How are OpenStack components related?

By V. K. Cody Bumgardner

If you'll be working with OpenStack on a high level, even if you're simply responsible for a vendor-managed solution, this article will help you understand the collection of interacting components that make up an OpenStack deployment.

This article is excerpted from Openstack in Action. Save 39% on Openstack in Action with code 15dzamia at manning.com.

Since the first public release of OpenStack in 2010, the framework has grown from a few core components to nearly ten. There are now hundreds of OpenStack-related projects, each with various levels of interoperability. These projects range from OpenStack library dependencies to projects where the OpenStack framework is the dependency.

In an effort to provide structure around the diverse set of projects, the OpenStack Foundation created several project designations, including core, incubated, library, gating, supporting, and related. These project designations and their descriptions can be found in table 1.
Table 1  Project designations

<table>
<thead>
<tr>
<th>Project designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Official OpenStack projects (most people use these)</td>
</tr>
<tr>
<td>Incubated</td>
<td>Core projects in development (on track to become core)</td>
</tr>
<tr>
<td>Library</td>
<td>Dependencies of core projects</td>
</tr>
<tr>
<td>Gating</td>
<td>Integration test suites and deployment tools</td>
</tr>
<tr>
<td>Supporting</td>
<td>Documentation and development infrastructure</td>
</tr>
<tr>
<td>Related</td>
<td>Unofficial projects (self-associated projects)</td>
</tr>
</tbody>
</table>

Incubated projects, once fully developed and accepted, will eventually function in the same way core projects do. Library functions will be abstracted (not observable) by core project interaction. Gating and supporting projects aren't used to provide resources in a deployed system, so you don't need to worry about those. That leaves the related projects, which as the name implies, have some affiliation with OpenStack, even if the affiliation is self-nominated.

**Understanding component communication**

Often when we refer to “OpenStack,” we’re referring to projects with a “core” designation. Core projects can use the OpenStack trademark and must pass all “must-pass” tests, as defined by the OpenStack Foundation. Simply put, core components are those that almost everyone will use in an OpenStack deployment. Projects like Compute, Networking, Storage, shared services, and Dashboard are examples of projects with a core designation, as shown in table 2 below.
Table 2  Core projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Codename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>Nova</td>
<td>Manages virtual machine (VM) resources, including CPU, memory, disk, and network interfaces</td>
</tr>
<tr>
<td>Networking</td>
<td>Neutron</td>
<td>Provides resources used by VM network interfaces, including IP addressing, routing, and software-defined networking (SDN)</td>
</tr>
<tr>
<td>Object Storage</td>
<td>Swift</td>
<td>Provides object-level storage accessible via RESTful API</td>
</tr>
<tr>
<td>Block Storage</td>
<td>Cinder</td>
<td>Provides block-level (traditional disk) storage to VMs</td>
</tr>
<tr>
<td>Identity Service (shared service)</td>
<td>Keystone</td>
<td>Manages role-based access control (RBAC) for OpenStack components; provides authorization services</td>
</tr>
<tr>
<td>Image Service (shared service)</td>
<td>Glance</td>
<td>Manages VM disk images; provides image delivery to VMs and snapshot (backup) services</td>
</tr>
<tr>
<td>Telemetry Service (shared service)</td>
<td>Ceilometer</td>
<td>Centralized collection for metering and monitoring OpenStack components</td>
</tr>
<tr>
<td>Orchestration Service (shared service)</td>
<td>Heat</td>
<td>Template-based cloud application orchestration for OpenStack environments</td>
</tr>
</tbody>
</table>
Database Service (shared service) | Trove | Provides users with relational and non-relational database services
--- | --- | ---
Dashboard | Horizon | Provides a web-based GUI for working with OpenStack

In addition to various project designations, there are also several topologies in which you can deploy project components. Specifically, if more of a specific resource (Storage, Compute, Networking, and so on) is required, more component-specific servers can be added.

**DASHBOARD AUTHENTICATION COMMUNICATION**

Let’s jump right in and take a look at how core components communicate. We’ll walk through the process of accessing the OpenStack Dashboard, reviewing the VM creation options, and creating a virtual machine.

You must first provide the Dashboard with your login credentials and obtain an authentication token. The authentication token will be saved as a cookie in your web browser and be used with subsequent requests. As shown in figure 1, you obtain an authentication token from the Identity service. While you can use the Dashboard (instead of the CLI or APIs) to navigate through the rest of this example, we won’t show the interaction with the Dashboard. Even during the login process, the Dashboard just displays interactions between the web browser and the OpenStack APIs. We’re primarily concerned with component interaction on the API level.

1 Dashboard login

1. A User supplies user credentials to log in: `<username> <password>`
2. B Check if user is who they say they are
3. C If user is correctly identified, supply an authentication token that they can use for the rest of their session

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/bumgardner/](http://www.manning.com/bumgardner/)
Welcome user to the OpenStack Dashboard, and provide token for authentication.

Once you have your authentication token, you can take the second step and access the Compute component to create your virtual machine (VM).

RESOURCE QUERY AND REQUEST COMMUNICATION

OpenStack works on a tenant model. If the OpenStack deployment is a hotel of resources, you can think of tenants as rooms in the hotel. Each tenant (room) is assigned a resource quota (a number of towels, beds, and so on). OpenStack users (guests) are assigned to tenants (rooms) through the use of roles. The identity information is kept by the Identity component, and the quota information is maintained by the Compute component.

In the Dashboard, when you click on the Launch Instance button, the Compute component is queried to determine what resources and configurations are available in your current tenant. Based on the available options, you describe the VM you want and submit the configuration for creation.

The communication between components during a VM creation request is shown in figure 2. Because the creation of a VM isn’t instantaneous, the process is executed asynchronously, so after you submit a VM for provisioning, you’re returned to the Dashboard. In the Dashboard, your browser will periodically update the VM status information.

Figure 2 Resource query and request

What resources are available for the creation of virtual machines?

You have a quota of X units of {CPU, RAM, storage} resources, access to private and public networks, and an image for Ubuntu Linux 12.04.
I want to create a VM named myVM, with {CPU: 2, RAM: 8 GB, storage: 40 GB} on the private network. Please load the Ubuntu Linux 12.04 image on myVM.

OK, I will start the process of provisioning myVM.

**RESOURCE PROVISIONING COMMUNICATION**

When VM creation requests are submitted, the Compute Service component will interact with other components to provision the requested VM. The first thing that happens is that the VM object record is registered with the Compute Service component. This object record contains information about the VM status and configuration—the VM object isn't the VM instance, only a record describing the instance.

When components communicate in the VM creation process, they reference common objects, like this VM object. For instance, the Compute Service component might request a storage assignment from the Storage Service component. The Storage Service component would then provision the requested storage and provide a reference to a Storage object, which would then be referenced in the VM object record.

As shown in figure 3, the Compute Service component communicates with other core components to provision and assign resources to the VM object. Compute will first request infrastructure components like Storage and Networking. When the virtual infrastructure has been assigned to the VM and referenced in the VM object, Compute will clone the requested operating system to the virtual storage volume. At this point, the VM creation process is complete and the VM is ready to start.

![Figure 3 Provisioning of resources](image-url)
Let’s create a VM instance named myVM with a resource reservation of {CPU: 2, RAM: 8 GB}.

I need a 40 GB volume for myVM.

OK, a 40 GB volume has been allocated for myVM.

I need a virtual adapter created on the “private” network for myVM.

OK, an adapter has been assigned to myVM and placed on the “private” network.

I need the Ubuntu Linux 12.04 image cloned to the 40 GB volume on myVM.

OK. I have cloned the image to the volume on myVM.

As demonstrated in the previous figures, core components work in concert to provide OpenStack services. OpenStack interactions, even those in the Dashboard, eventually find their way to the OpenStack API. Related projects will often use API calls exclusively to interact with OpenStack. As you’ll see next, related projects use the API exclusively for OpenStack interaction.

RELATED PROJECT COMMUNICATION

Let’s take a look at how Ubuntu Juju, a related project, interacts with OpenStack. Juju is a cloud automation package that uses OpenStack for virtual infrastructure. Juju automates the deployment and configuration of applications on virtual infrastructure using application-specific charms.

For the lack of a better description, Juju charms are collections of installation scripts that define how services and applications integrate into virtual infrastructure. Because infrastructure, including networks and storage, can be provisioned programmatically using OpenStack, Juju can deploy entire application suites from a charm. Simply, Juju turns newly provisioned VM instances into applications. We’ll discuss this process in detail in later chapters, but essentially you tell an application charm how large you want your instances to be and how many instances you want, and it does the work to deploy your applications.

The first step in using Juju in your OpenStack deployment is to deploy what Juju calls its bootstrap, using the Juju CLI. The bootstrap is a VM that Juju uses to control its automation processes. The deployment of the bootstrap, from a component perspective, is similar to what you’ve seen in recent figures (see figures 1, 2, and 3). The difference here is that in place of the web browser making the request, it’s the Juju application.

Once the bootstrap node has been started, Juju commands will be issued to the bootstrap node, not directly to OpenStack APIs. The reason for this is that the provisioning process is asynchronous, as mentioned earlier, and it’s sometimes time-consuming. You don’t want to maintain a connection from the desktop to the OpenStack deployment while a 20-VM application is deployed.
Let’s consider an example where you use Juju and OpenStack to deploy a load-balanced, five-node WordPress service, with a clustered MySQL backend. In this case, you have three types of service nodes: load-balancing, WordPress (Apache/PHP), and MySQL DB. Using the Juju charm developed for WordPress, you’d describe the number of nodes for each service, their virtual size (CPU, RAM, and so on), and how the nodes relate. You’d submit this charm to your bootstrap node, which would then interact with OpenStack to provision the application. This process is shown in figure 4.

![OpenStack interacting with a related project](image)

**Figure 4** OpenStack interacting with a related project

#A I want a five-node WordPress site with a pair of load balancers and a clustered database.

#B OK. I will start the process.

#C Create two LB_ instances with a resource reservation of {CPU: 2, RAM: 4 GB, storage: 10 GB}. Create five Web_ instances with a resource reservation of {CPU: 2, RAM: 8 GB, storage: 40 GB}. Create two DB_ instances with a resource reservation of {CPU: 4, RAM: 16 GB, storage: 80 GB}. Create a virtual network named WB_net between the LB_, Web_, and DB_ instances. Assign the “public” network to LB_.

#D OK. I will start the process.

Let’s assume that OpenStack, through the direction of the bootstrap node, successfully provisions all the required virtual infrastructure. At this point you have a collection of VMs, but no applications. The bootstrap node polls OpenStack, watching for its requested VMs to come online. Once the VMs are online, it will start a process outside the OpenStack framework to complete the install. As shown in figure 5, the bootstrap node will communicate directly with the newly provisioned VMs. From this point forward, OpenStack simply provides the virtual infrastructure and is unaware of the application roles assigned to each VM.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/bumgardner/](http://www.manning.com/bumgardner/)
Figure 5 Juju bootstrap controls the VMs

*A Install MySQL and configure an active/passive DB cluster using host DB_0 and DB_1
*B Install Apache, PHP, and WordPress using the database cluster DB_0/DB_1
*C Install HAProxy and load-balance web traffic for Web_0–Web_N.

We’ve now discussed how the components of OpenStack communicate on the logical level. In the figures we’ve illustrated component communication, as if everything was communicating inside a single big node (physical instance). In practice, however, OpenStack components will be distributed across many physical commodity servers in a multinode topology.