e7a8-4917-89f1-0f3ad31344ee",
            "rel": "self"
        },
        {
            "rel": "bookmark",
            "href": "http://127.0.0.1:6385/nodes/ba031dea-e7a8-4917-89f1-0f3ad31344ee"
        }
    }
    }
} ```
GET /v1/nodes/ba031dea-e7a8-4917-89f1-0f3ad31344ee

```json
{
    "name": "nuc",
    "uuid": "ba031dea-e7a8-4917-89f1-0f3ad31344ee",
    "driver": "pxe_amt",
    "properties": {
        "ram": 8096,
        "cpu_arch": "x86_64",
        "cpus": 2,
        "disk_size": 500
    },
    "driver_info": {
        "amt_password": "*****",
        "amt_address": "192.168.2.3",
        "amt_username": "admin"
    },
    "power_state": "power off",
    "target_power_state": null
}
```
Every driver is different and requires specific `driver_info` attributes.

You enter this once, when *enrolling* the Node.

Read the driver's documentation or discover it from the API.
GET /v1/drivers/drivername/properties

{
  "ilo_username": "username for the iLO with administrator privileges. Required.",
  "client_timeout": "timeout (in seconds) for iLO operations. Optional.",
  "ilo_address": "IP address or hostname of the iLO. Required.",
  "deploy_ramdisk": "UUID (from Glance) of the ramdisk that is mounted at boot time. Required.",
  "console_port": "node's UDP port to connect to. Only required for console access.",
  "ilo_change_password": "new password for iLO. Required if the clean step 'reset_iLO_credential' is enabled.",
  "deploy_kernel": "UUID (from Glance) of the deployment kernel. Required.",
  "client_port": "port to be used for iLO operations. Optional.",
  "ilo_password": "password for ilo_username. Required.",
}

{
  "snmp_outlet": "PDU power outlet index (1-based). Required.",
  "snmp_version": "SNMP protocol version: 1, 2c, 3 (optional, default 1).",
  "snmp_driver": "PDU manufacturer driver. Required.",
  "snmp_port": "SNMP port, default 161",
  "snmp_address": "PDU IP v4 address or hostname. Required.",
}
Instances are assumed to be different. Therefore, instance_info is cleared after instance deletion.
Vendors can implement additional capabilities which are passed directly to their driver.

These are implemented at:

/v1/drivers/NAME/vendor_passthru/
/v1/nodes/UUID/vendor_passthru/

In practice, this is little used, as drivers are encouraged to converge into a common API.
DEPLOYMENT SEQUENCE

This is just an example. Different drivers do things differently, after all.
Deployment with "agent" drivers

**API service**
- receives provisioning request
- passes request to conductor
- responds with HTTP 202 ACCEPTED

**Conductor service**
- validates requirements
- calls driver.prepare()
- prepares TFTP config

**HW Driver**
- this is driver specific
- attaches virtual media to BMC

**Host**
- success
- returns

**alt**
- PXE or IPXE driver
- ILO or IRMC driver
- other drivers may do different things, too
looks up Node UUID

API service

Conductor service

HW Driver

Host

returns

calls driver.deploy()

returns "in progress"

power on

boots Agent ramdisk

sends discovered MAC addresses

sends instance_info (image-source, etc)

download and write images to disk

sends completion notice

passes notice

calls finish deploy

power off

returns "deploy done"
PROVISIONING STATE MACHINE

STATE
- stable (or passive) state

R:verb
- request that begins a transition

[STATE*/TARGET]
- active, momentary, or error state
Progress is reflected in the API

```
"power_state" : "power on",                 # last known
"target_power_state" : null,                # non-null if
"provision_state" : "deploying",           # current pro
"target_provision_state" : "active",       # last request
"updated_at" : "2015-06-02T19:39:04+00:00",  # exposed tim
"reservation" : Leni,                      # exposed loc
```
PROVISIONING STATE MACHINE

(it's always more complicated than you expect)
KEEPING IT SIMPLE

Use Ansible for automation
Separate and reusable playbooks
First things first: input environment vars

$ cat bifrost/playbooks/inventory
---
node_default_network_interface: eth0
network_interface: eth2
ipv4_subnet_mask: 255.255.255.0
ipv4_gateway: 192.168.1.1
ipv4_nameserver: 8.8.8.8
dhcp_pool_start: 192.168.2.200
dhcp_pool_end: 192.168.2.250
deploy_kernel: "{{http_boot_folder}}/coreos_production_pxe.vmlinuz"
deploy_ramdisk: "{{http_boot_folder}}/coreos_production_pxe_image-oem"
deploy_image_filename: "deployment_image.qcow2"
deploy_image: "{{http_boot_folder}}/{{deploy_image_filename}}"
Install

installs external dependencies, configures your environment

$ bash ./scripts/env-setup.sh
$ source /opt/stack/ansible/hacking/env-setup
$ cd playbooks

$ ansible-playbook -K -vvvv -i inventory/localhost install.yml
Enroll
supply inventory file

$ ansible-playbook -vvvv -i inventory/localhost enroll.yml \
  -e baremetal_csv_file=baremetal.csv

OR

use CLI to create Nodes, Ports

$ ironic node-create -d agent_amttool -n nuc \
  -i amt_password='Pa$$w0rd' -i amt_address='192.168.2.3' -i amt_user= \n  -p cpu_arch=x86_64 -p local_gb=64 -p memory_mb=8192 -p cpus=2 \
  -i deploy_ramdisk='http://192.168.2.2:8080/coreos_production_pxe_\n  -i deploy_kernel='http://192.168.2.2:8080/coreos_production_pxe.\n
$ ironic port-create -n $UUID -a ec:a8:6b:fe:e1:b0
Deploy reads inventory file
OR

gathers list from Ironic directly

$ ansible-playbook -vvvv -i inventory/bifrost_inventory.py \ deploy-dynamic.yaml

then populates instance_info and starts the deploy
THANKS!

@devananda
devananda.github.io/talks
github.com/openstack/ironic
github.com/openstack/bifrost