HashiCorp Nomad

by Tara Sawyer

In the BSDs we have RC scripts, on Windows we have Service Manager, and on Linux we have systemd. All of them are missing things or make things more difficult than perhaps they should be. Pretty much none of them are declarative or let you easily define (and limit) resources. What if we could have all the things, do them across all platforms, interact with Containers, and include all the hot new bells and whistles as well?

Nomad by HashiCorp arguably does this in a sane, declarative format. Nomad handles the entire lifecycle of the task, from allocating resources, setting up the environment, executing the task, monitoring the task for health, and then on any failure, restarting the task and any dependent tasks across any number of nodes. This makes it super trivial to do things like:

- Rolling upgrades
- Blue/Green (A/B) deployments
- Canary deployments
- Failure recovery of tasks, nodes, or datacenters.

Nomad is officially part of the HashiCorp Suite and is corporate sponsored yet is a fairly standard Github project, written in Golang and released under the MPL-2.0. This makes it usable for many of us who prefer BSD-licensed code.

There is an enterprise commercial version of Nomad by HashiCorp that gives you a few extras, but most everything is available in the MPL-licensed version. The commercial version gets you features like support, fine-grained access control, global resource quotas, and namespaces. Most of these features are not required for even large production usage.

Nomad's architecture is of an agent and server variety, but the same Golang binary is used for either mode, and can be completely self-contained with no external dependencies. However, running a multi-node server will likely also require running Consul, a distributed KV store. We won't cover production setup/installation here.

Nomad's server mode is a leader/follower model, typically running three servers per datacenter, plus an agent on every node that's executing tasks. For smaller deployments, you can run the agent stand-alone and not require a server instance. For larger deployments, Nomad is fully datacenter– and region–aware, and can easily handle 10,000 nodes in production.

A typical Nomad agent that runs on every node is not very resource intensive. On one of my production nodes, the agent consumes 58MB RSS and about 2% of my CPU and 3MB of /var disk to hold state (not including job tasks, log files, etc.).

Nomad cluster communication can be encrypted with little configuration required. There is a full capability-based access control system, so you can optionally configure access control. Without the enterprise version, access control is strictly based on tokens and does not cover authentication. In practice, this isn't an issue, as you can either gate your job submission through a CI/CD system or you can tie your token generation to HashiCorp Vault. Alternatively, you can pay HashiCorp for high-quality access control in their enterprise version.

Tasks

Each "task" or application can be just running some command on the machine or can result from running a command in some sort of container, be it QEMU, docker, rkt, Java, etc. These "task drivers" are plugin based and writing your own isn't that difficult and typically written in Go. Tasks are defined in a very declarative manner using HCL, which is a saner, more human-friendly version of JSON and is completely compatible with JSON if you so desire.

Tasks can easily be scheduled across 10,000-plus nodes in production and can be constrained by most any aspect of a given node, including by region or datacenter. So, from the smallest of deployments of one node to a full-scale enterprise system, Nomad can handle the job of getting your tasks running.

Tasks are scheduled in a variety of ways as well, from periodic (think crontab), batch (run once), service (run always), or system (think run on every node).

Task configuration is done in an HCL-based job file. There are basically three parts to a job file

- Constraints and groups of tasks for a given job.
- The resources you need (to include the files and artifacts needed to run the job to the CPU, memory, and network resources required).
- The operational and failure settings—this is how many instances to run, how to handle failures, and how to handle updates to the job.

Let's go through each of the three things:

Constraints and Groups

Every job belongs to a group, which is grouping tasks that need to run on the same node. Each group can have one or more tasks/applications to run.

Constraints let you handle where a job or task should run (or not run). You can do the normal Boolean comparisons of various things like equal, greater, and less than, etc., but we can also specify more complicated things like regex compar-

isons, set comparisons (like node lists), and version comparisons. A complete list of constraint operators is available in the documentation.

Resources

For a given task in a group, you need to define the resources, artifacts, which system/driver in which to execute the task (jails for us), the runtime environment, and any templated configuration to render into the task before start-up.

Nomad strongly encourages you to declaratively state every artifact/file and resource your job requires. This includes network ports, bandwidth, CPU, memory, and any special hardware (for instance, GPUs). This also includes any dynamic payloads required for map/reduce type batch jobs.

Operational and Failure Settings

This includes things like the scheduler (batch, system, service, periodic), affinities, restart, reschedule, spread, and service check items. This also includes how many instances to run.

In general, it's fairly robust in all the options available here, and I won't get into massive detail, but some things Nomad can handle are:

- If you have a leader/follower set of tasks, you can specify a task as a leader, and it will stop all the follower tasks for you when the leader stops.
- Rolling updates, starting X copies at a time, and ensuring each of the X copies are running and healthy (passes health checks) before continuing to start more copies.
- Blue/Green deployments. When upgrading from version 11.2 to 11.3, start all copies of 11.3 while not changing any 11.2 versions that are running. Ensure they are healthy and then manually promote the 11.3 versions. Nomad will then auto-stop the 11.2 versions. This can also support Canary deployments, where you start, say one instance of the new version, verify that it's happy, and then have it roll out all the other copies.
- Failure handling and fault tolerance. Besides specifying affinities for where a job or task should get deployed, Nomad can also handle many different failures. You can, for instance, spread a number of the same tasks across different nodes, so in a node failure, no users notice. Examples of this might be if a local resource you need stopped responding, or if the application crashed and needs to be restarted, or even if an application isn't responding in a certain amount of time and you want to reschedule the task. Nomad handles all of these situations directly.

Now that we have a pretty good overview of all the things Nomad can handle, let's get to the fun bits of actually deploying a very simple Python application into a jail from Nomad. All declaratively! But first, we need to install Nomad and get it running.

Installing and Running

On FreeBSD a `pkg install nomad` will just work.

On HardenedBSD currently two patches are required and must be built from source.

git/source Install

First install the dependencies

```
pkg install git
pkg install go
```

then as a normal user

```
export GOPATH=~/go
mkdir -p $GOPATH/src/github.com/hashicorp && cd $_
git clone https://github.com/hashicorp/nomad.git
cd nomad
```

then follow the patching instructions from this issue and this issue if on hardenedBSD. Finally build it:

```
go install
```

Then copy to /usr/local/bin/nomad and run Nomad as root in development mode:

```
cp ~/go/bin/nomad /usr/local/bin/nomad
```

To run Nomad:

```
nomad agent -dev
```

This will get Nomad up and running.

Now to make some jobs and run them.

Running it in production mode with multiple masters is beyond this guide. You can use `service nomad enable` and then `service nomad start`, but you will have to edit the configuration file, and all logging output is currently set to /dev/null, so especially at the beginning, just run it on its own terminal.

Running Your First Job

A very simple example job to fetch your IP:

```
job "fetch" {
        datacenters = ["dc1"]
        type = "batch"
        group "fetch" {
                count = 1
                task "fetch" {
                driver = "raw_exec"
                config {
                        command = "/usr/bin/fetch"
                        args = ["-q", "https://canhazip.com/", "-o", "-"]
                }
                resources {
                        cpu = 100
                        memory = 10
                }
                }
        }
}
```

It should be pretty self-explanatory, but a few things should be said. Every job file has one `job {}` definition. Inside of a job, you define any global constraints, like here we are saying run in the dc1 datacenter (which is the default dc the Nomad agents join). We also specify the type—here it's type batch because it is not a long-running service; it runs once, and then it's done. Groups, as we may remember, group tasks to a node. Count is the instance count. If you wanted to run 100 fetch commands, you would simply say count=100 and Nomad will take care of the rest. Tasks is the details, where we specify the driver to use the commands to run, and the resources required to run the task. Nothing overly exciting here. More details are available in the Nomad documentation.

Save this to a file fetch.nomad and then run:

```
$ nomad run fetch.nomad
nomad run fetch.nomad
==> Monitoring evaluation "7503713a"
    Evaluation triggered by job "fetch"
    Allocation "fe19e216" created: node "b8010185", group "fetch"
    Allocation "fe19e216" status changed: "pending" -> "complete" (All
tasks have completed)
    Evaluation status changed: "pending" -> "complete"
==> Evaluation "7503713a" finished with status "complete"
```

Nomad tasks create allocations of tasks; here you can see an allocation of "fe19e216". Your allocation ID will of course be different. You can also find the allocations by using the `nomad status <jobname>` command. To see what's happening with the task, we can ask for the allocation status:

```
$ nomad alloc status fe19e216
ID                  = fe19e216
Eval ID             = 7503713a
Name                = fetch.fetch[0]
Node ID             = b8010185
Node Name           = userbsd
Job ID              = fetch
Job Version         = 0
Client Status       = complete
Client Description  = All tasks have completed
Desired Status      = run
Desired Description = <none>
Created             = 2m6s ago
Modified            = 2m6s ago

Task "fetch" is "dead"
Task Resources
CPU        Memory      Disk     Addresses
0/100 MHz  0 B/10 MiB  300 MiB

Task Events:
Started At     = 2019-07-12T15:39:15Z
Finished At    = 2019-07-12T15:39:15Z
```

continues next page

continued

```
Total Restarts = 0
Last Restart   = N/A

Recent Events:
Time                       Type        Description
2019-07-12T08:39:15-07:00  Terminated  Exit Code: 0
2019-07-12T08:39:15-07:00  Started     Task started by client
2019-07-12T08:39:15-07:00  Task Setup  Building Task Directory
2019-07-12T08:39:15-07:00  Received    Task received by client
```

Here we can see the allocation, which node it was placed on, the resources allocated for this task, and all the events. We see it ran successfully and terminated with exit 0, so it successfully ran. The logs and output of the task can be gotten with the logs command:

```
$ nomad logs fe19e216
23.67.94.253
```

The IP address has been changed, I don't really work for the NSA, or do I? :)

We have the output from the fetch command. To get the standard error output just add a **`-stderr`** to the logs command, but if we did this correctly, there won't be any stderr output for this command. This, of course, all works across nodes and datacenters.

Let's run a simple go binary http-echo, which is basically a hello-world with http. But let's build it and a jail for it, and then run it inside of a jail. I think that would be much more interesting.

First, let's set up a webserver with which to deploy artifacts.

```
echo "run the following as root; i.e. sudo -i"
pkg install nginx
echo 'nginx_enable="YES"' > /etc/rc.conf.d/nginx
service nginx start
# where we will put the nomad artifacts
mkdir -p /usr/local/www/nginx/nomad
```

```
job "makebasejail" {
        datacenters = ["dc1"]
        type = "batch"
        group "makebasejail" {
                count = 1
                task "makebasejail" {
                        template {
                        data =  <<EOH
mkdir -p /usr/jails/basejail
BASEJAILD=/usr/jails/basejail
cd /usr/src
make world DESTDIR=$BASEJAILD
make distribution DESTDIR=$BASEJAILD
mkdir /usr/jails/basejail/etc
mkdir -p /usr/jails/basejail/dev
echo nameserver 1.1.1.1 > /usr/jails/basejail/etc/resolv.conf
```

continues next page

continued

```
cd /usr/jails/basejail
tar -czvf /usr/local/www/nginx/nomad/basejail.tgz .
shasum -a 256 /usr/local/www/nginx/nomad/basejail.tgz
>/usr/local/www/nginx/nomad/basejail.tgz.sum
EOH
                        destination = "local/makebasejail.sh"
                }
                        driver = "raw_exec"
                        config {
                        command = "/bin/sh"
                        args = ["local/makebasejail.sh"]
                        }
                        resources {
                                cpu = 100
                                memory = 100
                                }
                }
                task "make-http-echo-artifact" {
                        template {
                                data = <<EOH
pkg install -y go
go get github.com/hashicorp/http-echo
mkdir t ; cp ~/go/bin/http-echo t; cd t
tar -czvf /usr/local/www/nginx/nomad/http-echo.tgz .
shasum -a 256 /usr/local/www/nginx/nomad/http-echo.tgz
>/usr/local/www/nginx/nomad/http-echo.tgz.sum
EOH
                        destination = "local/make-http-echo.sh"
                }
                        driver = "raw_exec"
                        config {
                        command = "/bin/sh"
                        args = ["local/make-http-echo.sh"]
                        }
                        resources {
                               cpu = 100
                               memory = 100
                               }
                }

        }
}
```

This job file has two tasks, one to create a basejail by compiling the source, and the second task builds the http-echo program. Save this to something like build-hello-jail.nomad and run it. Since it has a lot of compilation, it will take a while—on my laptop about two hours. Nomad alloc status is your friend here. It should eventually complete, and nginx should now be serving up both the basejail tarball and the http-echo tarball.

We used a template `{}` here, which uses the Golang templating system, but I didn't use any templating; it's just a handy way to include a shell script to run in your job file.

Now we need a job to actually run the hello example in a jail:

```
job "hello" {
```

continues next page

continued

```
  datacenters = ["dc1"]

  group "example" {
    task "hello" {
      driver = "raw_exec"
  artifact {
        source      = "http://127.0.0.1/nomad/http-echo.tgz"
        destination = "/usr/local/bin"
        options {
          checksum =
"sha256:ae81e029018ace7a134dcbbf4531df98a1588cd3de6215f7f6c2bf971ce70992"
        }
  }
  artifact {
        source = "http://127.0.0.1/nomad/basejail.tgz"
        destination = "."
        options {
                checksum =
"sha256:552f3b5b237441e28fdef171b3f6cc2ab016c5fbea06ae62f02be58f4d2750d0"
        }
      }
      config {
        command = "/usr/sbin/jail"
        # jail -c jid=1 name=build exec.start=./sleep path=/zroot/jails/build
        args = [
        "-c",
        "name=${NOMAD_ALLOC_ID}",
        "exec.start=/usr/local/bin/http-echo -listen :5678 -text hello",
        # yes /.. is needed as NOMAD_TASK_DIR will be the rootfs from basejail.tgz +
local/
        "path=${NOMAD_TASK_DIR}/..",
        "ip4=inherit",
        "host.hostname=${NOMAD_ALLOC_NAME}",
        ]
      }

      resources {
        network {
          mbits = 10
          port "http" {
            static = "5678"
          }
        }
      }
    }
  }
```

This introduces the `artifact {}` sections, which you may need to modify a little bit, as the SHA256 checksums may be different for your tarballs. Luckily Nomad computed them for you and can be found in the .sum files like so

```
cat /usr/local/www/nginx/nomad/*.sum
```

to update the .nomad file with the correct checksums, if needed.

Save the file and run it (`nomad run <FILENAME>`), and we should see something fun in the alloc status:

```
$ nomad alloc status e247aa09
ID                  = e247aa09
Eval ID             = 652083ba
```

continues next page

continued

```
Name                = hello.example[0]
Node ID             = b8010185
Node Name           = userfbsd
Job ID              = hello
Job Version         = 1
Client Status       = running
Client Description  = Tasks are running
Desired Status      = run
Desired Description = <none>
Created             = 1m4s ago
Modified            = 26s ago

Task "hello" is "running"
Task Resources
CPU        Memory           Disk     Addresses
1/100 MHz  47 MiB/300 MiB   300 MiB  http: 127.0.0.1:5678

Task Events:
Started At     = 2019-07-12T16:24:34Z
Finished At    = N/A
Total Restarts = 0
Last Restart   = N/A

Recent Events:
Time                       Type                 Description
2019-07-12T09:24:34-07:00  Started              Task started by client
2019-07-12T09:24:06-07:00  Downloading Artifacts  Client is downloading artifacts
2019-07-12T09:24:06-07:00  Task Setup           Building Task Directory
2019-07-12T09:24:06-07:00  Received             Task received by client
```

The task is "running"! because this is type "service". Nomad will do it's very best to keep it always running until you **`nomad stop <jobname>`**, which for us is **`nomad stop hello`**. But before we stop the job, let's see a few things:

```
$ jls
   JID  IP Address      Hostname                      Path
   135                  hello.example[0]
/tmp/NomadClient875518057/e247aa09-d9d4-bd82-add9-6d8c63856f7b/hello
$ fetch -q -o - http://127.0.0.1:5678
hello
```

We can see the jail is up and going! The Path is showing a complete root jail from the artifacts we asked. Also, whenever we stop this job, we don't have to worry about cleaning up those /tmp/Nomad* directories; Nomad will garbage collect and clean up after itself.

Nomad created a fresh jail and ran the http-echo server. In fact, if you go exploring, you will see the parent directory also has an alloc/ directory, and inside that, you will find the logs. Of course, accessing it that way is difficult if running across multiple nodes, hence the **`nomad logs`** command, and you can actually look at the entire filesystem of the nomad jobs with the `nomad fs` command.

Note, my trap commands don't reliably work, but I haven't bothered debugging as there is a better way (read on!). For now, if the trap doesn't catch and stop the jails for you (you can tell with **`jls`** command), you can remove them with **`jail -r <JID>`** where the JID is in the output of the **`jls`**

command. Sorry for not having perfect shell fu!

Now let's make a python3 http.server jail:

```
job "python" {
  datacenters = ["dc1"]

  group "python" {
    task "python" {
      driver = "raw_exec"
  artifact {
        source = "http://127.0.0.1/nomad/basejail.tgz"
        destination = "."
        options {
                checksum =
"sha256:b69f0dbb020674ef09074f8f377a26fea3179e677bc54f7e9a23e2ea693e5826"
        }
      }
      config {
        command = "/bin/sh"
        args = ["local/start.sh"]
      }
#
#
# This builds 2 shell scripts in local/, start.sh which is what nomad runs, and then
run.sh
# which is what the jail itself runs.
# start.sh is responsible for creating run.sh at the moment.
# mostly because I was to lazy to create a new template{} entry in the job file.
#
#
      template {
        destination = "local/start.sh"
        data =  <<EOH
_term() {
    echo "Caught SIGINT signal!"
    echo "removing jail ${NOMAD_ALLOC_ID}"
    jail -r ${NOMAD_ALLOC_ID}
}
# use the above functions as signal handlers;
# note that the SIG* constants are undefined in POSIX,
# and numbers are to be used for the signals instead
# 2 == SIGINT
trap _term 2
#do startup stuff if needed, before turning up the jail.
echo "hello world" >> local/index.html
cd local
JAILDIR="${NOMAD_TASK_DIR}/.."
echo "jaildir: $JAILDIR"
# build run.sh which will be ran inside the jail after startup.
echo "pkg install -y python3" >run.sh
echo "cd /local" >> run.sh
echo "exec python3 -m http.server" >>run.sh
jail -c name=${NOMAD_ALLOC_ID} exec.start="/bin/sh local/run.sh" path="$JAILDIR"
ip4=inherit host.hostname=${NOMAD_ALLOC_NAME}
EOH
        }
      resources {
        network {
          mbits = 10
          port "http" {
            static = "8000"
          }
        }
      }
    }
  }
}
```

This is a much more complicated job file, but I'll explain what's happening. First, we download the basejail tarball and open that up, and then we create two shell scripts, one that calls the other. The first happens inside the Nomad jail root, but not inside of the jail, start.sh, which is responsible for doing any startup, creating the run.sh script, and then starting the jail, which runs the run.sh script.

Our setup is just creating a very simple html file to then serve via python3's built-in http server. `nomad run python.nomad` should get you all set up and running the simplest python http server around.

This should get you started, but running jails in Nomad is actually a lot easier than above, by using a brand new Nomad jail driver, which we will talk about next.

Jails Under Nomad the Sane Way

So, let's make running jails easier on ourselves. Why do it the hard way with all these shell scripts to build jails, etc.? Let's use a Nomad task driver that will do all the jail setup, teardown, and management for us:

https://github.com/cneira/jail-task-driver

Installation:

```
echo "fetch and build the source"
go get github.com/cneira/jail-task-driver
echo "install to /usr/local/nomad/plugins"
sudo mkdir -p /usr/local/nomad/plugins
sudo ln -s ~/go/bin/jail-task-driver /usr/local/nomad/plugins/jail-task-driver
echo plugin_dir=\"/usr/local/nomad/plugins\" > nomad.config
echo plugin \"jail-task-driver\" {} >nomad.config
sudo nomad agent -dev -config=nomad.config
```

This should get the plugin installed and the Nomad agent running with the config to load the plugin.

Now let's run the same http-echo job but under the Nomad driver:

```
job "http-echo" {
  datacenters = ["dc1"]

  group "example" {
    task "http-echo" {
      artifact {
        source      = "http://127.0.0.1/nomad/http-echo.tgz"
        destination = "/usr/local/bin"
  }
  artifact {
        source = "http://127.0.0.1/nomad/basejail.tgz"
        destination = "."
      }
      driver = "jail-task-driver"

      config {
        Persist  = false
        Ip4_addr = "127.0.0.1"
```

continues next page

continued

```
        Exec_start = "/usr/local/bin/http-echo -listen :5678 -text hello"
      }

      resources {
        network {
          mbits = 10
          port "http" {
            static = "5678"
          }
        }
      }
    }
  }
}
```

Run that and good things should happen:

```
$ nomad run http-echo.nomad
==> Monitoring evaluation "12947450"
    Evaluation triggered by job "http-echo"
    Allocation "eb589cec" created: node "d1e64602", group "example"
    Evaluation status changed: "pending" -> "complete"
==> Evaluation "12947450" finished with status "complete"
$
$ nomad status http-echo
ID            = http-echo
Name          = http-echo
Submit Date   = 2019-07-12T09:57:23-07:00
Type          = service
Priority      = 50
Datacenters   = dc1
Status        = running
Periodic      = false
Parameterized = false

Summary
Task Group  Queued  Starting  Running  Failed  Complete  Lost
example     0       0         1        0       0         0

Allocations
ID        Node ID   Task Group  Version  Desired  Status   Created  Modified
c285a2e3  d1e64602  example     0        run      running  27s ago  3s ago
$
$ fetch -q -o - http://127.0.0.1:5678
hello
```

YAY! You can see having a jail-task-driver makes the config a lot easier to understand and reason about!

The examples are probably not how you want to actually run and build Nomad jobs. Ideally, you would have a Nomad batch job (or ci/cd system) that would create a jail tarball all set up with your application in it. Version and deploy the resulting tarball to your webserver for Nomad to then deploy either in the same batch job or after some testing, etc. See the makebasejail example for a Nomad batch job that will do this first part.

Then you would have a Nomad job that would fetch the tarball, unpack it,

and run it as a service. This is mostly up to you, but the Nomad jobs in this article show you all the different bits and pieces. This will get you started with Nomad and jails. Further information can be found via the Nomad community and their documentation. Or reach out to me. I'm happy to help. ●

TARA SAWYER ran other people's software for a while, then wrote some software, then settled for mostly operations, where she has been running a production Nomad cluster for a few years. She recently got back into the BSDs after a hiatus in various other operating systems. Current likes are almonds and compassion. Tara has yet to manage a formal education.