Creating a Linux service with systemd

Benjamin Morel Follow
Sep 5, 2017 · 3 min read

Crafting your own services — Photo by Jeff Sheldon on Unsplash

While writing web applications, I often need to offload compute-heavy tasks to an asynchronous worker script, schedule tasks for later, or even write a daemon that listens to a socket to communicate with clients directly.

While there might sometimes be better tools for the job — always consider using existing software first, such as a task queue server —writing your own service can give you a level of flexibility you'll never get when bound by the constraints of third-party software.

The cool thing is that it's fairly easy to create a Linux service: use your favourite programming language to write a long-running program, and turn it into a service using systemd.

The program

Let's create a small server using PHP. I can see your eyebrows rising, but it works surprisingly well. We'll listen to UDP port 10000, and return any message received with a ROT13 transformation:

```php
<?php

$sock = socket_create(AF_INET, SOCK_DGRAM, SOL_UDP);
socket_bind($sock, '0.0.0.0', 10000);

for (;;) {
    socket_recvfrom($sock, $message, 1024, 0, $ip, $port);
    $reply = str_rot13($message);
    socket_sendto($sock, $reply, strlen($reply), 0, $ip, $port);
}
```

server.php hosted with ❤ by GitHub view raw

Let's start it:

```
$ php server.php
```

And test it in another terminal:

```
$ nc -u 127.0.0.1 10000
Hello, world!
Uryyb, jbeyq!
```

Cool, it works. Now we want this script to run at all times, be restarted in case of a failure (unexpected exit), and even survive server restarts. That's where systemd comes into play.

Turning it into a service

Let's create a file called `/etc/systemd/system/rot13.service`:

```
[Unit]
Description=ROT13 demo service
After=network.target
StartLimitIntervalSec=0

[Service]
Type=simple
Restart=always
RestartSec=1
User=centos
ExecStart=/usr/bin/env php /path/to/server.php

[Install]
WantedBy=multi-user.target
```

You'll need to:

- set your actual username after `User=`
- set the proper path to your script in `ExecStart=`

That's it. We can now start the service:

```
$ systemctl start rot13
```

And automatically get it to start on boot:

```
$ systemctl enable rot13
```

Going further

Now that your service (hopefully) works, it may be important to dive a bit deeper into the configuration options, and ensure that it will always work as you expect it to.

Starting in the right order

You may have wondered what the `After=` directive did. It simply means that your service must be started *after* the network is ready. If your program expects the MySQL server to be up and running, you should add:

```
After=mysqld.service
```

Restarting on exit

By default, systemd does not restart your service if the program exits for whatever reason. This is usually not what you want for a service that must be always available, so we're instructing it to always restart on exit:

```
Restart=always
```

You could also use `on-failure` to only restart if the exit status is not `0`.

By default, systemd attempts a restart after 100ms. You can specify the number of seconds to wait before attempting a restart, using:

```
RestartSec=1
```

Avoiding the trap: the start limit

I personally fell into this one more than once. By default, when you configure `Restart=always` as we did, **systemd gives up restarting your service if it fails to start more than 5 times within a 10 seconds interval.** Forever.

There are two `[Unit]` configuration options responsible for this:

```
StartLimitBurst=5
StartLimitIntervalSec=10
```

The `RestartSec` directive also has an impact on the outcome: if you set it to restart after 3 seconds, then you can never reach 5 failed retries within 10 seconds.

The simple fix that always works is to set `StartLimitIntervalSec=0`. This way, systemd will attempt to restart your service forever.

It's a good idea to set `RestartSec` to at least 1 second though, to avoid putting too much stress on your server when things start going wrong.

As an alternative, you can leave the default settings, and ask systemd to restart your server if the start limit is reached, using `StartLimitAction=reboot`.

Is that really it?

That's all it takes to create a Linux service with systemd: writing a small configuration file that references your long-running program.

Systemd has been the default init system in RHEL/CentOS, Fedora, Ubuntu, Debian and others for several years now, so chances are that your server is ready to host your homebrew services!

DevOps Linux Programming

About Help Legal