Red Hat OpenStack Platform 13

파트너 통합

Red Hat OpenStack Platform 환경에서 타사 소프트웨어 및 하드웨어 통합 및 인증

Last Updated: 2023-04-08

Red Hat OpenStack Platform 13 파트너 통합

Red Hat OpenStack Platform 환경에서 타사 소프트웨어 및 하드웨어 통합 및 인증

OpenStack Team
rhos-docs@redhat.com

법적 공지

초록

이 가이드에서는 타사 구성 요소를 Red Hat OpenStack Platform 환경으로 통합하고 인증하는 방법에 대한 지침을 제공합니다. 여기에는 오버클라우드 이미지에 이러한 구성 요소를 추가하고, director를 사용하여 배포 구성을 생성하고, Red Hat Connect Build Service로 이러한 구성 요소를 인증하는 작업이 포함됩니다.

차례

1장. 귀하의 제3자 경쟁성을 통합해야 하는 이유

RHOSP(Red Hat OpenStack Platform)를 사용하여 솔루션을 RHOSP director와 통합할 수 있습니다. RHOSP director를 사용하여 RHOSP 환경의 배포 라이프사이클을 설치하고 관리합니다. 리소스를 최적화하고 배포 시간을 줄이며 라이프사이클 관리 비용을 줄일 수 있습니다.

RHOSP director 통합을 통해 기존 엔터프라이즈 관리 시스템 및 프로세스와 통합할 수 있습니다. CloudForms와 같은 Red Hat 제품은 director와의 통합을 확인하고 서비스 배포 관리를 위한 광범위한 노출을 제공할 것으로 예상됩니다.

1.1. 파트너 통합 사전 요구 사항

director를 사용하여 작업을 수행하려면 여러 사전 요구 사항을 충족해야 합니다. 파트너 통합의 목표는 Red Hat 엔지니어링, 파트너 관리자 및 지원 리소스를 기반으로 전체 통합에 대한 공유 이해를 만들어 협력하는 기술을 지원하는 것입니다.

Red Hat OpenStack Platform director와 함께 타사 구성 요소를 포함하려면 Red Hat OpenStack Platform으로 파트너 솔루션을 인증해야 합니다.

OpenStack 플러그인 인증 가이드

- Red Hat OpenStack Certification Policy Guide
- Red Hat OpenStack Certification Workflow 가이드

OpenStack Application Certification Guides

- Red Hat OpenStack Application Policy Guide
- Red Hat OpenStack Application Workflow 가이드

OpenStack Bare Metal 인증 가이드

- Red Hat OpenStack Platform 하드웨어 베어 메탈 인증 정책 가이드
- Red Hat OpenStack Platform 하드웨어 베어 메탈 인증 워크플로 가이드

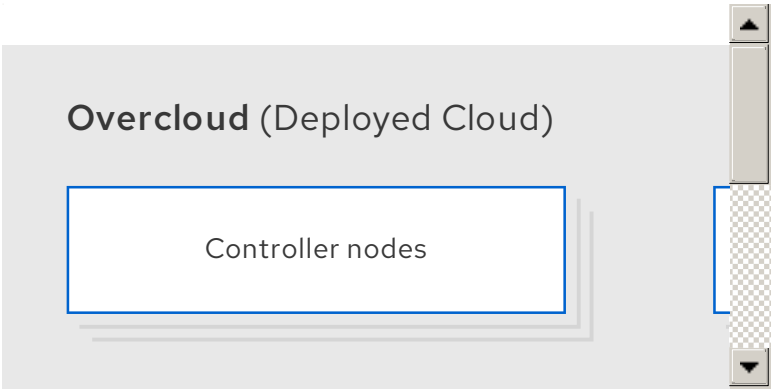
2장. DIRECTOR 아키텍처

Red Hat OpenStack Platform director는 OpenStack API를 사용하여 RHOSP(Red Hat OpenStack Platform) 환경을 구성, 배포 및 관리합니다. 즉, director와 통합하려면 이러한 OpenStack API 및 지원 구성 요소와 통합되어야 합니다. 이러한 API의 이점은 광범위한 통합 테스트 업스트림을 통해 잘 문서화되어 있으며 더욱 성숙하고 director가 RHOSP에 대한 기본 지식이 있는 사용자가 보다 쉽게 작동하는 방식을 이해할 수 있다는 것입니다. director는 핵심 OpenStack 기능 개선, 보안 패치 및 버그 수정을 자동으로 상속합니다.

director는 전체 RHOSP 환경을 설치하고 관리하는 데 사용하는 툴셋입니다. "OpenStack-On-OpenStack"의 약어인 OpenStack 프로젝트 TripleO를 주로 기반으로 합니다. 이 프로젝트는 RHOSP 구성 요소를 사용하여 완전히 작동하는 RHOSP 환경을 설치합니다. 여기에는 OpenStack 노드로 사용할 베어 메탈 시스템을 프로비저닝하고 제어하는 새로운 OpenStack 구성 요소가 포함됩니다. 이를 통해 가볍고 강력한 완전한 RHOSP 환경을 설치할 수 있는 간단한 방법을 제공합니다.

director는 언더클라우드(undercloud)와 오버클라우드(overcloud)의 두 가지 주요 개념을 사용합니다. director는 언더클라우드라고도 하는 단일 시스템 OpenStack 환경을 구성하는 OpenStack 구성 요소의 하위 집합입니다. 언더클라우드는 워크로드를 실행할 프로덕션 수준 클라우드를 생성할 수 있는 관리 시스템 역할을 합니다. 이 프로덕션 수준의 클라우드는 오버클라우드입니다. 오버클라우드 및 Undercloud에 대한 자세한 내용은 *Director Installation and Usage* 가이드를 참조하십시오.

그림 2.1. 언더클라우드 및 오버클라우드 아키텍처



director에는 오버클라우드 구성을 생성하는 데 사용할 수 있는 툴, 유틸리티 및 예제 템플릿이 포함되어 있습니다. director는 구성 데이터, 매개변수 및 네트워크 토폴로지 정보를 캡처하고 이 정보를 ironic, heat, Puppet과 같은 구성 요소와 함께 사용하여 오버클라우드 설치를 오케스트레이션합니다.

2.1. 핵심 구성 요소 및 오버클라우드

다음은 Red Hat OpenStack Platform director의 핵심 구성 요소이며 오버클라우드 생성에 기여합니다.

- OpenStack Bare Metal Provisioning 서비스(ironic)
- OpenStack Orchestration 서비스(heat)
- Puppet
- tripleo 및 TripleO heat 템플릿
- 구성 가능 서비스
- 컨테이너화된 서비스 및 Kolla
- Ansible

2.1.1. OpenStack Bare Metal Provisioning 서비스(ironic)

베어 메탈 프로비저닝 서비스에서는 셀프 서비스 프로비저닝을 통해 최종 사용자에게 전용 베어 메탈 호스트를 제공합니다. director는 베어 메탈 프로비저닝을 사용하여 오버클라우드에서 베어 메탈 하드웨어의 라이프사이클을 관리합니다. Bare Metal Provisioning은 자체 API를 사용하여 베어 메탈 노드를 정의합니다.

director를 사용하여 OpenStack 환경을 프로비저닝하려면 특정 드라이버를 사용하여 베어 메탈 프로비저닝에 노드를 등록해야 합니다. 대부분의 하드웨어에는 IPMI 전원 관리 기능에 대한 지원이 포함되어 있으므로 주요 지원되는 드라이버는 IPMI(Intelligent Platform Management Interface)입니다. 그러나 베어 메탈 프로비저닝에는 HP iLO, Cisco UCS 또는 Dell DRAC와 같은 공급업체별 동등한 기능도 포함되어 있습니다.

Bare Metal Provisioning은 노드의 전원 관리를 제어하고 인트로스펙션 메커니즘을 사용하여 하드웨어 정보 또는 팩트를 수집합니다. director는 인트로스펙션 프로세스의 정보를 사용하여 컨트롤러 노드, 컴퓨팅 노드, Storage 노드와 같은 다양한 OpenStack 환경 역할과 노드를 일치시킵니다. 예를 들어 10개의 디스크가 있는 검색된 노드는 일반적으로 스토리지 노드로 프로비저닝됩니다.

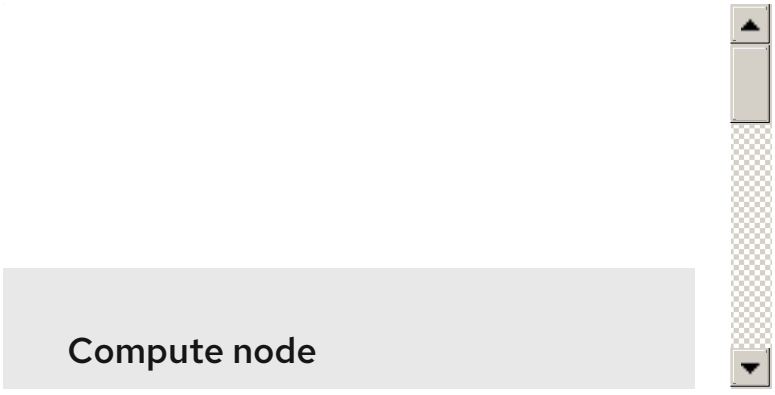
그림 2.2. Bare Metal Provisioning 서비스를 사용하여 노드의 전원 관리를 제어합니다.

하드웨어에 대한 director를 지원하려면 베어 메탈 프로비저닝 서비스에 드라이버를 적용해야 합니다.

2.1.2. heat

Heat는 애플리케이션 스택 오케스트레이션 엔진입니다. heat를 사용하여 클라우드에 배포하기 전에 애플리케이션의 요소를 정의할 수 있습니다. 구성 매개 변수를 사용하여 여러 인프라 리소스(예: 인스턴스, 네트워크, 스토리지 볼륨, 탄력적 IP)를 포함하는 스택 템플릿을 생성합니다. heat를 사용하여 지정된 종속성 체인을 기반으로 이러한 리소스를 생성하고, 가용성에 대한 리소스를 모니터링하고, 필요한 경우 스케일링합니다. 이러한 템플릿을 사용하여 애플리케이션 스택을 이식하고 반복 가능한 결과를 얻을 수 있습니다.

그림 2.3. heat 서비스를 사용하여 클라우드에 배포하기 전에 애플리케이션의 요소를 정의합니다.



director는 기본 OpenStack heat API를 사용하여 오버클라우드 배포와 관련된 리소스를 프로비저닝하고 관리합니다. 여기에는 노드 역할별로 프로비저닝할 노드 수 정의, 각 노드에 구성할 소프트웨어 구성 요소, director에서 이러한 구성 요소 및 노드 유형을 구성하는 순서와 같은 정확한 세부 정보가 포함됩니다. director는 또한 heat를 사용하여 배포 문제를 해결하고 배포 후 변경합니다.

다음 예제는 컨트롤러 노드의 매개변수를 정의하는 heat 템플릿의 스니펫입니다.

```
NeutronExternalNetworkBridge:
  description: Name of bridge used for external network traffic.
  type: string
  default: 'br-ex'
NeutronBridgeMappings:
  description: >
    The OVS logical->physical bridge mappings to use. See the Neutron
    documentation for details. Defaults to mapping br-ex - the external
    bridge on hosts - to a physical name 'datacentre' which can be used
    to create provider networks (and we use this for the default floating
    network) - if changing this either use different post-install network
    scripts or be sure to keep 'datacentre' as a mapping network name.
  type: string
  default: "datacentre:br-ex"
```

Heat는 director에 포함된 템플릿을 사용하여 오버클라우드 생성을 용이하게 하며, 이 템플릿을 사용하면 노드의 전원을 켜기 위해 ironic 호출이 포함됩니다. 표준 heat 툴을 사용하여 진행 중인 오버클라우드의 리소스 및 상태를 확인할 수 있습니다. 예를 들어 heat 툴을 사용하여 오버클라우드를 중첩 애플리케이션 스택으로 표시할 수 있습니다. heat 템플릿의 구문을 사용하여 프로덕션 OpenStack 클라우드를 선언하고 생성합니다. 모든 파트너 통합 사용 사례에는 heat 템플릿이 필요하므로 파트너 통합을 위한 사전 이해와 실력이 있어야 합니다.

2.1.3. Puppet

Puppet은 시스템의 최종 상태를 설명하고 유지 관리하는 데 사용할 수 있는 구성 관리 및 시행 도구입니다. 이 종료 상태를 Puppet 매니페스트로 정의합니다. Puppet은 다음 두 가지 모델을 지원합니다.

- 매니페스트 형식의 지침을 로컬로 실행하는 독립 실행형 모드
- Puppet이 Puppet 마스터라는 중앙 서버에서 매니페스트를 검색하는 서버 모드

다음 두 가지 방법으로 변경할 수 있습니다.

- 노드에 새 매니페스트를 업로드하고 로컬에서 실행합니다.
- Puppet 마스터의 클라이언트/서버 모델에서 수정합니다.

director는 다음 영역에서 Puppet을 사용합니다.

- 언더클라우드 호스트에서 **undercloud.conf** 파일의 설정에 따라 패키지를 설치하고 구성합니다.
- **openstack-puppet-modules** 패키지를 기본 오버클라우드 이미지에 삽입하면 Puppet 모듈에서 배포 후 설정을 위해 준비됩니다. 기본적으로 각 노드의 모든 OpenStack 서비스가 포함된 이미지를 생성합니다.
- 노드에 추가 Puppet 매니페스트 및 heat 매개변수를 제공하고 오버클라우드 배포 후 설정을 적용합니다. 여기에는 노드 유형에 따라 구성을 활성화하고 시작하는 서비스가 포함됩니다.
- 노드에 Puppet hieradata 제공. Puppet 모듈과 매니페스트는 사이트 또는 노드별 매개변수에서

무료로 제공되어 매니페스트를 일관되게 유지합니다. hieradata는 Puppet 모듈로 푸시하고 다른 영역에서 참조할 수 있는 매개 변수화된 값의 형태로 작동합니다. 예를 들어 매니페스트 내에서 MySQL 암호를 참조하려면 이 정보를 hieradata로 저장하고 매니페스트 내에서 참조합니다. hieradata를 보려면 다음 명령을 입력합니다.

```
[root@localhost ~]# grep mysql_root_password hieradata.yaml # View the data in the hieradata file
openstack::controller::mysql_root_password: 'redhat123'
```

Puppet 매니페스트에서 hieradata를 참조하려면 다음 명령을 입력합니다.

```
[root@localhost ~]# grep mysql_root_password example.pp # Now referenced in the Puppet manifest
mysql_root_password  => hiera('openstack::controller::mysql_root_password')
```

패키지 설치 및 서비스 사용이 필요한 파트너 통합 서비스는 요구 사항을 충족하는 Puppet 모듈을 생성할 수 있습니다. 현재 OpenStack Puppet 모듈 및 예제를 가져오는 방법에 대한 자세한 내용은 4.2절. "OpenStack Puppet 모듈 가져오기" 을 참조하십시오.

2.1.4. tripleo 및 TripleO heat 템플릿

director는 업스트림 TripleO 프로젝트를 기반으로 합니다. 이 프로젝트는 OpenStack 서비스 세트를 다음 목표와 결합합니다.

- Image 서비스(glance)를 사용하여 오버클라우드 이미지 저장
- 오케스트레이션 서비스(heat)를 사용하여 오버클라우드 오케스트레이션
- Bare Metal Provisioning(ironic) 및 Compute(nova) 서비스를 사용하여 베어 메탈 머신 프로비저닝

tripleo에는 Red Hat에서 지원하는 오버클라우드 환경을 정의하는 heat 템플릿 컬렉션도 포함되어 있습니다. heat를 사용하여 director는 이 템플릿 컬렉션을 읽고 오버클라우드 스택을 오케스트레이션합니다.

2.1.5. 구성 가능 서비스

Red Hat OpenStack Platform의 각 측면은 구성 가능 서비스로 나뉩니다. 즉, 다양한 서비스 조합을 사용하는 다양한 역할을 정의할 수 있습니다. 예를 들어 네트워킹 에이전트를 기본 컨트롤러 노드에서 독립 실행형 Networker 노드로 이동할 수 있습니다.

구성 가능 서비스 아키텍처에 대한 자세한 내용은 6장. *구성 가능 서비스* 을 참조하십시오.

2.1.6. 컨테이너화된 서비스 및 Kolla

각 주요 RHOSP(Red Hat OpenStack Platform) 서비스는 컨테이너에서 실행됩니다. 이를 통해 각 서비스를 호스트와 분리된 자체 네임스페이스 내에서 분리한 상태로 유지할 수 있습니다. 다음과 같은 효과가 있습니다.

- 배포 중에 RHOSP는 Red Hat 고객 포털에서 컨테이너 이미지를 가져와서 실행합니다.
- **podman** 명령은 서비스 시작 및 중지와 같은 관리 기능을 수행합니다.
- 컨테이너를 업그레이드하려면 새 컨테이너 이미지를 가져와 기존 컨테이너를 최신 버전으로 교체해야 합니다.

Red Hat OpenStack Platform은 **Kolla** 툴셋으로 빌드 및 관리되는 컨테이너 세트를 사용합니다.

2.1.7. Ansible

Red Hat OpenStack Platform은 구성 가능 서비스 업그레이드와 관련된 특정 기능을 수행하기 위해 Ansible을 사용합니다. 여기에는 서비스 시작 및 중지 및 데이터베이스 업그레이드 수행과 같은 기능이 포함됩니다. 이러한 업그레이드 작업은 구성 가능 서비스 템플릿 내에 정의됩니다.

3장. 오버클라우드 이미지 작업

RHOSP(Red Hat OpenStack Platform) director는 오버클라우드의 이미지를 제공합니다. 이 컬렉션의 QCOW 이미지에는 컴퓨팅, 컨트롤러 및 스토리지 노드와 같은 다양한 오버클라우드 역할을 형성하기 위해 통합하는 기본 소프트웨어 구성 요소 세트가 포함되어 있습니다. 노드에 추가 구성 요소를 설치하는 등 필요에 맞게 오버클라우드 이미지의 특정 요소를 수정하는 것이 좋습니다.

virt-customize 툴을 사용하여 기존 오버클라우드 이미지를 수정하여 기존 컨트롤러 노드를 보강할 수 있습니다. 예를 들어 다음 절차를 사용하여 초기 이미지와 함께 제공되지 않는 추가 **ml2** 플러그인, Cinder 백엔드 또는 모니터링 에이전트를 설치합니다.

중요

타사 소프트웨어를 포함하도록 오버클라우드 이미지를 수정하고 문제가 보고되면 Red Hat에서 일반 타사 지원 정책에 따라 수정되지 않은 이미지로 문제를 재현하도록 요청할 수 있습니다. https://access.redhat.com/articles/1067.

3.1. 오버클라우드 이미지 가져오기

director에는 오버클라우드 노드를 프로비저닝하기 위한 여러 개의 디스크 이미지가 필요합니다.

- **인트로스펙션 커널 및 램디스크 - PXE 부팅을 통한**베어 메탈 시스템 인트로스펙션의 경우.
- **배포 커널 및 램디스크** - 시스템 프로비저닝 및 배포용.
- **오버클라우드 커널, 램디스크 및 전체 이미지** - director가 노드의 하드 디스크에 쓰는 기본 오버클라우드 시스템입니다.

절차

1. 이러한 이미지를 얻으려면 **rhosp-director-images** 및 **rhosp-director-images-ipa** 패키지를 설치합니다.

```
$ sudo yum install rhosp-director-images rhosp-director-images-ipa
```

2. **stack** 사용자 홈, **/home/stack/ images 의 images 디렉터리에 압축 파일을 풉** 니다.

```
$ cd ~/images
$ for i in /usr/share/rhosp-director-images/overcloud-full-latest-13.0.tar /usr/share/rhosp-director-images/ironic-python-agent-latest-13.0.tar; do tar -xvf $i; done
```

3.2. INITRD: 초기 램디스크 수정

초기 램디스크를 수정해야 하는 경우도 있습니다. 예를 들어 인트로스펙션 또는 프로비저닝 프로세스 중에 노드를 부팅할 때 특정 드라이버를 사용할 수 있어야 할 수 있습니다. 오버클라우드 컨텍스트에서 여기에는 다음 램디스크 중 하나가 포함됩니다.

- 인트로스펙션 ramdisk - **ironic-python-agent.initramfs**
- 프로비저닝 ramdisk - **overcloud-full.initrd**

이 절차에서는 예제로 **ironic-python-agent.initramfs** ramdisk에 RPM 패키지를 추가합니다.

절차

1. **root** 사용자로 로그인하여 ramdisk에 대한 임시 디렉터리를 생성합니다.

```
# mkdir ~/ipa-tmp
# cd ~/ipa-tmp
```

2. **skipcpio** 및 **cpio** 명령을 사용하여 ramdisk를 임시 디렉터리로 추출합니다.

```
# /usr/lib/dracut/skipcpio ~/images/ironic-python-agent.initramfs | zcat | cpio -ivd | pax -r
```

3. 추출된 콘텐츠에 RPM 패키지를 설치합니다.

```
# rpm2cpio ~/RPMs/python-proliantutils-2.1.7-1.el7ost.noarch.rpm | pax -r
```

4. 새 ramdisk를 다시 생성합니다.

```
# find . 2>/dev/null | cpio --quiet -c -o | gzip -8  > /home/stack/images/ironic-python-agent.initramfs
# chown stack: /home/stack/images/ironic-python-agent.initramfs
```

5. 새 패키지가 ramdisk에 있는지 확인합니다.

```
# lsinitrd /home/stack/images/ironic-python-agent.initramfs | grep proliant
```

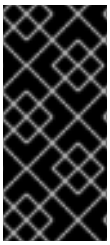
3.3. QCOW: VIRT-CUSTOMIZE를 DIRECTOR에 설치

libguestfs-tools 패키지에는 **virt-customize** 툴이 포함되어 있습니다.

절차

- **rhel-8-for-x86_64-appstream-eus-rpms** 리포지토리에서 **libguestfs-tools** 를 설치합니다.

```
$ sudo yum install libguestfs-tools
```



중요

언더클라우드에 **libguestfs-tools** 패키지를 설치하는 경우 언더클라우드에서 **tripleo_iscsid** 서비스와의 포트 충돌을 방지하려면 **iscsid.socket** 을 비활성화합니다.

```
$ sudo systemctl disable --now iscsid.socket
```

3.4. QCOW: 오버클라우드 이미지 검사

overcloud-full.qcow2 이미지의 내용을 검토하려면 이 이미지를 사용하는 가상 머신을 생성해야 합니다.

절차

1. **overcloud-full.qcow2** 이미지를 사용하는 가상 머신 인스턴스를 생성하려면 **guestmount** 명령을 사용합니다.

```
$ mkdir ~/overcloud-full
$ guestmount -a overcloud-full.qcow2 -i --ro ~/overcloud-full
```

~/overcloud-full 에서 QCOW2 이미지의 내용을 검토할 수 있습니다.

2. 또는 virt-manager를 사용하여 다음 부팅 옵션으로 가상 머신을 생성할 수도 있습니다.
   - **커널 경로:** /overcloud-full.vmlinuz
   - **initrd 경로:** /overcloud-full.initrd
   - **커널 인수:** root=/dev/sda

3.5. QCOW: 루트 암호 설정

콘솔을 통해 노드에 대한 관리자 수준 액세스를 제공하도록 루트 암호를 설정합니다.

절차

- 이미지에서 **root** 사용자의 암호를 설정합니다.

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --root-password password:test
[  0.0] Examining the guest ...
[ 18.0] Setting a random seed
[ 18.0] Setting passwords
[ 19.0] Finishing off
```

3.6. QCOW: 이미지 등록

오버클라우드 이미지를 Red Hat Content Delivery Network에 등록합니다.

절차

1. 사용자 정의와 관련된 Red Hat 리포지토리를 활성화하려면 이미지를 임시로 등록합니다.

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager register --username=[username] --password=[password]'
[  0.0] Examining the guest ...
[ 10.0] Setting a random seed
[ 10.0] Running: subscription-manager register --username=[username] --password=
[password]
[ 24.0] Finishing off
```

2. **[username]** 및 **[password]** 를 Red Hat 고객 계정 정보로 교체합니다. 이렇게 하면 이미지에서 다음 명령이 실행됩니다.

```
subscription-manager register --username=[username] --password=[password]
```

3.7. QCOW: 서브스크립션 연결 및 RED HAT 리포지토리 활성화

절차

1. 계정 서브스크립션에서 풀 ID 목록을 찾습니다.

```
$ sudo subscription-manager list
```

2. 서브스크립션 풀 ID를 선택하고 이미지에 연결합니다.

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager attach --pool [subscription-pool]'
[  0.0] Examining the guest ...
[ 12.0] Setting a random seed
[ 12.0] Running: subscription-manager attach --pool [subscription-pool]
[ 52.0] Finishing off
```

3. **[subscription-pool]** 을 선택한 서브스크립션 풀 ID로 교체합니다.

```
subscription-manager attach --pool [subscription-pool]
```

그러면 리포지토리를 활성화할 수 있도록 풀에 풀이 추가됩니다.

4. Red Hat 리포지토리를 활성화합니다.

```
$ subscription-manager repos --enable=[repo-id]
```

3.8. QCOW: 사용자 정의 리포지토리 파일 복사

타사 소프트웨어를 이미지에 추가하려면 추가 리포지토리가 필요합니다. 다음은 OpenDaylight 리포지토리 콘텐츠를 사용하는 구성이 포함된 리포지토리 파일 예입니다.

절차

1. **opendaylight.repo** 파일의 내용을 나열합니다.

```
$ cat opendaylight.repo

[opendaylight]
name=OpenDaylight Repository
baseurl=https://nexus.opendaylight.org/content/repositories/opendaylight-yum-epel-6-
x86_64/
gpgcheck=0
```

2. 의 리포지토리 파일을 이미지에 복사합니다.

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --upload
opendaylight.repo:/etc/yum.repos.d/
[  0.0] Examining the guest ...
[ 12.0] Setting a random seed
[ 12.0] Copying: opendaylight.repo to /etc/yum.repos.d/
[ 13.0] Finishing off
```

--upload 옵션은 리포지토리 파일을 오버클라우드 이미지의 **/etc/yum.repos.d/** 에 복사합니다.

중요: Red Hat은 인증되지 않은 벤더의 소프트웨어에 대한 지원을 제공하지 않습니다. 설치하려는 소프트웨어가 지원되는 Red Hat 지원 담당자에게 확인하십시오.

3.9. QCOW: RPM 설치

절차

- **virt-customize** 명령을 사용하여 이미지에 패키지를 설치합니다.

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --install opendaylight
[  0.0] Examining the guest ...
[ 11.0] Setting a random seed
[ 11.0] Installing packages: opendaylight
[ 91.0] Finishing off
```

--install 옵션을 사용하여 설치할 패키지를 지정합니다.

3.10. QCOW: 서브스크립션 풀 정리

절차

- 이미지를 사용자 지정하는 데 필요한 패키지를 설치한 후 서브스크립션 풀을 제거하고 이미지를 등록 취소합니다.

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager remove --all'
[  0.0] Examining the guest ...
[ 12.0] Setting a random seed
[ 12.0] Running: subscription-manager remove --all
[ 18.0] Finishing off
```

3.11. QCOW: 이미지 등록 취소

절차

- 오버클라우드 배포 프로세스에서 노드에 이미지를 배포하고 각각 등록할 수 있도록 이미지 등록을 해제합니다.

```
$ virt-customize --selinux-relabel -a overcloud-full.qcow2 --run-command 'subscription-
manager unregister'
[  0.0] Examining the guest ...
[ 11.0] Setting a random seed
[ 11.0] Running: subscription-manager unregister
[ 17.0] Finishing off
```

3.12. QCOW: 머신 ID 재설정

절차

- 이 이미지를 사용하는 시스템에서 중복 머신 ID를 사용하지 않도록 이미지의 시스템 ID를 재설정합니다.

```
$ virt-sysprep --operation machine-id -a overcloud-full.qcow2
```

3.13. DIRECTOR에 이미지 업로드

이미지를 수정한 후 director에 업로드해야 합니다.

절차

1. 명령줄에서 director **에** 액세스할 수 있도록 stackrc 파일을 소싱합니다.

```
$ source stackrc
```

2. 오버클라우드 배포에 사용할 기본 director 이미지를 업로드합니다.

```
$ openstack overcloud image upload --image-path /home/stack/images/
```

이렇게 하면 다음 이미지가 director에 업로드됩니다.

- bm-deploy-kernel
- bm-deploy-ramdisk
- overcloud-full
- overcloud-full-initrd
- overcloud-full-vmlinuz
  이 스크립트는 director의 PXE 서버에 인트로스펙션 이미지도 설치합니다.

3. CLI에서 이미지 목록을 확인합니다.

```
$ openstack image list
+--------------------------------------+------------------------+
| ID                                   | Name                   |
+--------------------------------------+------------------------+
| 765a46af-4417-4592-91e5-a300ead3faf6 | bm-deploy-ramdisk      |
| 09b40e3d-0382-4925-a356-3a4b4f36b514 | bm-deploy-kernel       |
| ef793cd0-e65c-456a-a675-63cd57610bd5 | overcloud-full         |
| 9a51a6cb-4670-40de-b64b-b70f4dd44152 | overcloud-full-initrd  |
| 4f7e33f4-d617-47c1-b36f-cbe90f132e5d | overcloud-full-vmlinuz |
+--------------------------------------+------------------------+
```

이 목록에는 인트로스펙션 PXE 이미지(agent.*)가 표시되지 않습니다. director는 이러한 파일을 **/httpboot** 에 복사합니다.

```
[stack@host1 ~]$ ls /httpboot -l
total 151636
-rw-r--r--. 1 ironic ironic       269 Sep 19 02:43 boot.ipxe
-rw-r--r--. 1 root   root         252 Sep 10 15:35 inspector.ipxe
-rwxr-xr-x. 1 root   root     5027584 Sep 10 16:32 agent.kernel
-rw-r--r--. 1 root   root   150230861 Sep 10 16:32 agent.ramdisk
drwxr-xr-x. 2 ironic ironic      4096 Sep 19 02:45 pxelinux.cfg
```

4장. OPENSTACK PUPPET 모듈에 추가 구성

이 장에서는 OpenStack Puppet 모듈에 추가하는 방법을 살펴봅니다. 여기에는 Puppet 모듈 개발에 대한 몇 가지 기본 지침이 포함됩니다.

4.1. PUPPET 구문 및 모듈 구조

다음 섹션에서는 Puppet 구문 및 Puppet 모듈의 구조를 이해하는 데 도움이 되는 몇 가지 기본 사항을 제공합니다.

4.1.1. Puppet 모듈 분석

OpenStack 모듈에 기여하기 전에 Puppet 모듈을 생성하는 구성 요소를 이해해야 합니다.

CrashLoopBackOffmanifests

매니페스트는 리소스 집합 및 해당 속성을 정의하는 코드가 포함된 파일입니다. 리소스는 시스템의 구성 가능한 부분입니다. 리소스의 예로는 패키지, 서비스, 파일, 사용자, 그룹, SELinux 구성, SSH 키 인증, cron 작업 등이 있습니다. 매니페스트는 특성에 키-값 쌍 세트를 사용하여 각 필수 리소스를 정의합니다.

```
package { 'httpd':
  ensure => installed,
}
```

예를 들어 이 선언은 **httpd** 패키지가 설치되었는지 확인합니다. 그렇지 않은 경우 매니페스트는 **dnf** 를 실행하고 설치합니다. 매니페스트는 모듈의 매니페스트 디렉터리에 있습니다. Puppet 모듈은 테스트 매니페스트에 테스트 디렉터리를 사용합니다. 이러한 매니페스트는 공식 매니페스트에 있는 특정 클래스를 테스트하는 데 사용됩니다.

클래스

클래스는 매니페스트에 여러 리소스를 통합합니다. 예를 들어 HTTP 서버를 설치하고 구성하는 경우 세 가지 리소스(HTTP 서버 패키지를 설치하는 방법, HTTP 서버를 구성하는 방법, 서버를 시작하거나 활성화하는 클래스)를 생성할 수 있습니다. 또한 해당 구성에 적용되는 다른 모듈의 클래스를 참조할 수도 있습니다. 예를 들어 웹 서버도 필요한 애플리케이션을 구성하려면 HTTP 서버에 대해 이전에 언급된 클래스를 참조할 수 있습니다.

정적 파일

모듈에는 Puppet에서 시스템의 특정 위치에 복사할 수 있는 정적 파일이 포함될 수 있습니다. 매니페스트에서 파일 리소스 선언을 사용하여 위치 및 권한과 같은 기타 속성을 정의합니다.
정적 파일은 모듈의 파일 디렉터리에 있습니다.

behindtemplates

구성 파일에 사용자 지정 콘텐츠가 필요한 경우가 있습니다. 이 경우 사용자는 정적 파일 대신 템플릿을 생성합니다. 정적 파일과 마찬가지로 템플릿은 매니페스트에 정의되어 시스템의 위치에 복사됩니다. 차이점은 템플릿을 통해 Ruby 표현식이 사용자 정의 콘텐츠 및 변수 입력을 정의할 수 있다는 것입니다. 예를 들어 사용자 지정 가능한 포트를 사용하여 httpd를 구성하려면 구성 파일의 템플릿에 다음이 포함됩니다.

```
Listen <%= @httpd_port %>
```

이 경우 **httpd_port** 변수는 이 템플릿을 참조하는 매니페스트에 정의되어 있습니다.

템플릿은 모듈의 템플릿 디렉터리에 있습니다.

Plugins

Puppet의 핵심 기능을 넘어 확장할 수 있는 측면에 대한 플러그인을 사용합니다. 예를 들어 플러그인을 사용하여 사용자 지정 팩트, 사용자 지정 리소스 또는 새 함수를 정의할 수 있습니다. 예를 들어 데이터베이스 관리자에게 PostgreSQL 데이터베이스의 리소스 유형이 필요할 수 있습니다. 데이터베이스 관리자가 PostgreSQL을 설치한 후 새 데이터베이스 세트로 PostgreSQL을 채울 수 있습니다. 따라서 데이터베이스 관리자는 PostgreSQL 설치 및 데이터베이스 생성 후 생성되도록 하는 Puppet 매니페스트만 생성해야 합니다.
플러그인은 모듈의 lib 디렉터리에 있습니다. 여기에는 플러그인 유형에 따라 하위 디렉터리 세트가 포함됩니다.

- **/lib/facter** - 사용자 지정 팩트의 위치입니다.
- **/lib/puppet/type** - 특성의 키-값 쌍을 간략하게 설명하는 사용자 정의 리소스 유형 정의 위치입니다.
- **/lib/puppet/provider** - 리소스를 제어하는 데 리소스 유형 정의와 함께 사용되는 사용자 정의 리소스 공급자의 위치입니다.
- **/lib/puppet/parser/functions** - 사용자 정의 함수의 위치입니다.

4.1.2. 서비스 설치

일부 소프트웨어에는 패키지 설치가 필요합니다. 이는 Puppet 모듈이 수행할 수 있는 하나의 기능입니다. 이를 위해서는 특정 패키지에 대한 구성을 정의하는 리소스 정의가 필요합니다.

예를 들어 **mymodule** 모듈을 통해 **httpd** 패키지를 설치하려면 **mymodule** 모듈의 Puppet 매니페스트에 다음 콘텐츠를 추가합니다.

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
}
```

이 코드는 **httpd** 라는 **mymodule** 의 하위 클래스를 정의한 다음 **httpd** 패키지에 대한 패키지 리소스 선언을 정의합니다. **ensure => installed** 속성은 패키지가 설치되었는지 확인하도록 Puppet에 지시합니다. 설치되지 않은 경우 Puppet은 **yum** 을 실행하여 설치합니다.

4.1.3. 서비스 시작 및 활성화

패키지를 설치한 후 서비스를 시작할 수 있습니다. **service** 라는 다른 리소스 선언을 사용합니다. 다음 콘텐츠를 사용하여 매니페스트를 편집합니다.

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
  service { 'httpd':
    ensure => running,
    enable => true,
    require => Package["httpd"],
  }
}
```

결과는 다음과 같습니다.

- **ensure => running** 속성은 서비스가 실행 중인지 확인합니다. 그렇지 않으면 Puppet에서 활성화합니다.
- **enable => true** 특성은 시스템이 부팅될 때 서비스가 실행되도록 설정합니다.
- **require => Package["httpd"]** 속성은 하나의 리소스 선언과 다른 리소스 선언 간의 순서 관계를 정의합니다. 이 경우 **httpd** 패키지를 설치한 후 **httpd** 서비스가 시작되는지 확인합니다. 이렇게 하면 서비스와 해당 패키지 간에 종속성이 생성됩니다.

4.1.4. 서비스 구성

HTTP 서버는 포트 80에서 웹 호스트를 제공하는 **/etc/httpd/conf/httpd.conf** 에 몇 가지 기본 구성을 제공합니다. 그러나 사용자 지정 포트에 추가 웹 호스트를 제공하기 위해 구성을 추가할 수 있습니다.

절차

1. 사용자 정의 포트에 변수 입력이 필요하므로 템플릿 파일을 사용하여 HTTP 구성 파일을 저장해야 합니다. 모듈 템플릿 디렉터리에서 다음 내용이 포함된 **myserver.conf.erb** 파일을 추가합니다.

```
Listen <%= @httpd_port %>
NameVirtualHost *:<%= @httpd_port %>
<VirtualHost *:<%= @httpd_port %>>
  DocumentRoot /var/www/myserver/
  ServerName *:<%= @fqdn %>>
  <Directory "/var/www/myserver/">
    Options All Indexes FollowSymLinks
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>
```

   이 템플릿은 Apache 웹 서버 구성의 표준 구문을 따릅니다. 유일한 차이점은 Ruby 이스케이프 문자를 포함하여 모듈에서 변수를 삽입하는 것입니다. 예를 들어 웹 서버 포트를 지정하는 데 사용하는 **httpd_port** 입니다.

   fqdn 이 포함된 변수는 시스템의 정규화된 도메인 이름을 저장하는 변수입니다. 이것을 **시스템 팩트** 라고 합니다. 시스템 팩트는 시스템의 각 Puppet 카탈로그를 생성하기 전에 각 시스템에서 수집됩니다. Puppet은 **facter** 명령을 사용하여 이러한 시스템 팩트를 수집하고 팩트를 실행하여 이러한 팩트 목록을 볼 수도 있습니다.

2. **myserver.conf.erb** 를 저장합니다.
3. 모듈의 Puppet 매니페스트에 리소스를 추가합니다.

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
  service { 'httpd':
    ensure => running,
    enable => true,
    require => Package["httpd"],
  }
```

```
  file {'/etc/httpd/conf.d/myserver.conf':
  notify => Service["httpd"],
    ensure => file,
    require => Package["httpd"],
    content => template("mymodule/myserver.conf.erb"),
  }
  file { "/var/www/myserver":
    ensure => "directory",
  }
}
```

결과는 다음과 같습니다.

- 서버 구성 파일 (**/etc/httpd/conf.d/myserver.conf**)에 대한 파일 리소스 선언을 추가합니다. 이 파일의 내용은 생성한 **myserver.conf.erb** 템플릿입니다.
- 이 파일을 추가하기 전에 **httpd** 패키지가 설치되었는지 확인합니다.
- 웹 서버에 대한 디렉터리 **/var/www/myserver** 를 생성하는 두 번째 파일 리소스 선언을 추가합니다.
- **notify => Service[" httpd "] 속성을 사용하여 구성 파일과 httpd** 서비스 간의 관계를 추가합니다. 이렇게 하면 구성 파일에서 변경 사항이 있는지 확인합니다. 파일이 변경되면 Puppet에서 서비스를 다시 시작합니다.

4.2. OPENSTACK PUPPET 모듈 가져오기

Red Hat OpenStack Platform은 공식 OpenStack Puppet 모듈을 사용합니다. OpenStack Puppet 모듈을 얻으려면 **Github** 의 **openstack** 그룹을 참조하십시오.

절차

1. 브라우저에서 https://github.com/openstack 으로 이동합니다.
2. filters 섹션에서 **Puppet** 을 검색합니다. 모든 Puppet 모듈은 접두사 **puppet-** 을 사용합니다.
3. 원하는 Puppet 모듈을 복제합니다. 예를 들어 공식 OpenStack Block Storage(cinder) 모듈:

```
$ git clone https://github.com/openstack/puppet-cinder.git
```

4.3. PUPPET 모듈의 구성 예

OpenStack 모듈은 주로 핵심 서비스를 구성하는 것을 목표로 합니다. 대부분의 모듈에는 추가 서비스를 구성하는 추가 매니페스트도 포함되어 있으며, 경우에 따라 **백엔드,에이전트** 또는 **플러그인** 이라고 합니다. 예를 들어 **cinder** 모듈에는 NFS, iSCSI, Red Hat Ceph Storage 등과 같은 다양한 스토리지 장치에 대한 구성 옵션이 포함된 **backends** 라는 디렉터리가 있습니다.

예를 들어 **manifests/backends/nfs.pp** 파일에는 다음 구성이 포함되어 있습니다.

```
define cinder::backend::nfs (
  $volume_backend_name  = $name,
  $nfs_servers          = [],
  $nfs_mount_options    = undef,
  $nfs_disk_util        = undef,
```

```
  $nfs_sparsed_volumes  = undef,
  $nfs_mount_point_base = undef,
  $nfs_shares_config    = '/etc/cinder/shares.conf',
  $nfs_used_ratio       = '0.95',
  $nfs_oversub_ratio    = '1.0',
  $extra_options        = {},
) {

  file {$nfs_shares_config:
    content => join($nfs_servers, "\n"),
    require => Package['cinder'],
    notify  => Service['cinder-volume']
  }

  cinder_config {
    "${name}/volume_backend_name":  value => $volume_backend_name;
    "${name}/volume_driver":        value =>
      'cinder.volume.drivers.nfs.NfsDriver';
    "${name}/nfs_shares_config":    value => $nfs_shares_config;
    "${name}/nfs_mount_options":    value => $nfs_mount_options;
    "${name}/nfs_disk_util":        value => $nfs_disk_util;
    "${name}/nfs_sparsed_volumes":  value => $nfs_sparsed_volumes;
    "${name}/nfs_mount_point_base": value => $nfs_mount_point_base;
    "${name}/nfs_used_ratio":       value => $nfs_used_ratio;
    "${name}/nfs_oversub_ratio":    value => $nfs_oversub_ratio;
  }

  create_resources('cinder_config', $extra_options)

}
```

결과는 다음과 같습니다.

- **define** 문은 **cinder::backend::nfs** 라는 정의된 유형을 만듭니다. 정의된 유형은 클래스와 유사합니다. 주요 차이점은 Puppet이 정의된 유형을 여러 번 평가합니다. 예를 들어, NFS 백엔드가 여러 개 있어야 하므로 각 NFS 공유에 대해 여러 평가가 필요할 수 있습니다.

- 다음 몇 줄에서는 이 구성의 매개 변수와 해당 기본값을 정의합니다. 사용자가 **cinder::backend::nfs** 정의 유형으로 새 값을 전달하면 기본값을 덮어씁니다.

- **file** 함수는 파일 생성을 호출하는 리소스 선언입니다. 이 파일에는 NFS 공유 목록이 있으며 이 파일의 이름은 **$nfs_shares_config = '/etc/cinder/shares.conf** 에 정의되어 있습니다. 추가 특성을 확인합니다.

  - **content** 속성은 **$nfs_servers** 매개 변수를 사용하여 목록을 생성합니다.

  - **require** 특성을 사용하면 **cinder** 패키지가 설치되어 있는지 확인합니다.

  - **notify** 속성은 **cinder-volume** 서비스를 재설정하도록 지시합니다.

- **cinder_config** 함수는 모듈의 **lib/puppet/** 디렉터리의 플러그인을 사용하는 리소스 선언입니다. 이 플러그인은 **/etc/cinder/cinder.conf** 파일에 구성을 추가합니다. 이 리소스의 각 행은 **cinder.conf** 파일의 관련 섹션에 구성 옵션을 추가합니다. 예를 들어 **$name** 매개 변수가 **mynfs** 인 경우 다음 특성을 사용합니다.

  ```
  "${name}/volume_backend_name":  value => $volume_backend_name;
  ```

```
  "${name}/volume_driver":        value =>
   'cinder.volume.drivers.nfs.NfsDriver';
  "${name}/nfs_shares_config":    value => $nfs_shares_config;
```

cinder.conf 파일에 다음 스니펫을 저장합니다.

```
[mynfs]
volume_backend_name=mynfs
volume_driver=cinder.volume.drivers.nfs.NfsDriver
nfs_shares_config=/etc/cinder/shares.conf
```

- **create_resources** 함수는 해시를 리소스 세트로 변환합니다. 이 경우 매니페스트는 **$extra_options** 해시를 백엔드의 추가 구성 옵션 집합으로 변환합니다. 이를 통해 매니페스트의 핵심 매개변수에 포함되지 않은 추가 구성 옵션을 추가할 수 있는 유연한 방법을 제공합니다.

이는 하드웨어의 OpenStack 드라이버를 구성하는 매니페스트 포함의 중요성을 보여줍니다. 매니페스트는 director가 하드웨어와 관련된 설정 옵션을 포함하는 방법을 제공합니다. 이는 director가 하드웨어를 사용하도록 오버클라우드를 설정하는 주요 통합 지점 역할을 합니다.

4.4. PUPPET 구성에 HIERA 데이터를 추가하는 예

Puppet에는 노드별 구성을 제공하는 키 값 시스템으로 작동하는 **hiera** 라는 툴이 포함되어 있습니다. 이러한 키와 값은 일반적으로 **/etc/puppet/hieradata** 에 있는 파일에 저장됩니다. **/etc/puppet/hiera.yaml** 파일은 Puppet이 **hieradata** 디렉터리에서 파일을 읽는 순서를 정의합니다.

오버클라우드 설정 중에 Puppet은 hiera 데이터를 사용하여 특정 Puppet 클래스의 기본값을 덮어씁니다. 예를 들어 **puppet-cinder** 의 **cinder::backend::nfs** 의 기본 NFS 마운트 옵션은 정의되지 않습니다.

```
  $nfs_mount_options    = undef,
```

그러나 **cinder::backend::nfs** 정의 유형을 호출하는 고유한 매니페스트를 만들고 이 옵션을 hiera 데이터로 교체할 수 있습니다.

```
cinder::backend::nfs { $cinder_nfs_backend:
  nfs_mount_options   => hiera('cinder_nfs_mount_options'),
}
```

즉 **nfs_mount_options** 매개변수는 **cinder_nfs_mount_options** 키의 hiera 데이터 값을 사용합니다.

```
cinder_nfs_mount_options: rsize=8192,wsize=8192
```

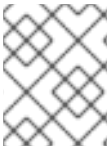
또는 NFS 구성의 모든 평가에 적용하도록 hiera 데이터를 사용하여 **cinder::backend::nfs::nfs::nfs_mount_options** 매개변수를 직접 덮어쓸 수 있습니다.

```
cinder::backend::nfs::nfs_mount_options: rsize=8192,wsize=8192
```

위의 hiera 데이터는 **cinder::backend::nfs** 의 각 평가에서 이 매개변수를 덮어씁니다.

5장. 오케스트레이션

RHOSP(Red Hat OpenStack Platform) director는 오버클라우드 배포 계획의 템플릿 형식으로 HOT(Heat Orchestration Templates)를 사용합니다. HOT 형식의 템플릿은 일반적으로 YAML 형식으로 표시됩니다. 템플릿의 목적은 heat가 생성하는 리소스 컬렉션 및 리소스 구성인 *스택*을 정의하고 생성하는 것입니다. 리소스는 RHOSP의 오브젝트이며 컴퓨팅 리소스, 네트워크 구성, 보안 그룹, 스케일링 규칙 및 사용자 정의 리소스를 포함할 수 있습니다.



참고

RHOSP에서 heat 템플릿 파일을 사용자 지정 템플릿 리소스로 사용하려면 파일 확장자는 **.yaml** 또는 **.template** 이어야 합니다.

이 장에서는 고유한 템플릿 파일을 생성할 수 있도록 HOT 구문을 이해하기 위한 몇 가지 기본 사항을 제공합니다.

5.1. HEAT 템플릿 기본 사항 학습

5.1.1. heat 템플릿 이해

Heat 템플릿에는 다음 세 가지 주요 섹션이 있습니다.

매개 변수

이러한 설정은 스택을 사용자 지정하기 위해 heat에 전달됩니다. heat 매개변수를 사용하여 기본값을 사용자 지정할 수도 있습니다. 이러한 설정은 템플릿의 **parameters** 섹션에 정의되어 있습니다.

리소스

스택의 일부로 생성 및 구성하는 특정 오브젝트입니다. RHOSP(Red Hat OpenStack Platform)에는 모든 구성 요소에 걸쳐 있는 코어 리소스 세트가 포함되어 있습니다. 이러한 리소스는 템플릿의 **resources** 섹션에 정의되어 있습니다.

출력 결과

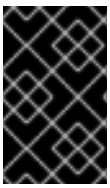
스택 생성 후 heat에서 전달되는 값입니다. heat API 또는 클라이언트 도구를 통해 이러한 값에 액세스할 수 있습니다. 이러한 값은 템플릿의 **output** 섹션에 정의되어 있습니다.

다음은 기본 heat 템플릿의 예입니다.

```
heat_template_version: 2013-05-23

description: > A very basic Heat template.

parameters:
  key_name:
    type: string
    default: lars
    description: Name of an existing key pair to use for the instance
  flavor:
    type: string
    description: Instance type for the instance to be created
    default: m1.small
  image:
    type: string
    default: cirros
    description: ID or name of the image to use for the instance
```

```
resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      name: My Cirros Instance
      image: { get_param: image }
      flavor: { get_param: flavor }
      key_name: { get_param: key_name }

output:
  instance_name:
    description: Get the instance's name
    value: { get_attr: [ my_instance, name ] }
```

이 템플릿은 리소스 유형 **유형인 OS::Nova::Server** 를 사용하여 특정 플레이버, 이미지 및 키로 **my_instance** 라는 인스턴스를 생성합니다. 스택은 **My Cirros Instance** 라고도 하는 **instance_name** 의 값을 반환할 수 있습니다.



중요

heat 템플릿에는 사용할 구문 버전과 사용 가능한 기능을 정의하는 **heat_template_version** 매개변수도 필요합니다. 자세한 내용은 공식 Heat 문서 를 참조하십시오.

5.1.2. 환경 파일 이해

환경 파일은 heat 템플릿에 대한 사용자 지정을 제공하는 특수 유형의 템플릿입니다. 여기에는 세 가지 핵심 부분이 포함됩니다.

리소스 레지스트리

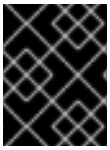
이 섹션에서는 다른 heat 템플릿에 연결된 사용자 정의 리소스 이름을 정의합니다. 이를 통해 코어 리소스 컬렉션에 없는 사용자 정의 리소스를 생성할 수 있습니다. 이러한 값은 환경 파일의 **resource_registry** 섹션에 정의되어 있습니다.

매개 변수

최상위 템플릿의 매개 변수에 적용하는 일반적인 설정입니다. 예를 들어 리소스 레지스트리 매핑과 같이 중첩 스택을 배포하는 템플릿이 있는 경우 매개변수는 중첩 리소스에 대한 템플릿이 아닌 최상위 템플릿에만 적용됩니다. 매개변수는 환경 파일의 **parameters** 섹션에 정의되어 있습니다.

매개변수 기본값

이러한 매개변수는 모든 템플릿에서 매개변수의 기본값을 수정합니다. 예를 들어 리소스 레지스트리 매핑과 같이 중첩 스택을 배포하는 heat 템플릿이 있는 경우 매개변수 기본값은 모든 템플릿에 적용됩니다. 매개변수 기본값은 환경 파일의 **parameter_defaults** 섹션에 정의되어 있습니다.



중요

오버클라우드의 사용자 지정 환경 파일을 생성할 때 **매개변수** 대신 **parameter_defaults** 를 사용합니다. 이는 매개변수가 오버클라우드의 모든 스택 템플릿에 적용되도록 합니다.

기본 환경 파일의 예:

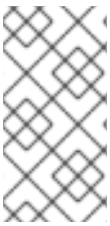
```
resource_registry:
  OS::Nova::Server::MyServer: myserver.yaml
```

```
parameter_defaults:
  NetworkName: my_network

parameters:
  MyIP: 192.168.0.1
```

heat 템플릿 **my_template.yaml** 에서 스택을 생성할 때 환경 파일 **my_env.yaml** 이 포함될 수 있습니다. **my_env.yaml** 파일은 **OS::Nova::Server::MyServer** 라는 새 리소스 유형을 생성합니다. **myserver.yaml** 파일은 기본 제공 항목을 재정의하는 이 리소스 유형의 구현을 제공하는 heat 템플릿 파일입니다. **my_template.yaml** 파일에 **OS::Nova::Server::MyServer** 리소스를 포함할 수 있습니다.

MyIP 는 이 환경 파일과 함께 배포하는 기본 heat 템플릿에만 매개변수를 적용합니다. 이 예에서는 **my_template.yaml** 의 매개변수에만 적용됩니다.

NetworkName 은 기본 heat 템플릿인 **my_template.yaml** 및 이 예제의 **OS::Nova::Server::MyServer** 리소스와 같은 기본 템플릿이 포함된 리소스와 연결된 템플릿 모두에 적용됩니다.



참고

RHOSP에서 heat 템플릿 파일을 사용자 지정 템플릿 리소스로 사용하려면 파일 확장자는 .yaml 또는 .template 이어야 합니다.

5.2. 기본 DIRECTOR 템플릿 가져오기

director는 고급 heat 템플릿 컬렉션을 사용하여 오버클라우드를 생성합니다. 이 컬렉션은 openstack-tripleo-heat-templates 리포지토리의 Github 의 openstack 그룹에서 사용할 수 있습니다.

절차

- **이 템플릿 컬렉션의 복제본을 가져오려면 다음 명령을 입력합니다.**

```
$ git clone https://github.com/openstack/tripleo-heat-templates.git
```

참고

이 템플릿 컬렉션의 Red Hat 특정 버전은 /usr/share/openstack-tripleo-heat-templates 에 컬렉션을 설치하는 openstack-tripleo-heat-template 패키지에서 사용할 수 있습니다.

이 템플릿 컬렉션의 주요 파일 및 디렉터리는 다음과 같습니다.

overcloud.j2.yaml

이는 오버클라우드 환경을 생성하는 기본 템플릿 파일입니다. 이 파일은 **Jinja2** 구문을 사용하여 템플릿의 특정 섹션을 반복하여 사용자 지정 역할을 생성합니다. **Jinja2** 포맷은 **Overcloud** 배포 프로세스 중에 **YAML**로 렌더링됩니다.

overcloud-resource-registry-puppet.j2.yaml

이는 오버클라우드 환경을 생성하는 기본 환경 파일입니다. 오버클라우드 이미지에 저장된 **Puppet** 모듈에 대한 구성 세트를 제공합니다. **director**가 각 노드에 오버클라우드 이미지를 작성한 후 **heat**는 이 환경 파일에 등록된 리소스를 사용하여 각 노드의 **Puppet** 구성을 시작합니다. 이 파일은 **Jinja2** 구문을 사용하여 템플릿의 특정 섹션을 반복하여 사용자 지정 역할을 생성합니다. **Jinja2** 포맷은 **Overcloud** 배포 프로세스 중에 **YAML**로 렌더링됩니다.

roles_data.yaml

이 파일은 오버클라우드에서 역할을 정의하고 서비스를 각 역할에 매핑하는 파일입니다.

network_data.yaml

이 파일은 오버클라우드의 네트워크와 서브넷, 할당 풀, **VIP** 상태와 같은 해당 속성을 정의하는 파일입니다. 기본 **network_data** 파일에는 기본 네트워크 **External, Internal Api, Storage, Storage Management, Tenant** 및 **Management**가 포함됩니다. 사용자 지정 **network_data** 파일을 생성하고 **-n** 옵션을 사용하여 **openstack overcloud deploy** 명령에 추가할 수 있습니다.

plan-environment.yaml

이 파일은 오버클라우드 계획에 대한 메타데이터를 정의하는 파일입니다. 여기에는 계획 이름, 사용할 기본 템플릿, 오버클라우드에 적용할 환경 파일이 포함됩니다.

capabilities-map.yaml

이는 오버클라우드 계획의 환경 파일 매핑입니다. 이 파일을 사용하여 **Director web UI**에서 환경 파일을 설명하고 활성화합니다. 오버클라우드 계획의 환경 디렉터리에서 탐지되었지만 **capabilities-map.yaml** 에 정의되어 있지 않은 사용자 지정 환경 파일은 **2**의 기타 하위 탭에 나열됩니다. **2**의 기타 하위에는 배포 구성 **>** 전체 설정을 지정합니다.

환경

오버클라우드 생성과 함께 사용할 수 있는 추가 **heat** 환경 파일이 포함되어 있습니다. 이러한 환경 파일을 사용하면 결과 **RHOSP(Red Hat OpenStack Platform)** 환경에 추가 기능을 사용할 수 있습니다. 예를 들어 디렉터리에는 **Cinder NetApp** 백엔드 스토리지(**cinder-netapp-config.yaml**)를 활성화하는 환경 파일이 포함되어 있습니다. **capabilities-map.yaml** 파일에 정의되지 않은 이 디렉터리에서 탐지되는 환경 파일은 **2**의 기타 하위 탭에 나열됩니다. **2**의 기타 하위 탭에는 **Director**의 웹 **UI**에서 배포 구성 **> Overall Settings**를 지정합니다.

network

이는 격리된 네트워크 및 포트를 생성하는 데 도움이 되는 일련의 **heat** 템플릿입니다.

Puppet

이러한 템플릿은 주로 **Puppet**으로 구성에 의해 구동되는 템플릿입니다. **overcloud-resource-registry-puppet.j2.yaml** 환경 파일은 이 디렉터리의 파일을 사용하여 각 노드에서 **Puppet** 설정 애플리케이션을 구동합니다.

Puppet/services

구성 가능 서비스 아키텍처의 모든 서비스에 대한 **heat** 템플릿이 포함된 디렉터리입니다.

extraconfig

이러한 템플릿은 추가 기능을 활성화하는 템플릿입니다.

firstboot

director가 노드를 처음 생성할 때 사용하는 **first_boot** 스크립트의 예를 제공합니다.

그러면 **director**가 **Overcloud** 생성을 오케스트레이션하는 데 사용하는 템플릿에 대한 일반적인 개요가 제공됩니다. 다음 몇 섹션에서는 **Overcloud** 배포에 추가할 수 있는 고유한 사용자 지정 템플릿과 환경 파일을 생성하는 방법을 보여줍니다.

5.3. 첫 번째 부팅: 첫 번째 부팅 구성 사용자 지정

director는 **Overcloud** 초기 생성 시 모든 노드에서 설정을 수행하는 메커니즘을 제공합니다. **director**는 **OS::TripleO::NodeUserData** 리소스 유형을 사용하여 호출할 수 있는 **cloud-init** 를 통해 이 작업을 수행합니다.

이 예에서는 모든 노드에서 사용자 지정 **IP** 주소로 네임서버를 업데이트합니다. 먼저 각 노드의 **resolv.conf** 를 특정 이름 서버로 추가하는 스크립트를 실행하는 기본 **heat** 템플릿 (**/home/stack/templates/nameserver.yaml**)을 생성해야 합니다. **OS::TripleO::MultipartMime** 리소스 유형을 사용하여 구성 스크립트를 보낼 수 있습니다.

```
heat_template_version: 2014-10-16

description: >
  Extra hostname configuration

resources:
  userdata:
    type: OS::Heat::MultipartMime
    properties:
      parts:
      - config: {get_resource: nameserver_config}

  nameserver_config:
```

```
    type: OS::Heat::SoftwareConfig
    properties:
      config: |
        #!/bin/bash
        echo "nameserver 192.168.1.1" >> /etc/resolv.conf

outputs:
  OS::stack_id:
    value: {get_resource: userdata}
```

다음으로 **heat** 템플릿을 **OS::TripleO::NodeUserData** 리소스 유형으로 등록하는 환경 파일 **(/home/stack/templates/firstboot.yaml)**을 생성합니다.

```
resource_registry:
  OS::TripleO::NodeUserData: /home/stack/templates/nameserver.yaml
```

첫 번째 부팅 구성을 추가하려면 **Overcloud**를 처음 생성할 때 다른 환경 파일과 함께 환경 파일을 스택에 추가합니다. 예를 들면 다음과 같습니다.

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/firstboot.yaml \
    ...
```

-e 는 환경 파일을 **Overcloud** 스택에 적용합니다.

이렇게 하면 처음 생성되고 부팅될 때 모든 노드에 구성이 추가됩니다. **Overcloud** 스택 업데이트와 같은 이러한 템플릿을 나중에 포함해도 해당 스크립트가 실행되지 않습니다.

중요

OS::TripleO::NodeUserData 를 하나의 **heat** 템플릿에만 등록할 수 있습니다. 후속 사용은 사용할 **heat** 템플릿을 재정의합니다.

이렇게 하면 다음이 수행됩니다.

1. **OS::TripleO::NodeUserData** 는 컬렉션의 다른 템플릿에 사용되는 **director** 기반 **Heat** 리소스입니다. 이 리소스는 **cloud-init** 에서 사용할 데이터를 전달합니다. 기본 **NodeUserData** 는 빈 값**(firstboot/userdata_default.yaml)**을 생성하는 **Heat** 템플릿을 나타냅니다. 이 경우

firstboot.yaml 환경 파일은 이 기본값을 자체 **nameserver.yaml** 파일에 대한 참조로 대체합니다.

2. **nameserver_config** 는 첫 번째 부팅 시 실행할 **Bash** 스크립트를 정의합니다. **OS::Heat::SoftwareConfig** 리소스는 적용할 구성 조각으로 정의합니다.

3. **userdata** 는 **OS::Heat::MultipartMime** 리소스를 사용하여 **nameserver_config** 의 구성을 다중 파트 **MIME** 메시지로 변환합니다.

4. 출력 은 출력 매개 변수 **OS::stack_id** 를 제공하여 **userdata** 의 **MIME** 메시지를 가져와 호출하는 **Heat** 템플릿/리소스에 제공합니다.

결과적으로 각 노드는 첫 번째 부팅 시 다음 **Bash** 스크립트를 실행합니다.

```
#!/bin/bash
echo "nameserver 192.168.1.1" >> /etc/resolve.conf
```

이 예에서는 **Heat** 템플릿이 한 리소스에서 다른 리소스로 전달 및 **modfy** 구성을 전달하는 방법을 보여줍니다. 또한 환경 파일을 사용하여 새 **Heat** 리소스를 등록하거나 기존 리소스를 수정하는 방법도 보여줍니다.

5.4. 사전 구성: 특정 OVERCLOUD 역할 사용자 지정

중요

이전 버전의 이 문서에서는 **OS::TripleO::Tasks::*PreConfig** 리소스를 사용하여 역할별로 사전 구성 후크를 제공했습니다. director의 **Heat** 템플릿 컬렉션에는 이러한 후크를 전용으로 사용해야 하므로 사용자 지정 용도로 사용해서는 안 됩니다. 대신 아래에 설명된 **OS::TripleO::*ExtraConfigPre** 후크를 사용하십시오.

Overcloud는 **OpenStack** 구성 요소의 핵심 구성에 **Puppet**을 사용합니다. **director**는 첫 번째 부팅이 완료되고 코어 구성이 시작되기 전에 특정 노드 역할에 대한 사용자 정의 구성을 제공하는 후크 세트를 제공합니다. 다음 후크는 다음과 같습니다.

OS::TripleO::ControllerExtraConfigPre

코어 **Puppet** 구성보다 먼저 컨트롤러 노드에 적용되는 추가 구성입니다.

OS::TripleO::ComputeExtraConfigPre

코어 **Puppet** 구성보다 먼저 컴퓨팅 노드에 적용된 추가 구성입니다.

OS::TripleO::CephStorageExtraConfigPre

코어 **Puppet** 구성보다 먼저 **Ceph Storage** 노드에 적용된 추가 구성입니다.

OS::TripleO::ObjectStorageExtraConfigPre

핵심 **Puppet** 구성보다 먼저 **Object Storage** 노드에 적용된 추가 구성입니다.

OS::TripleO::BlockStorageExtraConfigPre

코어 **Puppet** 구성보다 먼저 블록 스토리지 노드에 적용된 추가 구성입니다.

OS::TripleO::[ROLE]ExtraConfigPre

코어 **Puppet** 구성 전에 사용자 정의 노드에 적용된 추가 설정입니다. **[ROLE]** 을 구성 가능 역할 이름으로 교체합니다.

이 예에서는 먼저 변수 이름 서버가 있는 노드의 **resolv.conf** 에 쓰는 스크립트를 실행하는 기본 **heat** 템플릿(**/home/stack/templates/nameserver.yaml**)을 생성합니다.

```
heat_template_version: 2014-10-16

description: >
  Extra hostname configuration

parameters:
  server:
    type: string
  nameserver_ip:
    type: string
  DeployIdentifier:
    type: string

resources:
  CustomExtraConfigPre:
    type: OS::Heat::SoftwareConfig
    properties:
      group: script
      config:
        str_replace:
          template: |
            #!/bin/sh
            echo "nameserver _NAMESERVER_IP_" > /etc/resolv.conf
          params:
            _NAMESERVER_IP_: {get_param: nameserver_ip}

  CustomExtraDeploymentPre:
```

```
    type: OS::Heat::SoftwareDeployment
    properties:
      server: {get_param: server}
      config: {get_resource: CustomExtraConfigPre}
      actions: ['CREATE','UPDATE']
      input_values:
        deploy_identifier: {get_param: DeployIdentifier}

outputs:
  deploy_stdout:
    description: Deployment reference, used to trigger pre-deploy on changes
    value: {get_attr: [CustomExtraDeploymentPre, deploy_stdout]}
```

이 예에서 **resources** 섹션에는 다음이 포함됩니다.

CustomExtraConfigPre

이는 소프트웨어 구성을 정의합니다. 이 예제에서는 **Bash** 스크립트 를 정의하고 **Heat**가 **_NAMESERVER_IP_** 를 **nameserver_ip** 매개변수에 저장된 값으로 대체합니다.

CustomExtraDeploymentPre

이 명령은 소프트웨어 구성을 실행합니다. 이 구성은 **CustomExtraConfigPre** 리소스의 소프트웨어 구성입니다. 다음을 확인합니다.

- **config** 매개변수는 **CustomExtraConfigPre** 리소스를 참조하므로 **Heat**에서 적용할 구성을 알 수 있습니다.
- **server** 매개변수는 **Overcloud** 노드의 맵을 검색합니다. 이 매개변수는 상위 템플릿에서 제공하며 이 후크의 템플릿에서 필수입니다.
- **actions** 매개변수는 구성을 적용할 시기를 정의합니다. 이 경우 **Overcloud**가 생성되거나 업데이트된 경우에만 구성을 적용합니다. 가능한 작업에는 **CREATE,UPDATE,DELETE,SUSPEND** 및 **RESUME** 가 포함됩니다.
- **input_values** 에는 상위 템플릿에서 **DeployIdentifier** 를 저장하는 **deploy_ octets** 라는 매개 변수가 포함되어 있습니다. 이 매개변수는 배포 각 업데이트의 리소스에 타임스탬프를 제공합니다. 이렇게 하면 리소스가 후속 오버클라우드 업데이트에 다시 적용됩니다.

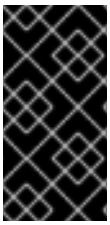
다음으로 **heat** 템플릿을 역할 기반 리소스 유형에 등록하는 환경 파일 (**/home/stack/templates/pre_config.yaml**)을 생성합니다. 예를 들어 컨트롤러 노드에만 적용하려면 **ControllerExtraConfigPre** 후크를 사용합니다.

```
resource_registry:
  OS::TripleO::ControllerExtraConfigPre: /home/stack/templates/nameserver.yaml

parameter_defaults:
  nameserver_ip: 192.168.1.1
```

구성을 적용하려면 오버클라우드를 만들거나 업데이트할 때 다른 환경 파일과 함께 환경 파일을 스택에 추가합니다. 예를 들면 다음과 같습니다.

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/pre_config.yaml \
    ...
```

이는 코어 구성이 초기 오버클라우드 생성 또는 후속 업데이트에서 시작되기 전에 모든 컨트롤러 노드에 구성을 적용합니다.



중요

후크당 하나의 **Heat** 템플릿에만 각 리소스를 등록할 수 있습니다. 후속 사용은 사용할 **Heat** 템플릿을 재정의합니다.

이렇게 하면 다음이 수행됩니다.

1. **OS::TripleO::ControllerExtraConfigPre** 은 **Heat** 템플릿 컬렉션의 구성 템플릿에 사용되는 **director** 기반 **Heat** 리소스입니다. 이 리소스는 각 컨트롤러 노드에 구성을 전달합니다. 기본 **ControllerExtraConfigPre** 은 빈 값(**puppet/extraconfig/pre_deploy/default.yaml**)을 생성하는 **Heat** 템플릿을 나타냅니다. 이 경우 **pre_config.yaml** 환경 파일은 이 기본값을 자체 **nameserver.yaml** 파일에 대한 참조로 대체합니다.

2. 환경 파일은 또한 **nameserver_ip** 를 환경의 **parameter_default** 값으로 전달합니다. 이 매개변수는 이름 서버의 **IP** 주소를 저장합니다. 그런 다음 **nameserver.yaml Heat** 템플릿은 **parameters** 섹션에 정의된 대로 이 매개변수를 허용합니다.

3. 템플릿은 **OS::Heat::SoftwareConfig** 를 통해 **CustomExtraConfigPre** 을 구성 리소스로 정의합니다. 그룹**: script** 속성을 확인합니다. 그룹 은 **Heat**의 일련의 후크를 통해 사용할 수 있는 소프트웨어 구성 도구를 정의합니다. 이 경우 스크립트 후크는 **SoftwareConfig** 리소스에 **config** 속성으로 정의한 실행 가능한 스크립트를 실행합니다.

4. 스크립트 자체는 이름 서버 **IP** 주소를 사용하여 **/etc/resolve.conf** 를 추가합니다.

str_replace 속성을 참조하여 **template** 섹션의 변수를 **params** 섹션의 매개 변수로 교체할 수 있습니다. 이 경우 ***NAMESERVER_IP*** 를 이름 서버 **IP** 주소로 설정하여 스크립트에서 동일한 변수를 대체합니다. 그러면 다음 스크립트가 생성됩니다.

```
#!/bin/sh
echo "nameserver 192.168.1.1" >> /etc/resolve.conf
```

이 예에서는 코어 구성 전에 구성을 정의하고 **OS::Heat::SoftwareConfig** 및 **OS::Heat::SoftwareDeployments** 를 사용하여 배포하는 **Heat** 템플릿을 생성하는 방법을 보여줍니다. 또한 환경 파일에서 매개 변수를 정의하고 구성의 템플릿에 전달하는 방법을 보여줍니다.

5.5. 사전 구성: 모든 OVERCLOUD 역할 사용자 지정

Overcloud는 **OpenStack** 구성 요소의 핵심 구성에 **Puppet**을 사용합니다. **director**는 첫 번째 부팅이 완료된 후 및 코어 구성이 시작되기 전에 모든 노드 유형을 구성하는 후크를 제공합니다.

OS::TripleO::NodeExtraConfig

핵심 **Puppet** 구성보다 먼저 모든 노드 역할에 적용되는 추가 구성입니다.

이 예제에서는 먼저 각 노드의 **resolv.conf** 를 변수 이름 서버에 추가하는 스크립트를 실행하는 기본 **heat** 템플릿(**/home/stack/templates/nameserver.yaml**)을 생성합니다.

```
heat_template_version: 2014-10-16

description: >
  Extra hostname configuration

parameters:
  server:
    type: string
  nameserver_ip:
    type: string
  DeployIdentifier:
    type: string

resources:
  CustomExtraConfigPre:
    type: OS::Heat::SoftwareConfig
    properties:
      group: script
      config:
        str_replace:
          template: |
            #!/bin/sh
            echo "nameserver _NAMESERVER_IP_" >> /etc/resolv.conf
```

```
        params:
          _NAMESERVER_IP_: {get_param: nameserver_ip}

  CustomExtraDeploymentPre:
    type: OS::Heat::SoftwareDeployment
    properties:
      server: {get_param: server}
      config: {get_resource: CustomExtraConfigPre}
      actions: ['CREATE','UPDATE']
      input_values:
        deploy_identifier: {get_param: DeployIdentifier}

outputs:
  deploy_stdout:
    description: Deployment reference, used to trigger pre-deploy on changes
    value: {get_attr: [CustomExtraDeploymentPre, deploy_stdout]}
```

이 예에서 **resources** 섹션에는 다음이 포함됩니다.

CustomExtraConfigPre

이는 소프트웨어 구성을 정의합니다. 이 예제에서는 **Bash** 스크립트 를 정의하고 **Heat**가 **_NAMESERVER_IP_** 를 **nameserver_ip** 매개변수에 저장된 값으로 대체합니다.

CustomExtraDeploymentPre

이 명령은 소프트웨어 구성을 실행합니다. 이 구성은 **CustomExtraConfigPre** 리소스의 소프트웨어 구성입니다. 다음을 확인합니다.

- **config** 매개변수는 **CustomExtraConfigPre** 리소스를 참조하므로 **Heat**에서 적용할 구성을 알 수 있습니다.
- **server** 매개변수는 **Overcloud** 노드의 맵을 검색합니다. 이 매개변수는 상위 템플릿에서 제공하며 이 후크의 템플릿에서 필수입니다.
- **actions** 매개변수는 구성을 적용할 시기를 정의합니다. 이 경우 **Overcloud**가 생성되거나 업데이트된 경우에만 구성을 적용합니다. 가능한 작업에는 **CREATE,UPDATE,DELETE,SUSPEND** 및 **RESUME** 가 포함됩니다.
- **input_values** 매개 변수에는 상위 템플릿에서 **DeployIdentifier** 를 저장하는 **deploy_tekton** 라는 하위 매개변수가 포함되어 있습니다. 이 매개변수는 배포 각 업데이트의 리소스에 타임스탬프를 제공합니다. 이렇게 하면 리소스가 후속 오버클라우드 업데이트에 다시 적용됩니다.

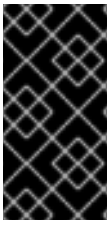
다음으로 **heat** 템플릿을 **OS::TripleO::NodeExtraConfig** 리소스 유형으로 등록하는 환경 파일 (**/home/stack/templates/pre_config.yaml**)을 생성합니다.

```
resource_registry:
  OS::TripleO::NodeExtraConfig: /home/stack/templates/nameserver.yaml

parameter_defaults:
  nameserver_ip: 192.168.1.1
```

구성을 적용하려면 오버클라우드를 만들거나 업데이트할 때 다른 환경 파일과 함께 환경 파일을 스택에 추가합니다. 예를 들면 다음과 같습니다.

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/pre_config.yaml \
    ...
```

이는 코어 구성이 초기 오버클라우드 생성 또는 후속 업데이트에서 시작되기 전에 모든 노드에 구성을 적용합니다.



중요

OS::TripleO::NodeExtraConfig 를 하나의 **Heat** 템플릿에만 등록할 수 있습니다. 후속 사용은 사용할 **Heat** 템플릿을 재정의합니다.

이렇게 하면 다음이 수행됩니다.

1. **OS::TripleO::NodeExtraConfig** 는 **Heat** 템플릿 컬렉션의 구성 템플릿에 사용되는 **director** 기반 **Heat** 리소스입니다. 이 리소스는 각 노드에 구성을 전달합니다. 기본 **NodeExtraConfig** 는 빈 값(**puppet/extraconfig/pre_deploy/default.yaml**)을 생성하는 **Heat** 템플릿을 나타냅니다. 이 경우 **pre_config.yaml** 환경 파일은 이 기본값을 자체 **nameserver.yaml** 파일에 대한 참조로 대체합니다.

2. 환경 파일은 또한 **nameserver_ip** 를 환경의 **parameter_default** 값으로 전달합니다. 이 매개변수는 이름 서버의 **IP** 주소를 저장합니다. 그런 다음 **nameserver.yaml Heat** 템플릿은 **parameters** 섹션에 정의된 대로 이 매개변수를 허용합니다.

3. 템플릿은 **OS::Heat::SoftwareConfig** 를 통해 **CustomExtraConfigPre** 을 구성 리소스로 정의합니다. 그룹**: script** 속성을 확인합니다. 그룹 은 **Heat**의 일련의 후크를 통해 사용할 수 있는 소

프트웨어 구성 도구를 정의합니다. 이 경우 스크립트 후크는 **SoftwareConfig** 리소스에 **config** 속성으로 정의한 실행 가능한 스크립트를 실행합니다.

4. 스크립트 자체는 이름 서버 **IP** 주소를 사용하여 **/etc/resolve.conf** 를 추가합니다. **str_replace** 속성을 참조하여 **template** 섹션의 변수를 **params** 섹션의 매개 변수로 교체할 수 있습니다. 이 경우 ***NAMESERVER_IP*** 를 이름 서버 **IP** 주소로 설정하여 스크립트에서 동일한 변수를 대체합니다. 그러면 다음 스크립트가 생성됩니다.

```
#!/bin/sh
echo "nameserver 192.168.1.1" >> /etc/resolve.conf
```

이 예에서는 코어 구성 전에 구성을 정의하고 **OS::Heat::SoftwareConfig** 및 **OS::Heat::SoftwareDeployments** 를 사용하여 배포하는 **Heat** 템플릿을 생성하는 방법을 보여줍니다. 또한 환경 파일에서 매개 변수를 정의하고 구성의 템플릿에 전달하는 방법을 보여줍니다.

5.6. POST-CONFIGURATION: 모든 OVERCLOUD 역할 사용자 정의

중요

이전 버전의 이 문서에서는 **OS::TripleO::Tasks::*PostConfig** 리소스를 사용하여 역할별로 사후 구성 후크를 제공했습니다. **director**의 **Heat** 템플릿 컬렉션에는 이러한 후크를 전용으로 사용해야 하므로 사용자 지정 용도로 사용해서는 안 됩니다. 대신 아래에 설명된 **OS::TripleO::NodeExtraConfigPost** 후크를 사용하십시오.

Overcloud 생성을 완료했지만 초기 생성 또는 **Overcloud** 후속 업데이트 시 모든 역할에 추가 구성을 추가하려는 상황이 발생할 수 있습니다. 이 경우 다음 사후 구성 후크를 사용합니다.

OS::TripleO::NodeExtraConfigPost

코어 **Puppet** 구성 후 모든 노드 역할에 적용되는 추가 구성입니다.

이 예제에서는 먼저 각 노드의 **resolv.conf** 를 변수 이름 서버에 추가하는 스크립트를 실행하는 기본 **heat** 템플릿(**/home/stack/templates/nameserver.yaml**)을 생성합니다.

```
description: >
  Extra hostname configuration

parameters:
  servers:
    type: json
  nameserver_ip:
```

```
    type: string
  DeployIdentifier:
    type: string

resources:
  CustomExtraConfig:
    type: OS::Heat::SoftwareConfig
    properties:
      group: script
      config:
        str_replace:
          template: |
            #!/bin/sh
            echo "nameserver _NAMESERVER_IP_" >> /etc/resolv.conf
          params:
            _NAMESERVER_IP_: {get_param: nameserver_ip}

  CustomExtraDeployments:
    type: OS::Heat::SoftwareDeploymentGroup
    properties:
      servers:  {get_param: servers}
      config: {get_resource: CustomExtraConfig}
      actions: ['CREATE','UPDATE']
      input_values:
        deploy_identifier: {get_param: DeployIdentifier}
```

이 예에서 **resources** 섹션에는 다음이 포함됩니다.

CustomExtraConfig

이는 소프트웨어 구성을 정의합니다. 이 예제에서는 **Bash** 스크립트 를 정의하고 **Heat**가 **_NAMESERVER_IP_** 를 **nameserver_ip** 매개변수에 저장된 값으로 대체합니다.

CustomExtraDeployments

이 명령은 소프트웨어 구성을 실행합니다. 이 구성은 **CustomExtraConfig** 리소스의 소프트웨어 구성입니다. 다음을 확인합니다.

- **config** 매개변수는 **CustomExtraConfig** 리소스를 참조하므로 **Heat**에서 적용할 구성을 알 수 있습니다.
- **servers** 매개변수는 **Overcloud** 노드의 맵을 검색합니다. 이 매개변수는 상위 템플릿에서 제공하며 이 후크의 템플릿에서 필수입니다.
- **actions** 매개변수는 구성을 적용할 시기를 정의합니다. 이 경우 **Overcloud**가 생성된 경우에만 구성을 적용합니다. 가능한 작업에는 **CREATE,UPDATE,DELETE,SUSPEND** 및

RESUME 가 포함됩니다.

- **input_values** 에는 상위 템플릿에서 **DeployIdentifier** 를 저장하는 **deploy_ octets** 라는 매개 변수가 포함되어 있습니다. 이 매개변수는 배포 각 업데이트의 리소스에 타임스탬프를 제공합니다. 이렇게 하면 리소스가 후속 오버클라우드 업데이트에 다시 적용됩니다.

다음으로 **heat** 템플릿을 **OS::TripleO::NodeExtraConfigPost:** 리소스 유형으로 등록하는 환경 파일 (**/home/stack/templates/post_config.yaml**)을 생성합니다.

```
resource_registry:
  OS::TripleO::NodeExtraConfigPost: /home/stack/templates/nameserver.yaml

parameter_defaults:
  nameserver_ip: 192.168.1.1
```

구성을 적용하려면 오버클라우드를 만들거나 업데이트할 때 다른 환경 파일과 함께 환경 파일을 스택에 추가합니다. 예를 들면 다음과 같습니다.

```
$ openstack overcloud deploy --templates \
    ...
    -e /home/stack/templates/post_config.yaml \
    ...
```

이는 초기 오버클라우드 생성 또는 후속 업데이트 중 하나에서 코어 구성이 완료된 후 모든 노드에 구성을 적용합니다.

중요

OS::TripleO::NodeExtraConfigPost 를 하나의 **Heat** 템플릿에만 등록할 수 있습니다. 후속 사용은 사용할 **Heat** 템플릿을 재정의합니다.

이렇게 하면 다음이 수행됩니다.

1. **OS::TripleO::NodeExtraConfigPost** 는 컬렉션의 구성 후 템플릿에서 사용되는 **director** 기반 **Heat** 리소스입니다. 이 리소스는 ***-post.yaml** 템플릿을 통해 각 노드 유형으로 구성을 전달합니다. 기본 **NodeExtraConfigPost** 는 빈 값(**extraconfig/post_deploy/default.yaml**)을 생성하는 **Heat** 템플릿을 나타냅니다. 이 경우 **post_config.yaml** 환경 파일은 이 기본값을 자체 **nameserver.yaml** 파일에 대한 참조로 대체합니다.

2. 환경 파일은 또한 **nameserver_ip** 를 환경의 **parameter_default** 값으로 전달합니다. 이 매개변수는 이름 서버의 **IP** 주소를 저장합니다. 그런 다음 **nameserver.yaml Heat** 템플릿은 **parameters** 섹션에 정의된 대로 이 매개변수를 허용합니다.

3. 템플릿은 **OS::Heat::SoftwareConfig** 를 통해 **CustomExtraConfig** 를 구성 리소스로 정의합니다. 그룹**: script** 속성을 확인합니다. 그룹 은 **Heat**의 일련의 후크를 통해 사용할 수 있는 소프트웨어 구성 도구를 정의합니다. 이 경우 스크립트 후크는 **SoftwareConfig** 리소스에 정의된 실행 가능한 스크립트를 **config** 속성으로 실행합니다.

4. 스크립트 자체는 이름 서버 **IP** 주소를 사용하여 **/etc/resolve.conf** 를 추가합니다. **str_replace** 속성을 참조하여 **template** 섹션의 변수를 **params** 섹션의 매개 변수로 교체할 수 있습니다. 이 경우 ***NAMESERVER_IP*** 를 이름 서버 **IP** 주소로 설정하여 스크립트에서 동일한 변수를 대체합니다. 그러면 다음 스크립트가 생성됩니다.

```
#!/bin/sh
echo "nameserver 192.168.1.1" >> /etc/resolve.conf
```

이 예에서는 구성을 정의하고 **OS::Heat::SoftwareConfig** 및 **OS::Heat::SoftwareDeployments** 를 사용하여 배포하는 **Heat** 템플릿을 생성하는 방법을 보여줍니다. 또한 환경 파일에서 매개 변수를 정의하고 구성의 템플릿에 전달하는 방법을 보여줍니다.

5.7. PUPPET: 오버클라우드에 사용자 지정 설정 적용

이전에는 **OpenStack Puppet** 모듈에 새 백엔드의 구성 추가에 대해 논의했습니다. 이 섹션에서는 **director**가 새 구성의 애플리케이션을 실행하는 방법을 보여줍니다.

Heat 템플릿은 후크를 제공하여 **OS::Heat::SoftwareConfig** 리소스와 함께 **Puppet** 구성을 적용할 수 있습니다. 프로세스는 **Bash** 스크립트를 포함하고 실행하는 방법과 유사합니다. 그러나 그룹인 **script hook** 대신 **puppet** 후크 그룹 을 사용합니다.

예를 들어 공식 **Cinder Puppet** 모듈을 사용하여 **NFS Cinder** 백엔드를 활성화하는 **Puppet** 매니페스트**(example-puppet-manifest.pp)**가 있을 수 있습니다.

```
cinder::backend::nfs { 'mynfsserver':
  nfs_servers          => ['192.168.1.200:/storage'],
}
```

이 **Puppet** 구성은 **cinder::backend::nfs** 정의 유형을 사용하여 새 리소스를 생성합니다. **Heat**를 통해 이 리소스를 적용하려면 **Puppet** 매니페스트를 실행하는 기본 **Heat** 템플릿**(puppet-config.yaml)**을 생성

합니다.

```
heat_template_version: 2014-10-16

parameters:
  servers:
    type: json

resources:
  ExtraPuppetConfig:
    type: OS::Heat::SoftwareConfig
    properties:
      group: puppet
      config:
        get_file: example-puppet-manifest.pp
      options:
        enable_hiera: True
        enable_facter: False

  ExtraPuppetDeployment:
    type: OS::Heat::SoftwareDeployments
    properties:
      config: {get_resource: ExtraPuppetConfig}
      servers: {get_param: servers}
      actions: ['CREATE','UPDATE']
```

다음으로 **Heat** 템플릿을 **OS::TripleO::NodeExtraConfigPost** 리소스 유형으로 등록하는 환경 파일 **(puppet_config.yaml)**을 생성합니다.

```
resource_registry:
  OS::TripleO::NodeExtraConfigPost: puppet_config.yaml
```

이 예제는 이전 섹션의 스크립트 후크 예제에서 **SoftwareConfig** 및 **SoftwareDeployments** 를 사용하는 것과 유사합니다. 그러나 이 예제에는 몇 가지 차이점이 있습니다.

1. **group: Puppet** 후크를 실행하도록 **puppet** 을 설정합니다.

2. **config** 속성은 **get_file** 특성을 사용하여 추가 구성이 포함된 **Puppet** 매니페스트를 나타냅니다.

3. **options** 속성에는 **Puppet** 구성과 관련된 몇 가지 옵션이 포함되어 있습니다.

   - **enable_hiera** 옵션을 사용하면 **Puppet** 설정에서 **Hiera** 데이터를 사용할 수 있습니다.

- **enable_facter** 옵션을 사용하면 **Puppet** 구성에서 **facter** 명령의 시스템 팩트를 사용할 수 있습니다.

이 예에서는 **Overcloud**의 소프트웨어 구성의 일부로 **Puppet** 매니페스트를 포함하는 방법을 보여줍니다. 이렇게 하면 오버클라우드 이미지에 기존 **Puppet** 모듈의 특정 설정 클래스를 적용할 수 있으므로 특정 소프트웨어 및 하드웨어를 사용하도록 **Overcloud**를 사용자 지정할 수 있습니다.

5.8. PUPPET: 역할에 대한 HIERADATA 사용자 정의

Heat 템플릿 컬렉션에는 특정 노드 유형에 추가 구성을 전달하는 매개변수 세트가 포함되어 있습니다. 이러한 매개변수는 구성을 노드의 **Puppet** 구성에 대한 **hieradata**로 저장합니다. 이러한 매개변수는 다음과 같습니다.

ControllerExtraConfig

모든 컨트롤러 노드에 추가할 구성입니다.

ComputeExtraConfig

모든 컴퓨팅 노드에 추가할 구성입니다.

BlockStorageExtraConfig

모든 블록 스토리지 노드에 추가할 구성입니다.

ObjectStorageExtraConfig

모든 **Object Storage** 노드에 추가할 구성

CephStorageExtraConfig

모든 **Ceph Storage** 노드에 추가하기 위한 구성

[ROLE]ExtraConfig

구성 가능 역할에 추가할 구성입니다. **[ROLE]** 을 구성 가능 역할 이름으로 교체합니다.

ExtraConfig

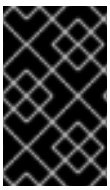
모든 노드에 추가할 구성입니다.

배포 후 구성 프로세스에 구성을 추가하려면 **parameter_defaults** 섹션에 이러한 매개변수가 포함된 환경 파일을 생성합니다. 예를 들어 **Compute** 호스트의 예약된 메모리를 **1024MB**로 늘리고 **VNC** 키맵을

Japanese로 설정하려면 다음을 수행합니다.

```
parameter_defaults:
  ComputeExtraConfig:
    nova::compute::reserved_host_memory: 1024
    nova::compute::vnc_keymap: ja
```

openstack overcloud deploy 를 실행할 때 이 환경 파일을 포함합니다.



중요

각 매개변수는 한 번만 정의할 수 있습니다. 이후의 사용은 이전 값을 재정의합니다.

5.9. 오버클라우드 배포에 환경 파일 추가

사용자 지정 구성과 관련된 환경 파일 세트를 개발한 후 **Overcloud** 배포에 이러한 파일을 포함합니다. 즉, **-e** 옵션과 함께 **openstack overcloud deploy** 명령을 실행한 다음 환경 파일을 실행합니다. 사용자 지정에 필요한 횟수만큼 **-e** 옵션을 지정할 수 있습니다. 예를 들면 다음과 같습니다.

```
$ openstack overcloud deploy --templates -e network-configuration.yaml -e storage-
configuration.yaml -e first-boot.yaml
```

중요

환경 파일은 연속으로 누적됩니다. 즉, 기본 **Heat** 템플릿 컬렉션과 이전 환경 파일 모두의 각 후속 파일 스택입니다. 이를 통해 리소스 정의를 재정의할 수 있습니다. 예를 들어 **Overcloud** 배포의 모든 환경 파일이 **NodeExtraConfigPost** 리소스를 정의하는 경우 **Heat**는 마지막 환경 파일에 정의된 **NodeExtraConfigPost** 를 사용합니다. 따라서 환경 파일의 순서가 중요합니다. 올바르게 처리 및 스택되도록 환경 파일을 주문하십시오.

주의

-e 옵션을 사용하여 **Overcloud**에 추가된 환경 파일은 **Overcloud** 스택 정의의 일부가 됩니다. **director**에는 배포 후 또는 재배포 기능에 이러한 환경 파일이 필요합니다. 이러한 파일을 포함하지 않으면 오버클라우드가 손상될 수 있습니다.

6장. 구성 가능 서비스

RHOSP(Red Hat OpenStack Platform)에는 사용자 지정 역할을 정의하고 역할에서 서비스 조합을 구성하는 기능이 포함되어 있습니다. 자세한 내용은 ***Advanced Overcloud Customization*** 가이드의 **{defaultURL}/advanced_overcloud_customization/chap-roles[Composable Services and Custom Roles]**를 참조하십시오. 통합의 일부로 고유한 사용자 지정 서비스를 정의하고 선택한 역할에 포함할 수 있습니다.

6.1. 구성 가능 서비스 아키텍처 검사

코어 **heat** 템플릿 컬렉션에는 구성 가능 서비스 템플릿 두 세트가 포함되어 있습니다.

- **Puppet/services** 에는 구성 가능 서비스를 구성하기 위한 기본 템플릿이 포함되어 있습니다.
- **Docker/services** 에는 주요 **OpenStack Platform** 서비스에 대한 컨테이너화된 템플릿이 포함되어 있습니다. 이러한 템플릿은 일부 기본 템플릿에 대해 보강하고 기본 템플릿으로 다시 참조합니다.

각 템플릿에는 용도를 식별하는 설명이 포함되어 있습니다. 예를 들어 **ntp.yaml** 서비스 템플릿에는 다음 설명이 포함되어 있습니다.

```
description: >
  NTP service deployment using puppet, this YAML file
  creates the interface between the HOT template
  and the puppet manifest that actually installs
  and configure NTP.
```

이러한 서비스 템플릿은 **RHOSP** 배포에 고유한 리소스로 등록됩니다. 즉, **overcloud-resource-registry-puppet.j2.yaml** 파일에 정의된 고유한 **heat** 리소스 네임스페이스를 사용하여 각 리소스를 호출할 수 있습니다. 모든 서비스는 해당 리소스 유형에 **OS::TripleO::Services** 네임스페이스를 사용합니다.

일부 리소스는 기본 구성 가능 서비스 템플릿을 직접 사용합니다.

```
resource_registry:
  ...
  OS::TripleO::Services::Ntp: puppet/services/time/ntp.yaml
  ...
```

그러나 코어 서비스에는 컨테이너가 필요하며 컨테이너화된 서비스 템플릿을 사용합니다. 예를 들어,

keystone 컨테이너화된 서비스는 다음을 사용합니다.

```
resource_registry:
  ...
  OS::TripleO::Services::Keystone: docker/services/keystone.yaml
  ...
```

이러한 컨테이너화된 템플릿은 일반적으로 **Puppet** 구성을 포함하도록 기본 템플릿을 참조합니다. 예를 들어 **docker/services/keystone.yaml** 템플릿은 기본 템플릿의 출력을 **KeystoneBase** 매개변수에 저장합니다.

```
KeystoneBase:
  type: ../../puppet/services/keystone.yaml
```

그런 다음 컨테이너화된 템플릿은 기본 템플릿의 기능과 데이터를 통합할 수 있습니다.

overcloud.j2.yaml heat 템플릿에는 **roles_data.yaml** 파일에서 각 사용자 지정 역할에 대한 서비스 목록을 정의하는 **Jinja2** 기반 코드 섹션이 포함되어 있습니다.

```
{{role.name}}Services:
  description: A list of service resources (configured in the Heat
               resource_registry) which represent nested stacks
               for each service that should get installed on the {{role.name}} role.
  type: comma_delimited_list
  default: {{role.ServicesDefault|default([])}}
```

기본 역할의 경우 다음과 같은 서비스 목록 매개변수인 **ControllerServices,ComputeServices,BlockStorageServices,ObjectStorageServices, CephStorageServices** 가 생성됩니다.

roles_data.yaml 파일에서 각 사용자 지정 역할에 대한 기본 서비스를 정의합니다. 예를 들어 기본 **Controller** 역할에는 다음 콘텐츠가 포함됩니다.

```
- name: Controller
  CountDefault: 1
  ServicesDefault:
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CephMon
    - OS::TripleO::Services::CephExternal
    - OS::TripleO::Services::CephRgw
    - OS::TripleO::Services::CinderApi
    - OS::TripleO::Services::CinderBackup
    - OS::TripleO::Services::CinderScheduler
    - OS::TripleO::Services::CinderVolume
```

```
    - OS::TripleO::Services::Core
    - OS::TripleO::Services::Kernel
    - OS::TripleO::Services::Keystone
    - OS::TripleO::Services::GlanceApi
    - OS::TripleO::Services::GlanceRegistry
...
```

그러면 이러한 서비스가 **ControllerServices** 매개변수의 기본 목록으로 정의됩니다.

환경 파일을 사용하여 서비스 매개변수의 기본 목록을 덮어쓸 수도 있습니다. 예를 들어 환경 파일에서 **ControllerServices** 를 **parameter_default** 로 정의하여 **roles_data.yaml** 파일의 서비스 목록을 덮어쓸 수 있습니다.

6.2. 사용자 정의 구성 가능 서비스 생성

이 예제에서는 사용자 정의 구성 가능 서비스를 생성하는 방법을 검토하고 일(**motd**) 서비스의 메시지 구현에 중점을 둡니다. 이 예에서 오버클라우드 이미지에는 설정 후크를 통해 로드되거나 오버클라우드 이미지를 수정하여 로드된 사용자 지정 **motd Puppet** 모듈이 포함되어 있습니다. 자세한 내용은 *3장. 오버클라우드 이미지 작업*의 내용을 참조하십시오.

자체 서비스를 생성할 때 서비스의 **heat** 템플릿에 다음 항목을 정의해야 합니다.

parameters

다음은 서비스 템플릿에 포함해야 하는 필수 매개변수입니다.

- **ServiceNetMap -** 네트워크에 대한 서비스 맵입니다. 이 매개변수가 상위 **Heat** 템플릿의 값으로 재정의되므로 빈 해시(**{}**)를 기본값 으로 사용합니다.
- **DefaultPasswords -** 기본 암호 목록입니다. 이 매개변수가 상위 **Heat** 템플릿의 값으로 재정의되므로 빈 해시(**{}**)를 기본값 으로 사용합니다.
- **EndpointMap -** 프로토콜에 대한 **OpenStack** 서비스 끝점 목록입니다. 이 매개변수가 상위 **Heat** 템플릿의 값으로 재정의되므로 빈 해시(**{}**)를 기본값 으로 사용합니다.

서비스에 필요한 추가 매개변수를 정의합니다.

outputs

다음 출력 매개 변수는 호스트에서 서비스 구성을 정의합니다. 자세한 내용은 부록 A. *구성 가능 서비스 매개변수*의 내용을 참조하십시오.

다음은 **motd** 서비스의 **heat** 템플릿(**service.yaml**)의 예입니다.

```
heat_template_version: 2016-04-08

description: >
  Message of the day service configured with Puppet

parameters:
  ServiceNetMap:
    default: {}
    type: json
  DefaultPasswords:
    default: {}
    type: json
  EndpointMap:
    default: {}
    type: json
  MotdMessage: 1
    default: |
      Welcome to my Red Hat OpenStack Platform environment!

    type: string
    description: The message to include in the motd

outputs:
  role_data:
    description: Motd role using composable services.
    value:
      service_name: motd
      config_settings: 2
        motd::content: {get_param: MotdMessage}
      step_config: | 3
        if hiera('step') >= 2 {
          include ::motd
        }
```

1

템플릿에는 오늘의 메시지를 정의하는 **MotdMessage** 매개 변수가 포함되어 있습니다. 매개 변수에는 기본 메시지가 포함되지만 사용자 지정 환경 파일에서 동일한 매개변수를 사용하여 재정의할 수 있습니다.

2

outputs 섹션에서는 **config_settings** 에서 일부 서비스 **hieradata**를 정의합니다. **motd::content hieradata**는 **MotdMessage** 매개변수의 콘텐츠를 저장합니다. **motd Puppet** 클래스는 이 **hieradata**를 읽고 사용자 정의 메시지를 **/etc/motd** 파일에 전달합니다.

3

outputs 섹션에는 **step_config** 에 **Puppet** 매니페스트 스니펫이 포함되어 있습니다. 코드 조각은 구성에 **2**단계에 도달했는지 확인하고 있는 경우 **motd Puppet** 클래스를 실행합니다.

6.3. 사용자 정의 구성 가능 서비스 포함

오버클라우드 컨트롤러 노드에서만 사용자 정의 **motd** 서비스를 구성할 수 있습니다. 이를 위해서는 배포에 포함된 사용자 지정 환경 파일 및 사용자 지정 역할 데이터 파일이 필요합니다. 요구 사항에 따라 이 절차의 예제 입력을 교체합니다.

절차

1. 환경 파일 **env-motd.yaml** 에 새 서비스를 **OS::TripleO::Services** 네임스페이스 내의 등록된 **heat** 리소스로 추가합니다. 이 예에서 **motd** 서비스의 리소스 이름은 **OS::TripleO::Services::Motd:**

```
resource_registry:
  OS::TripleO::Services::Motd: /home/stack/templates/motd.yaml

parameter_defaults:
  MotdMessage: |
    You have successfully accessed my Red Hat OpenStack Platform environment!
```

이 사용자 지정 환경 파일에는 **MotdMessage** 의 기본값을 재정의하는 새 메시지도 포함되어 있습니다.

이제 배포에 **motd** 서비스가 포함됩니다. 그러나 이 새 서비스가 필요한 각 역할에는 사용자 지정 **roles_data.yaml** 파일에 업데이트된 **ServicesDefault** 목록이 있어야 합니다.

2. 기본 **roles_data.yaml** 파일의 사본을 생성합니다.

```
$ cp /usr/share/openstack-tripleo-heat-templates/roles_data.yaml ~/custom_roles_data.yaml
```

3. 이 파일을 편집하려면 **Controller** 역할로 스크롤하여 **ServicesDefault** 목록에 서비스를 포함합니다.

```
- name: Controller
  CountDefault: 1
```

```
  ServicesDefault:
    - OS::TripleO::Services::CACerts
    - OS::TripleO::Services::CephMon
    - OS::TripleO::Services::CephExternal
...
    - OS::TripleO::Services::FluentdClient
    - OS::TripleO::Services::VipHosts
    - OS::TripleO::Services::Motd           # Add the service to the end
```

4. **오버클라우드를 생성할 때 다른 환경 파일 및 배포 옵션과 함께 결과 환경 파일 및 custom_roles_data.yaml 파일을 포함합니다.**

```
$ openstack overcloud deploy --templates -e /home/stack/templates/env-motd.yaml -r
~/custom_roles_data.yaml [OTHER OPTIONS]
```

여기에는 배포에 사용자 지정 motd 서비스가 포함되어 있으며 컨트롤러 노드에서만 서비스를 구성합니다.

7장. 인증된 컨테이너 이미지 빌드

파트너 빌드 서비스 를 사용하여 인증을 위해 애플리케이션 컨테이너를 빌드할 수 있습니다. **Build Service** 는 **SSH** 키를 사용하여 공개적으로 또는 개인적으로 액세스할 수 있는 **Git** 리포지토리에서 컨테이너를 빌드합니다.

이 섹션에서는 **Red Hat OpenStack** 및 **NFV Zone** 의 일부로 자동화된 **Build Service** 를 사용하여 컨테이너화된 파트너 플랫폼 플러그인을 **Red Hat OpenStack Platform 13** 기본 컨테이너로 자동으로 빌드하는 단계를 설명합니다.

사전 요구 사항

- **Red Hat Connect for Technology Partners**에 등록합니다.
- **Red Hat OpenStack** 및 **NFV** 영역에 대한 영역 액세스 권한을 적용합니다.
- 제품 생성. 사용자가 제공하는 정보는 인증서가 당사 카탈로그에 게시될 때 사용됩니다.
- **Dockerfile** 및 컨테이너에 포함할 모든 구성 요소를 사용하여 플러그인의 **git** 리포지토리를 생성합니다.

Red Hat Connect 사이트에 등록하거나 액세스할 때 문제가 있는 경우 Red Hat Technology Partner Success Desk 에 문의하십시오.

7.1. 컨테이너 프로젝트 추가

하나의 프로젝트는 하나의 파트너 이미지를 나타냅니다. 이미지가 여러 개 있는 경우 여러 프로젝트를 생성해야 합니다.

절차

1. Red Hat Connect for Technology Partners 에 로그인하고 영역을 클릭합니다.
2. 아래로 스크롤하여 **Red Hat OpenStack & NFV** 영역을 선택합니다. 상자의 아무 곳이나 클릭합니다.

3. **Certify** 를 클릭하여 회사의 기존 제품 및 프로젝트에 액세스합니다.

4. **Add Project** (프로젝트 추가)를 클릭하여 새 프로젝트를 만듭니다.

5. 프로젝트 이름 을 으로 설정합니다.

   - 프로젝트 이름은 시스템 외부에 표시되지 않습니다.
   - 프로젝트 이름에 **[product][version]-[extended-base-container-image]-[your-plugin]**이 포함되어야 합니다.
   - **OpenStack**의 용도는 **rhospXX-baseimage-myplugin** 입니다.
   - 예 : **rhosp13-openstack-cinder-volume-myplugin**

6. 제품 또는 플러그인에 따라제품, 제품 버전 및 릴리스 카테고리 를 선택하고 해당 버전을 선택합니다.

   - 프로젝트를 생성하기 전에 제품 및 해당 버전을 생성합니다.
   - 레이블 릴리스 범주를 기술 프리뷰로 설정합니다. 일반적으로 **Red Hat** 자격증으로 **API** 테스트를 마칠 때까지는 사용할 수 없습니다. 컨테이너 이미지를 인증한 경우 플러그인 인증 요구 사항을 참조하십시오.

7. 파트너 플러그인으로 수정하는 기본 이미지를 기반으로 **Red Hat** 제품 및 **Red Hat** 제품 버전을 선택합니다. 이번 릴리스에서는 **Red Hat OpenStack Platform** 및 **13** 을 선택하십시오.

8. **Submit** (제출)을 클릭하여 새 프로젝트를 만듭니다.

결과는 다음과 같습니다.

Red Hat은 프로젝트 인증을 평가 및 확인합니다.

플러그인이 트리에 있는지 아니면 업스트림 코드와 관련하여 트리 아웃인지 여부에 관계없이 connect@redhat.com에 이메일을 보냅니다.

- **Tree**는 플러그인이 **OpenStack** 업스트림 코드 베이스에 포함되어 있으며 플러그인 이미지는 **Red Hat**에서 빌드하고 **Red Hat OpenStack Platform {osp_curr_ver}**와 함께 배포됨을 의미합니다.

- **Out of Tree** 는 플러그인 이미지가 **OpenStack** 업스트림 코드 베이스에 포함되지 않고 **RHOSP {osp_curr_ver}** 내에 배포되지 않음을 의미합니다.

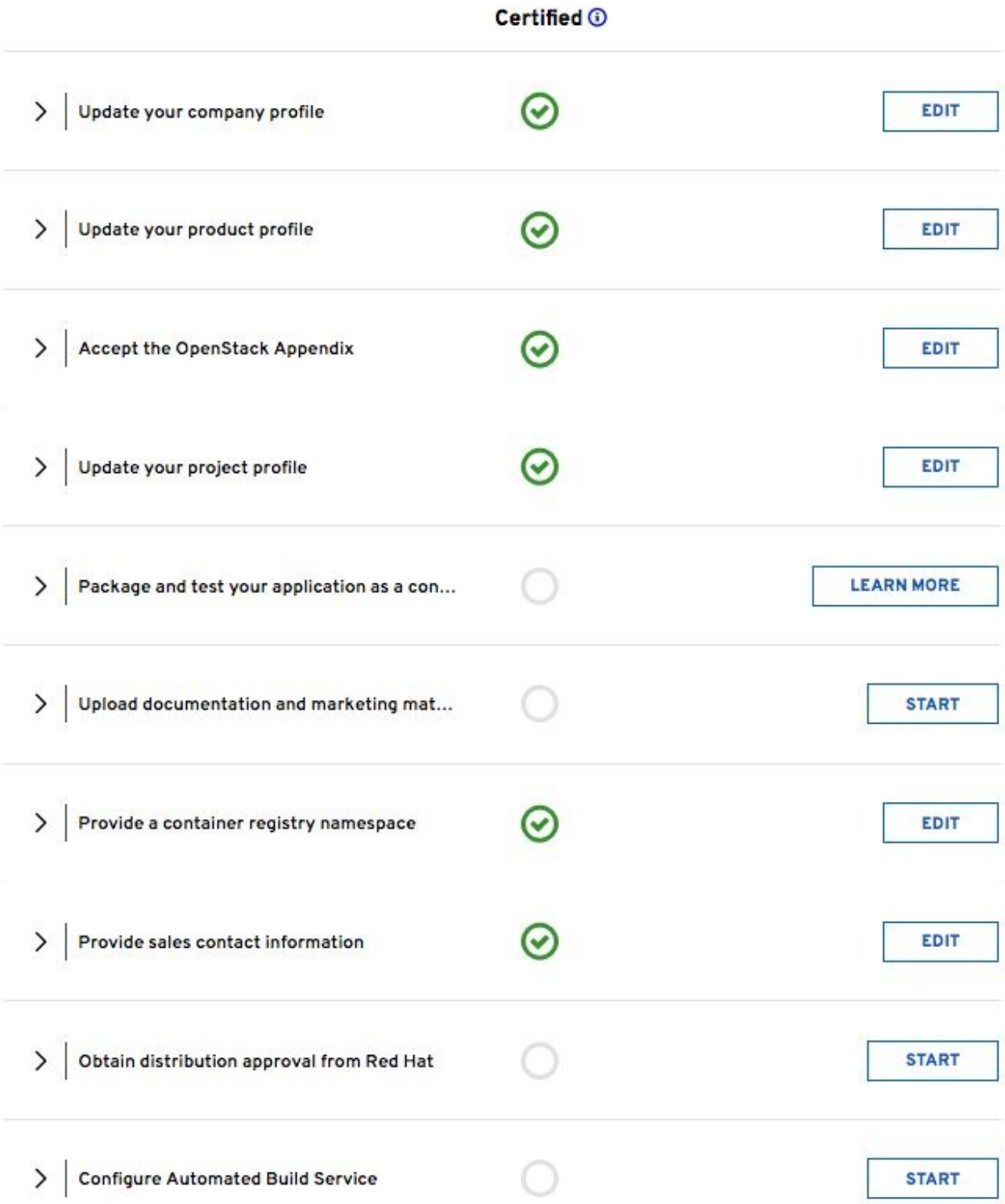
7.2. 컨테이너 인증 체크리스트 다음에

인증된 컨테이너는 패키징, 배포 및 유지 관리를 위한 **Red Hat** 표준을 충족합니다. **Red Hat**에서 인증한 컨테이너에는 **RHOSP(Red Hat OpenStack Platform)**를 포함하여 컨테이너 지원 플랫폼에서 높은 수준의 신뢰 및 지원이 가능합니다. 이를 유지하기 위해 파트너는 이미지를 최신 상태로 유지해야 합니다.

절차

1. **Certification Checklist** 를 클릭합니다.

2. 체크리스트의 모든 섹션을 완료하십시오. 체크리스트의 항목에 대한 자세한 정보가 필요한 경우 왼쪽의 드롭다운 화살표를 클릭하여 항목 정보 및 다른 리소스에 대한 링크를 확인합니다.



체크리스트는 다음 항목을 포함합니다.

회사 프로필 업데이트

회사 프로필이 최신 상태인지 확인합니다.

제품 프로파일 업데이트

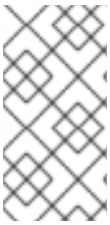
이 페이지는 제품 유형, 설명, 리포지토리 **URL**, 버전, 연락처 배포 목록을 포함하여 제품 프로필에 대한 세부 정보입니다.

OpenStack 부록 수락

컨테이너 약관에 대한 사이트 계약.

프로젝트 프로파일 업데이트

자동 게시, 레지스트리 네임스페이스, 릴리스 카테고리, 지원되는 플랫폼 등의 이미지 설정이 올바른지 확인합니다.



참고

지원 플랫폼 섹션에서 이 페이지의 다른 필수 필드를 저장할 수 있도록 옵션을 선택해야 합니다.

애플리케이션을 컨테이너로 패키징 및 테스트

이 페이지의 지침에 따라 빌드 서비스를 구성합니다. 빌드 서비스는 이전 단계의 완료에 따라 달라집니다.

문서 및 마케팅 자료 업로드

이렇게 하면 제품 페이지로 리디렉션됩니다. 하단으로 스크롤하고 추가를 클릭하여 제품 정보를 업로드합니다.

참고

최소 **3**개의 문서를 제출해야 합니다. 첫 번째는 문서 형식이 되어야 합니다.

컨테이너 레지스트리 네임스페이스 제공

이는 프로젝트 페이지 프로필 페이지와 동일합니다.

영업 연락처 정보 제공

이 정보는 회사 프로필과 동일합니다.

Red Hat 배포 승인 받기

Red Hat은 이 단계에 대한 승인을 제공합니다.

자동화된 빌드 서비스 구성

컨테이너 이미지의 빌드 및 검사를 수행하는 구성 정보입니다.

체크리스트의 마지막 항목은 자동화된 빌드 서비스 구성 입니다. 이 서비스를 구성하기 전에 프로젝트에 **Red Hat** 인증 표준을 준수하는 **dockerfile**이 포함되어 있는지 확인해야 합니다.

7.3. DOCKERFILE 요구사항

이미지 빌드 프로세스의 일부로 빌드 서비스는 빌드된 이미지를 검사하여 **Red Hat** 표준을 준수하는지 확인합니다. 프로젝트에 포함할 **dockerfile**의 기준으로 다음 지침을 사용합니다.

- 기본 이미지는 **Red Hat** 이미지여야 합니다. **Ubuntu, Debian** 및 **CentOS**를 기본으로 사용하는 이미지는 스캐너를 통과하지 않습니다.
- 필요한 라벨을 구성해야 합니다.
  - **name**
  - 유지 관리자
  - **vendor**
  - **version**
  - 릴리스
  - **summary**
- 소프트웨어 라이센스를 이미지 내에 텍스트 파일로 포함해야 합니다. 프로젝트 루트의 라이센스 디렉터리에 소프트웨어 라이센스를 추가합니다.

- **root** 사용자가 아닌 사용자를 구성해야 합니다.

다음 **dockerfile** 예제에서는 검사에 필요한 정보를 보여줍니다.

```
FROM registry.redhat.io/rhosp13/openstack-cinder-volume
MAINTAINER VenderX Systems Engineering <maintainer@vendorX.com>

###Required Labels
LABEL name="rhosp13/openstack-cinder-volume-vendorx-plugin" \
      maintainer="maintainer@vendorX.com" \
      vendor="VendorX" \
      version="3.7" \
      release="1" \
      summary="Red Hat OpenStack Platform 13.0 cinder-volume VendorX PluginY" \
      description="Red Hat OpenStack Platform 13.0 cinder-volume VendorX PluginY"


USER root

###Adding package
###repo exmple
COPY vendorX.repo /etc/yum.repos.d/vendorX.repo

###adding package with curl
RUN curl -L -o /verdorX-plugin.rpm http://vendorX.com/vendorX-plugin.rpm

###adding local package
COPY verdorX-plugin.rpm /

# Enable a repo to install a package
RUN yum clean all
RUN yum-config-manager --enable rhel-7-server-openstack-13-rpms
RUN yum install -y vendorX-plugin
RUN yum-config-manager --disable rhel-7-server-openstack-13-rpms

# Add required license as text file in Liceses directory (GPL, MIT, APACHE, Partner End User
Agreement, etc)
RUN mkdir /licenses
COPY licensing.txt /licenses

USER cinder
```

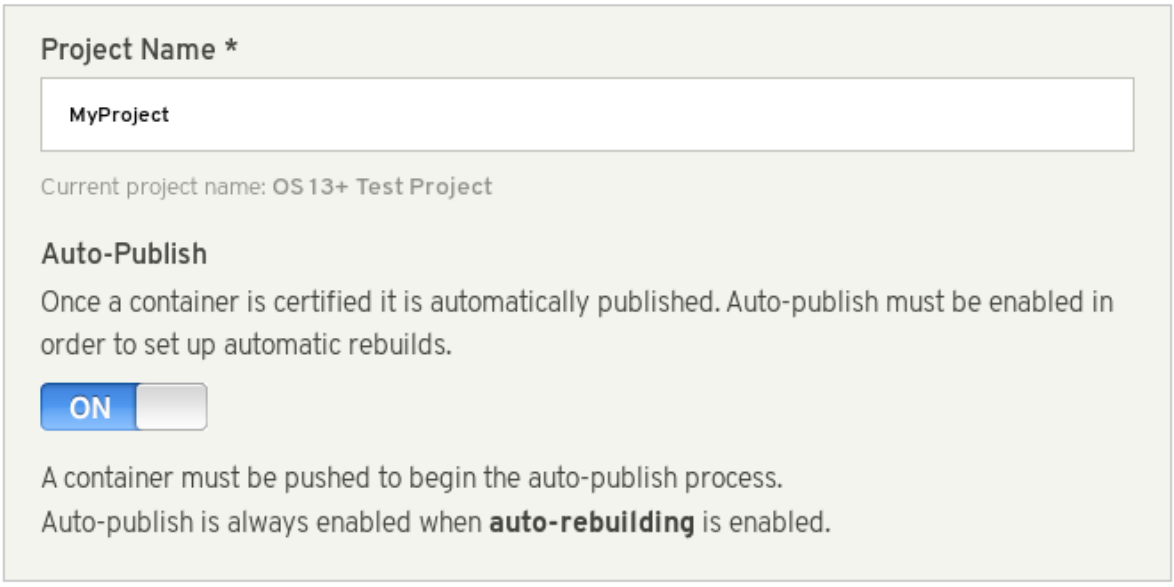
7.4. 프로젝트 세부 정보 설정

컨테이너 이미지의 네임스페이스 및 레지스트리를 포함하여 프로젝트의 세부 정보를 설정해야 합니다.

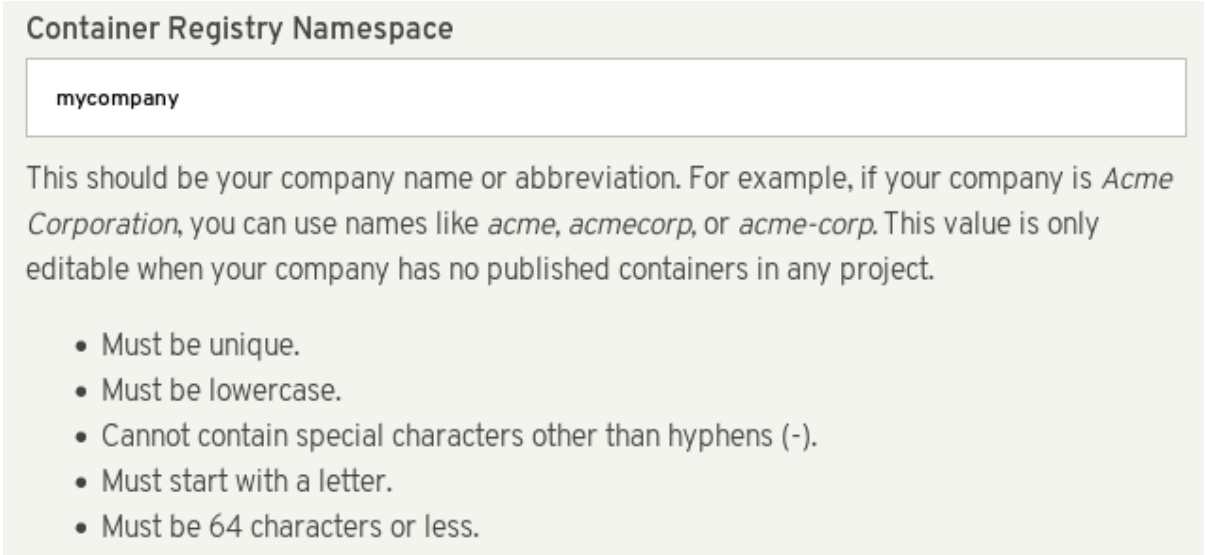
절차

1. **프로젝트 설정** 을 클릭합니다.

2. 프로젝트 이름이 올바른 형식인지 확인합니다. 인증을 전달하는 컨테이너를 자동으로 게시하려는 경우 **Auto-Publish** 를 **ON** 으로 설정합니다. 인증된 컨테이너는 **Red Hat Container Catalog**에 게시됩니다.

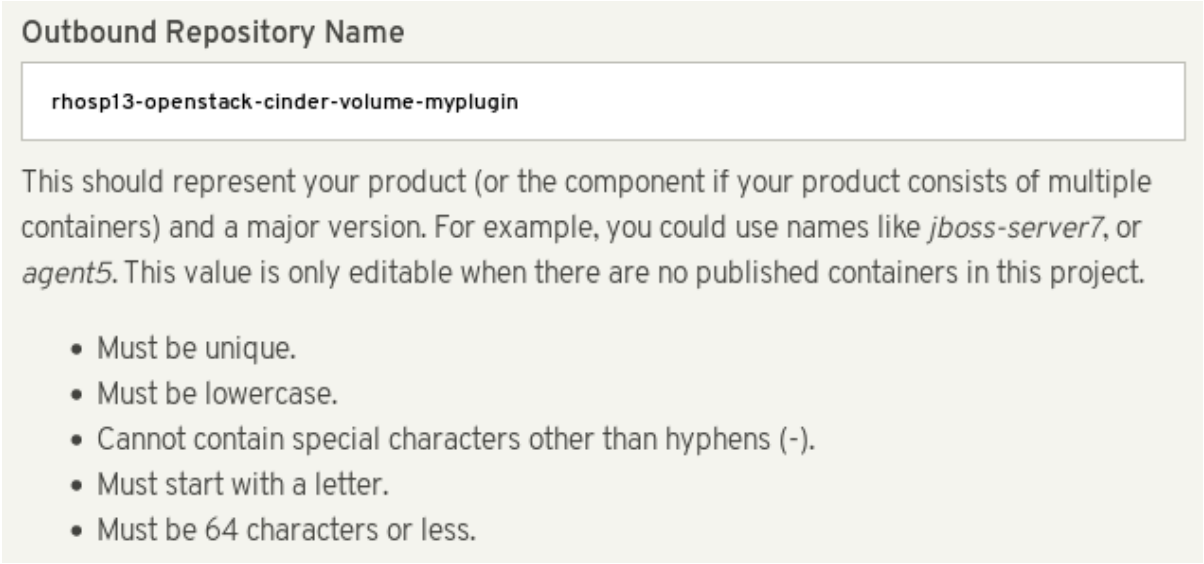


3. **컨테이너 레지스트리 네임스페이스** 를 설정하려면 온라인 지침을 따르십시오.



- 컨테이너 레지스트리 네임스페이스는 회사의 이름입니다.
- 최종 레지스트리 **URL**은 **registry.connect.redhat.com/namespace/repository:tag** 입니다.
- 예: **registry.connect.redhat.com/mycompany/myosp16-openstack-cinder-volume-myplugin:1.0**

4. **Outbound Repository Name** 및 **Outbound Repository Descriptions** 를 설정하려면 온라인 지침을 따르십시오. 아웃바운드 리포지토리 이름은 프로젝트 이름과 동일해야 합니다.



- **[product][version]-[extended_base_container_image]-[your_plugin]**
- **OpenStack**의 경우 형식은 **rhospXX-baseimage-myplugin**입니다.
- 그런 다음 최종 레지스트리 **URL**은 **registry.connect.redhat.com/namespace/repository:tag**입니다.
- 예: **registry.connect.redhat.com/mycompany/myosp13-openstack-cinder-volume-myplugin:1.0**

5. 관련 필드에서 프로젝트에 대한 추가 정보를 추가합니다.

- 리포지터리 설명
- 프라임드를 위한 문서 지원

6. 제출을 클릭합니다.

7.5. 빌드 서비스를 사용하여 컨테이너 이미지 빌드

파트너 플러그인의 컨테이너 이미지를 빌드합니다.

절차

1. **Build Service** 를 클릭합니다.
2. **Configure Build Service** (빌드 서비스 구성)를 클릭하여 빌드 세부 정보를 구성합니다.
   a. **Red Hat Container Build** 가 **ON** 으로 설정되어 있는지 확인합니다.
   b. **Git** 소스 **URL**을 추가하고 **git** 리포지토리가 보호되는 경우 소스 코드 **SSH** 키 를 선택적으로 추가합니다. **URL**은 **HTML** 또는 **SSH**일 수 있습니다. **SSH**는 보호된 **git** 리포지토리에 필요합니다.
   c. 선택 사항: **Dockerfile** 이름 을 추가하거나 **Dockerfile** 이름이 **Dockerfile** 인 경우 비워 둡니다.
   d. 선택 사항: **Docker** 빌드 컨텍스트 루트가 **git** 리포지토리의 루트가 아닌 경우 컨텍스트 디렉터리를 추가합니다. 그렇지 않으면 이 필드를 비워 둡니다.
   e. **Git** 리포지토리의 **Branch** 를 컨테이너 이미지를 기반으로 하도록 설정합니다.

f. **Submit (제출)**을 클릭하여 **Build Service** 설정을 완료합니다.

3. **Start Build (빌드 시작)**를 클릭합니다.

4. 태그 이름 을 추가하고 제출을 클릭합니다. 빌드를 완료하는 데 최대 6분이 걸릴 수 있습니다.

- 태그 이름은 플러그인의 버전이어야 합니다.
- 최종 참조 URL은 **registry.connect.redhat.com/namespace/repository:tag**입니다.
- 예: **registry.connect.redhat.com/mycompany/myosp13-openstack-cinder-volume-myplugin:1.0**

5. 새로 고침 을 클릭하여 빌드가 완료되었는지 확인합니다. 선택 사항: 일치하는 빌드 **ID** 를 클릭하여 빌드 세부 사항 및 로그를 확인합니다.

6. 빌드 서비스는 이미지를 빌드하고 스캔합니다. 일반적으로 완료하는 데 **10-15** 분이 걸립니다. 검사가 완료되면 **View (보기)** 링크를 클릭하여 검사 결과를 확장합니다.

7.6. 실패한 검사 결과 수정

Scan Details 페이지에 실패한 항목을 포함하여 검사 결과가 표시됩니다. 이미지 검사에서 **FAILED** 상태를 보고하는 경우 다음 절차에 따라 오류를 수정하는 방법을 조사하십시오.

절차

1. 컨테이너 정보 페이지에서 보기 링크를 클릭하여 검사 결과를 확장합니다.

2. 실패한 항목을 클릭합니다. 예를 들어 다음 스크린샷에서 **has_licenses** 검사가 실패했습니다.

Scan Details

Assessments

Name	Value
has_licenses	X
not_running_privileged	✓
rpm_list_successful	✓
rpm_verify_successful	✓
is_rhel	✓
vendor_label_exists	✓
free_of_critical_vulnerabilities	✓
good_tags	✓
good_layer_count	✓
release_label_exists	✓
not_running_as_root	✓
version_label_exists	✓
name_label_exists	✓

3. 실패한 항목을 클릭하여 관련 섹션에서 정책 가이드 를 열고 문제를 해결하는 방법에 대한 자세한 정보를 확인합니다.

참고

정책 가이드에 액세스할 때 액세스 거부 경고가 표시되면 이메일 (connect@redhat.com)

7.7. 컨테이너 이미지 게시

컨테이너 이미지가 검사를 통과한 후 컨테이너 이미지를 게시할 수 있습니다.

절차

1. **Container Information** 페이지에서 게시 링크를 클릭하여 컨테이너 이미지를 라이브로 게시합니다.

2. 게시 링크는 게시 취소 로 변경됩니다. 컨테이너 게시를 취소하려면 게시 취소 링크를 클릭합니다.

링크를 게시할 때 플러그인 인증에 대한 자세한 내용은 인증 문서를 확인하십시오. 인증 문서에 대한 자세한 내용은 1.1절. “파트너 통합 사전 요구 사항” 을 참조하십시오.

7.8. 벤더 플러그인 배포

타사 하드웨어를 블록 스토리지 백엔드로 사용하려면 벤더 플러그인을 배포해야 합니다. 다음 예제에서는 **Dell EMC** 하드웨어를 블록 스토리지 백엔드로 사용하기 위해 공급업체 플러그인을 배포하는 방법을 보여줍니다.

1. **registry.connect.redhat.com** 카탈로그에 로그인합니다.

```
$ docker login registry.connect.redhat.com
```

2. 플러그인을 다운로드합니다.

```
$ docker pull registry.connect.redhat.com/dellemc/openstack-cinder-volume-dellemc-rhosp13
```

3. **OpenStack** 배포와 관련된 언더클라우드 **IP** 주소를 사용하여 이미지에 태그를 지정하고 로컬 언더클라우드 레지스트리로 내보냅니다.

```
$ docker tag registry.connect.redhat.com/dellemc/openstack-cinder-volume-dellemc-rhosp13
192.168.24.1:8787/dellemc/openstack-cinder-volume-dellemc-rhosp13

$ docker push 192.168.24.1:8787/dellemc/openstack-cinder-volume-dellemc-rhosp13
```

4. 다음 매개변수가 포함된 추가 환경 파일을 사용하여 오버클라우드를 배포합니다.

```
parameter_defaults:
  DockerCinderVolumeImage: 192.168.24.1:8787/dellemc/openstack-cinder-volume-dellemc-
rhosp13
```

8장. OPENSTACK 구성 요소 통합 및 DIRECTOR 및 오버클라우드와의 관계

특정 통합 지점에 대한 다음 개념을 사용하여 하드웨어 및 소프트웨어를 RHOSP(Red Hat OpenStack Platform)와 통합할 수 있습니다.

8.1. BARE METAL PROVISIONING(IRONIC)

director 내에서 OpenStack Bare Metal Provisioning(ironic) 구성 요소를 사용하여 노드의 전원 상태를 제어합니다. director는 백엔드 드라이버 세트를 사용하여 특정 베어 메탈 전원 컨트롤러와 상호 작용합니다. 이러한 드라이버는 하드웨어 및 벤더 특정 확장 및 기능을 지원하는 데 중요한 요소입니다. 가장 일반적인 드라이버는 IPMI 드라이버pxe_ipmitool 입니다. 이 드라이버는 IPMI 드라이버로 IPMI(Intelligent Platform Management Interface)를 지원하는 모든 서버의 전원 상태를 제어합니다.

베어 메탈 프로비저닝과의 통합은 업스트림 OpenStack 커뮤니티와 함께 시작됩니다. 지원되는 Ironic 드라이버는 기본적으로 핵심 RHOSP 제품 및 director에 자동으로 포함됩니다. 그러나 인증 요구 사항에 따라 지원되지 않을 수 있습니다.

하드웨어 드라이버는 지속적인 기능을 보장하기 위해 지속적인 통합 테스트를 거쳐야 합니다. 타사 드라이버 테스트 및 적합성에 대한 자세한 내용은 OpenStack 커뮤니티 페이지 Ironic 테스트에서 참조하십시오.

업스트림 리포지토리:

- OpenStack: http://git.openstack.org/cgit/openstack/ironic/
- GitHub: https://github.com/openstack/ironic/

업스트림 Blueprints:

- Launchpad: http://launchpad.net/ironic

Puppet 모듈:

- GitHub: https://github.com/openstack/puppet-ironic

Bugzilla 구성 요소:

- **openstack-ironic**
- **python-ironicclient**
- **python-ironic-oscplugin**
- **openstack-ironic-discoverd**
- **openstack-puppet-modules**
- **openstack-tripleo-heat-templates**

통합 노트:

- 업스트림 프로젝트에는 **ironic/drivers** 디렉터리에 드라이버가 포함되어 있습니다.
- **director**는 **JSON** 파일에 정의된 노드를 대량으로 등록합니다. **os-cloud-config** 도구 https://github.com/openstack/os-cloud-config/ 는 이 파일을 구문 분석하여 노드 등록 세부 정보를 확인하고 등록을 수행합니다. 즉 **os-cloud-config** 도구, 특히 **nodes.py** 파일은 드라이버를 지원해야 합니다.
- **director**는 **Bare Metal Provisioning**을 사용하도록 자동 구성되어 있습니다. 즉 **Puppet** 구성은 수정하지 않아도 됩니다. 그러나 베어 메탈 프로비저닝에 드라이버가 포함된 경우 **/etc/ironic/ironic.conf** 파일에 드라이버를 추가해야 합니다. 이 파일을 편집하고 **enabled_drivers** 매개변수를 검색합니다.

```
enabled_drivers=pxe_ipmitool,pxe_ssh,pxe_drac
```

이렇게 하면 베어 메탈 프로비저닝에서 드라이버 디렉터리에서 지정된 드라이버 를 사용할 수 있습니다.

8.2. NETWORKING(NEUTRON)

OpenStack Networking(neutron)은 클라우드 환경 내에서 네트워크 아키텍처를 생성하는 기능을 제공합니다. 이 프로젝트는 소프트웨어 정의 네트워킹**(SDN)** 벤더에 대한 여러 통합 지점을 제공합니다. 이러한 통합 포인트는 일반적으로 플러그인 또는 에이전트 카테고리에 속합니다.

플러그인은 기존 **neutron** 기능을 확장 및 사용자 정의할 수 있습니다. 공급업체는 **neutron**과 인증된 소프트웨어와 하드웨어 간의 상호 운용성을 보장하기 위해 플러그인을 작성할 수 있습니다. 자체 드라이버 통합을 위한 모듈식 백엔드를 제공하는 **neutron Modular Layer 2(ml2)** 플러그인의 드라이버를 개발합니다.

에이전트는 특정 네트워크 기능을 제공합니다. 기본 **neutron** 서버와 해당 플러그인은 **neutron** 에이전트와 통신합니다. 기존 예제에는 **DHCP**, 계층 **3** 지원, 브리징 지원에 대한 에이전트가 있습니다.

플러그인과 에이전트 둘 다의 경우 다음 옵션 중 하나를 선택할 수 있습니다.

- **RHOSP(Red Hat OpenStack Platform)** 솔루션의 일부로 배포를 위해 이를 포함합니다.
- **RHOSP** 배포 후 오버클라우드 이미지에 추가

기존 플러그인 및 에이전트의 기능을 분석하여 인증된 하드웨어 및 소프트웨어를 통합하는 방법을 결정합니다. 특히, 먼저 **ml2** 플러그인의 일부로 드라이버를 개발하는 것이 좋습니다.

업스트림 리포지토리:

- **OpenStack:** http://git.openstack.org/cgit/openstack/neutron/
- **GitHub:** https://github.com/openstack/neutron/

업스트림 **Blueprints:**

- **Launchpad:** http://launchpad.net/neutron

Puppet 모듈:

- **GitHub:** https://github.com/openstack/puppet-neutron

Bugzilla 구성 요소:

- **openstack-neutron**
- **python-neutronclient**
- **openstack-puppet-modules**
- **openstack-tripleo-heat-templates**

통합 노트:

- 업스트림 **neutron** 프로젝트에는 다음과 같은 몇 가지 통합 지점이 있습니다.
  - 플러그인은 **neutron/plugins/**에 있습니다.
  - **ml2** 플러그인 드라이버는 **neutron/plugins/ml2/drivers/**에 있습니다.
  - 에이전트는 **neutron/agents/**에 있습니다.
- **OpenStack Liberty** 릴리스부터 많은 공급업체별 **ml2** 플러그인이 네트워킹 에서 시작되는 자체 리포지토리로 이동되었습니다. 예를 들어 **Cisco** 특정 플러그인은 https://github.com/openstack/networking-cisco에 있습니다.

- **puppet-neutron** 리포지토리에는 이러한 통합 지점을 구성하는 별도의 디렉터리도 포함되어 있습니다.
  - 플러그인 구성은 **manifests/plugins/**에 있습니다.
  - **ml2** 플러그인 드라이버 구성은 **manifests/plugins/ml2/**에 있습니다.
  - 에이전트 구성은 **manifests/agents/**에 있습니다.
- **puppet-neutron** 리포지토리에는 구성 기능을 위한 다양한 추가 라이브러리가 포함되어 있습니다. 예를 들어 **neutron_plugin_ml2** 라이브러리는 **ml2** 플러그인 구성 파일에 속성을 추가하는 함수를 추가합니다.

8.3. 블록 스토리지(CINDER)

OpenStack Block Storage(cinder)는 **RHOSP(Red Hat OpenStack Platform)**에서 볼륨을 생성하는 데 사용하는 블록 스토리지 장치와 상호 작용하는 **API**를 제공합니다. 예를 들어 **Block Storage**는 인스턴스에 대해 가상 스토리지 장치를 제공합니다. **Block Storage**는 다양한 스토리지 하드웨어 및 프로토콜을 지원하는 핵심 드라이버 세트를 제공합니다. 예를 들어 일부 핵심 드라이버에는 **NFS, iSCSI, Red Hat Ceph Storage** 지원이 포함됩니다. 공급업체는 추가 인증된 하드웨어를 지원하는 드라이버를 포함할 수 있습니다.

공급 업체에는 다음과 같은 두 가지 주요 옵션이 있으며 개발 중인 드라이버 및 구성:

- **RHOSP** 솔루션의 일부로 배포용 포함
- **RHOSP** 배포 후 오버클라우드 이미지에 추가

기존 드라이버의 기능을 분석하여 인증된 하드웨어 및 소프트웨어를 통합하는 방법을 확인합니다.

업스트림 리포지토리:

- OpenStack: http://git.openstack.org/cgit/openstack/cinder/
- GitHub: https://github.com/openstack/cinder/

업스트림 Blueprints:

- Launchpad: http://launchpad.net/cinder

Puppet 모듈:

- GitHub: https://github.com/openstack/puppet-cinder

Bugzilla 구성 요소:

- openstack-cinder
- python-cinderclient
- openstack-puppet-modules
- openstack-tripleo-heat-templates

통합 노트:

- 업스트림 cinder 리포지토리에는 cinder/volume/drivers/의 드라이버가 포함되어 있습니다.
- puppet-cinder 리포지토리에는 드라이버 설정을 위한 두 개의 기본 디렉터리가 포함되어 있습니다.

- **manifests/backend** 디렉터리에는 드라이버를 구성하는 정의된 유형 세트가 포함되어 있습니다.
- **manifests/volume** 디렉터리에는 기본 블록 스토리지 장치를 구성하는 클래스 세트가 포함되어 있습니다.

- **puppet-cinder** 리포지토리에는 **Cinder** 구성 파일에 속성을 추가하기 위한 **cinder_config** 라는 라이브러리가 포함되어 있습니다.

8.4. 이미지 스토리지(GLANCE)

OpenStack Image 서비스(**glance**)는 이미지용 스토리지를 제공하기 위해 스토리지 유형과 상호 작용하는 **API**를 제공합니다. 이미지 서비스는 다양한 스토리지 하드웨어 및 프로토콜을 지원하는 핵심 드라이버 세트를 제공합니다. 예를 들어 핵심 드라이버에는 파일, **OpenStack Object Storage(swift)**, **OpenStack Block Storage(cinder)** 및 **Red Hat Ceph Storage**에 대한 지원이 포함됩니다. 공급업체는 추가 인증된 하드웨어를 지원하는 드라이버를 포함할 수 있습니다.

업스트림 리포지토리:

- **OpenStack:**
  - http://git.openstack.org/cgit/openstack/glance/
  - http://git.openstack.org/cgit/openstack/glance_store/
- **GitHub:**
  - https://github.com/openstack/glance/
  - https://github.com/openstack/glance_store/

업스트림 **Blueprints:**

- **Launchpad:** http://launchpad.net/glance

Puppet 모듈:

- **GitHub:** https://github.com/openstack/puppet-glance

Bugzilla 구성 요소:

- **openstack-glance**
- **python-glanceclient**
- **openstack-puppet-modules**
- **openstack-tripleo-heat-templates**

통합 노트:

- 이미지 서비스에서 통합 지점이 포함된 블록 스토리지를 사용하여 이미지 스토리지를 관리할 수 있기 때문에 벤더별 드라이버를 추가할 필요가 없습니다.
- 업스트림 **glance_store** 리포지토리에는 **glance_store/_drivers** 의 드라이버가 포함되어 있습니다.
- **puppet-glance** 리포지토리에는 **manifests/backend** 디렉터리의 드라이버 구성이 포함되어 있습니다.
- **puppet-glance** 리포지토리에는 **Glance** 구성 파일에 속성을 추가하기 위한 **glance_api_config** 라는 라이브러리가 포함되어 있습니다.

8.5. 공유 파일 시스템(MANILA)

OpenStack Shared File System Service(Manila)는 공유 및 분산 파일 시스템 서비스에 대한 **API**를 제공합니다. 공급업체는 추가 인증된 하드웨어를 지원하는 드라이버를 포함할 수 있습니다.

업스트림 리포지토리:

- **OpenStack:** http://git.openstack.org/cgit/openstack/manila/
- **GitHub:** https://github.com/openstack/manila/

업스트림 **Blueprints:**

- **Launchpad:** http://launchpad.net/manila

Puppet 모듈:

- **GitHub:** https://github.com/openstack/puppet-manila

Bugzilla 구성 요소:

- **openstack-manila**
- **python-manilaclient**
- **openstack-puppet-modules**
- **openstack-tripleo-heat-templates**

통합 노트:s

- 업스트림 manila 리포지토리에는 manila/share/drivers/ 의 드라이버가 포함되어 있습니다.
- puppet-manila 리포지토리에는 manifests/backend 디렉터리에 드라이버 구성이 포함되어 있습니다.
- puppet-manila 리포지토리에는 manila 구성 파일에 속성을 추가하는 manila_config 라는 라이브러리가 포함되어 있습니다.

8.6. OPENSHIFT-ON-OPENSTACK

RHOSP(Red Hat OpenStack Platform)는 OpenShift-on-OpenStack 배포를 지원하는 것을 목표로 합니다. 이러한 배포의 파트너 통합에 대한 자세한 내용은 Red Hat OpenShift Partners 페이지를 참조하십시오.

부록 A. 구성 가능 서비스 매개변수

다음 매개변수는 모든 구성 가능 서비스의 출력에 사용됩니다.

- service_name
- config_settings
- service_config_settings
- global_config_settings
- step_config
- upgrade_tasks
- upgrade_batch_tasks

다음 매개변수는 컨테이너화된 구성 가능 서비스에 특히 출력에 사용됩니다.

- puppet_config
- kolla_config
- docker_config
- docker_puppet_tasks

- host_prep_tasks
- fast_forward_upgrade_tasks

A.1. 모든 구성 가능 서비스

다음 매개변수는 모든 구성 가능 서비스에 적용됩니다.

service_name

서비스 이름입니다. 이를 사용하여 service_config_settings 을 통해 다른 구성 가능 서비스의 구성을 적용할 수 있습니다.

config_settings

서비스에 대한 사용자 지정 **hieradata** 설정.

service_config_settings

다른 서비스의 사용자 지정 **hieradata** 설정. 예를 들어, 서비스에 **OpenStack ID(keystone)**에 등록된 엔드포인트가 필요할 수 있습니다. 이는 한 서비스에서 다른 서비스에 대한 매개 변수를 제공하고 서비스가 서로 다른 역할에 있더라도 교차 서비스 구성 방법을 제공합니다.

global_config_settings

모든 역할에 분산된 사용자 지정 **hieradata** 설정입니다.

step_config

서비스를 구성하는 **Puppet** 스니펫입니다. 이 스니펫은 서비스 구성 프로세스의 각 단계에서 생성되고 실행됩니다. 다음 단계는 다음과 같습니다.

- **1** 단계 **-** 로드 밸런서 구성
- **2** 단계 **-** 코어 고가용성 및 일반 서비스 **(**데이터베이스**, RabbitMQ, NTP)**

- 3 단계 - 초기 OpenStack Platform 서비스 설정 (스토리지, 링 빌딩)
- 4 단계 - 일반적인 OpenStack Platform 서비스
- 5단계 - 서비스 활성화(Pacemaker) 및 OpenStack Identity(keystone) 역할 및 사용자 생성

참조되는 모든 Puppet 매니페스트에서는 단계 hieradata(hiera(' step '))를 사용하여 배포 프로세스 중 특정 단계에서 특정 작업을 정의할 수 있습니다.

upgrade_tasks

서비스 업그레이드에 도움이 되는 Ansible 스니펫입니다. 코드 조각이 결합된 플레이북에 추가됩니다. 각 작업은 태그를 사용하여 다음을 포함하는 단계 를 정의합니다.

- Common - 모든 단계에 적용
- step0 - 검증
- step1 - 모든 OpenStack 서비스를 중지합니다.
- step2 - 모든 Pacemaker 제어 서비스 중지
- step3 - 패키지 업데이트 및 새 패키지 설치
- step4 - 데이터베이스 업그레이드에 필요한 OpenStack 서비스 시작
- step5 - 업그레이드 데이터베이스

upgrade_batch_tasks

upgrade_tasks 에 대해 유사한 기능을 수행하지만 나열된 순서대로 배치 Ansible 작업 세트만 실행합니다. 기본값은 1 이지만 roles_data.yaml 파일에서 upgrade_batch_size 매개변수를 사용하여 역할

별로 이 값을 변경할 수 있습니다.

A.2. 컨테이너화된 구성 가능 서비스

다음 매개변수는 컨테이너화된 구성 가능 서비스에 적용됩니다.

puppet_config

이 섹션은 **puppet**을 사용하여 구성 파일 생성을 유도하는 키 값 쌍의 중첩된 세트입니다. 필수 매개 변수는 다음과 같습니다.

puppet_tags

Puppet으로 구성 파일을 생성하는 데 사용되는 **Puppet** 리소스 태그 이름입니다. 명명된 구성 리소스만 파일을 생성하는 데 사용됩니다. 태그를 지정하는 모든 서비스에는 ***file,concat,file_line,augeas*** 의 기본 태그가 설정에 추가됩니다. 예: **keystone_config**

config_volume

이 서비스에 대한 구성 파일이 생성되는 볼륨(디렉터리)의 이름입니다. 구성을 위해 실행 중인 **Kolla** 컨테이너에 마운트를 바인딩하는 위치로 사용합니다.

config_image

구성 파일을 생성하는 데 사용할 **Docker** 이미지의 이름입니다. 이는 종종 런타임 서비스에서 사용하는 컨테이너와 동일합니다. 일부 서비스는 공통 기본 컨테이너에서 생성되는 공통 구성 파일 세트를 공유합니다.

step_config

이 설정은 **Puppet**을 통해 **Docker** 구성 파일을 생성하는 데 사용되는 매니페스트를 제어합니다. 아래 **Puppet** 태그는 이 매니페스트와 함께 사용하여 이 컨테이너의 구성 디렉터리를 생성합니다.

kolla_config

컨테이너에 **Kolla** 구성 맵을 생성합니다. 형식은 구성 파일의 절대 경로로 시작하여 다음 하위 매개변수에 사용합니다.

command

컨테이너가 시작될 때 실행할 명령입니다.

config_files

서비스를 시작하기 전에 서비스 구성 파일(소스)의 위치와 컨테이너(**dest**)의 대상입니다. 또한 파

일 권한 및 기타 속성(**preserve_properties**)을 유지할지 여부를 컨테이너에서 이러한 파일을병합하거나 교체하는 옵션을 포함합니다.

권한

컨테이너의 특정 디렉터리에 대한 권한을 설정합니다. 경로, 소유자 및 그룹이 필요합니다. 권한을 재귀적으로 적용할 수도 있습니다(재귀).

다음은 **keystone** 서비스에 대한 **kolla_config** 매개 변수의 예입니다.

```
kolla_config:
  /var/lib/kolla/config_files/keystone.json:
    command: /usr/sbin/httpd -DFOREGROUND
    config_files:
      - source: "/var/lib/kolla/config_files/src/*"
        dest: "/"
        merge: true
        preserve_properties: true
  /var/lib/kolla/config_files/keystone_cron.json:
    command: /usr/sbin/crond -n
    config_files:
      - source: "/var/lib/kolla/config_files/src/*"
        dest: "/"
        merge: true
        preserve_properties: true
    permissions:
      - path: /var/log/keystone
        owner: keystone:keystone
        recurse: true
```

docker_config

docker-cmd 후크에 전달된 데이터는 각 단계에서 컨테이너를 구성합니다.

- **step_0 - hiera** 설정별로 생성된 컨테이너 구성 파일입니다.
- **step_1 -** 로드 밸런서 구성
  a. **baremetal** 구성
  b. 컨테이너 구성

- **step_2 - Core Services (Database/Rabbit/NTP/etc)**

  a. **baremetal 구성**

  b. **컨테이너 구성**

- **step_3 - 초기 OpenStack 서비스 설정(Ringbuilder 등)**

  a. **baremetal 구성**

  b. **컨테이너 구성**

- **step_4 - 일반 OpenStack 서비스**

  a. **baremetal 구성**

  b. **컨테이너 구성**

  c. **Keystone 컨테이너 사후 초기화(테넌트, 서비스, 엔드포인트 생성)**

- **step_5 - 서비스 활성화 (Pacemaker)**

  a. **baremetal 구성**

  b. **컨테이너 구성**

YAML은 매개변수 세트를 사용하여 각 단계에서 실행되도록 컨테이너 컨테이너를 정의하고 각 컨테이너와 연결된 Docker 설정을 정의합니다. 예를 들면 다음과 같습니다.

```
docker_config:
  step_3:
    keystone:
      start_order: 2
      image: *keystone_image
      net: host
      privileged: false
      restart: always
      healthcheck:
        test: /openstack/healthcheck
      volumes: *keystone_volumes
      environment:
        - KOLLA_CONFIG_STRATEGY=COPY_ALWAYS
```

그러면 **keystone** 컨테이너가 생성되고 각 매개 변수를 사용하여 사용할 이미지, 네트워킹 유형 및 환경 변수를 포함하여 세부 정보를 정의합니다.

docker_puppet_tasks

docker-puppet.py 툴을 직접 구동하는 데이터를 제공합니다. 이 작업은 클러스터 내에서 한 번만 실행되며(각 노드에 아님) **keystone** 엔드포인트 및 데이터베이스 사용자와 같은 항목을 초기화하는 데 필요한 여러 **Puppet** 스니펫에 유용합니다. 예를 들면 다음과 같습니다.

```
docker_puppet_tasks:
  # Keystone endpoint creation occurs only on single node
  step_3:
    config_volume: 'keystone_init_tasks'
    puppet_tags:
'keystone_config,keystone_domain_config,keystone_endpoint,keystone_identity_provider,keystone_pas
e_ini,keystone_role,keystone_service,keystone_tenant,keystone_user,keystone_user_role,keystone_do
main'
    step_config: 'include ::tripleo::profile::base::keystone'
    config_image: *keystone_config_image
```

host_prep_tasks

이는 노드 호스트에서 실행할 수 있는 **ansible** 코드 조각으로 컨테이너화된 서비스를 준비합니다. 예를 들어 컨테이너를 생성하는 동안 컨테이너에 마운트할 특정 디렉터리를 생성해야 할 수 있습니다.

fast_forward_upgrade_tasks

빠른 진행 업그레이드 프로세스에 도움이 되는 **Ansible** 스니펫입니다. 이 스니펫은 결합된 플레이북에 추가됩니다. 각 작업에서는 태그를 사용하여 단계 및 릴리스를 정의합니다.

일반적으로 다음 단계 를 따릅니다.

- **step=0 -** 실행 중인지 확인
- **step=1 -** 서비스를 중지
- **step=2 -** 클러스터를 중지합니다.
- **step=3 -** 리포지토리 업데이트
- **step=4 -** 데이터베이스 백업
- **step=5 -** 사전 패키지 업데이트 명령
- **step=6 -** 패키지 업데이트
- **step=7 - Post-package** 업데이트 명령
- **step=8 -** 데이터베이스 업데이트
- **step=9 -** 검증

태그 는 릴리스에 해당합니다.

- **tag=ocata - OpenStack Platform 11**
- **tag=pike - OpenStack Platform 12**
- **tag=queens - OpenStack Platform 13**