Если вы видите что-то необычное, просто сообщите мне.

Запуск различных программ через systemd

- Запуск java как сервиса через systemd
- Deploying a service using ansible and systemd
- docker-compose как сервис
- Run a Java Application as a Service on Linux
- Simple vs Oneshot - Выбираем тип systemd сервиса
- Podman systemd

Запуск java как сервиса через systemd

Предоставим у вас есть jar файл и вам нужно его запустить как сервис. Так же есть необходимость запускать его автоматически когда система перезагружается.

Убунту имеет встроенный механизм для создания сервисом в ручном режиме, запуск во время загрузи системы и останавливать и запускать как сервис. В этой статье мы сделаем простую сервис обертку для вашего jar файла, который можно будет запускать как сервис. Начнем.

1) Создадим сервис

```
sudo vim /etc/systemd/system/my-webapp.service
```

Копируем следующее содержание в файл `/etc/systemd/system/my-webapp.service`:

```
[Unit]
Description=My Webapp Java REST Service
[Service]
User=ubuntu
# Конфигурационный файл приложения application.properties должен быть тут:

# Замените на вашу рабочую папку
WorkingDirectory=/home/ubuntu/workspace

# Путь к экзешнику.
# Экзешник это bash скрипт, который вызывает jar файл
ExecStart=/home/ubuntu/workspace/my-webapp

SuccessExitStatus=143
TimeoutStopSec=10
```

```
Restart=on-failure
RestartSec=5

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

2) Создаём Bash скрипт для вызова сервисов.

Это bash скрипт, который вызывает JAR файл: my-webapp

```
#!/bin/sh
sudo /usr/bin/java -jar my-webapp-1.0-SNAPSHOT.jar server config.yml
```

Не забудем выдать скрипту права на исполнение:

```
sudo od u+x my-webapp
```

3) Запуск сервиса

```
sudo systemctl daemon-reload
sudo systemctl enable my-webapp.service
sudo systemctl start my-webapp
sudo systemctl status my-webapp
```

4) Настройка логирования

Первый запуск:

```
sudo journalctl --unit=my-webapp
```

Чтобы увидеть логи в реальном времени используйте опцию `-f`.

Если вы хотите обрезать логи, используйте `-n <# количество линий>`, чтобы увидеть нужное количество строк лога:

```
sudo journalctl -f -n 1000 -u my-webapp
```

Для отображения части логов используйте ключ `-f`:

```
sudo journalctl -f -u my-webapp
```

Остановите сервис с помощью команды:

```
sudo systemctl stop my-webapp
```

Deploying a service using ansible and systemd

You may be a sole developer or member of a small development team with no dedicated ops people. You will probably have a handful of small-ish services, perhaps a few cronjobs and a couple of VPSs to run them on. Or you may have one or more servers at home and would like to automate the deployment of custom or open source tools and services. What are your options?

At one end of the spectrum, there's the current kubernetes zeitgeist as recommended™ by the internetz. However, it may be that you can't pay the price (i.e. time) or simply do not have the desire to ride the steep learning curve that this path entails. On the other end of the spectrum, there's always rsync/scp and bash scripts but you'd like something better than that (including process management, logs, infrastructure as code checked into a git repo etc.). So, is there anything worthwile in between these two extremes?

This article is about how to deploy and run a service in a remote server using ansible and systemd. All the "configuration" that is neccessary to do that will be checked into a git repo and will be easily reproducible on an arbitrary set of servers (including your localhost) without the need to log into the servers and do any manual work (apart from setting up passwordless ssh access - but you already have that, right?). Now, a few words about the components that we are going to use.

Ansible is a tool for automating task execution in remote servers. It runs locally on your development machine and can connect to a specified set of servers via ssh in order to execute a series of tasks without the need of an "agent" process on the server(s). There's a wide variety of modules that can accomplish common tasks such as creating users and groups, installing dependencies, copying files and many more. We will focus on the absolutely necessary in this guide, but for those who would like to do more there are these nice tutorials as well as ansible's official documentation.

systemd is the basic foundation of most linux systems nowadays as the replacement of sysvinit and has a wide variety of features including managing processes and services (the feature that we'll be using for this article).

For our demonstration, we will be using a simple custom service written in Go, which very nicely and conveniently consists of a single statically-linked binary, but the concepts are the same for anything that can be executed on the remote server (this includes programs writen in ruby/python/java/dotnet etc.). So, let's start!

Prerequisites

We will be needing the following on our local (development) machine:

- a working Go installation in order to build our service
- the ansible tool
- the make program (check your system using which make) I have assumed that you have passwordless ssh access to a remote server running linux (I use Debian Buster but any linux system with sshd and systemd should do).

All the work that follows is checked into this repo which can be cloned using git clone https://github.com/kkentzo/deployment-ansible-systemd-demo.git. The repo contains the following components:

- `cmd/demo/main.go`: our service
- `demo.yml`: the description of our deployment (ansible)
- `roles/demo`: the deployment specifics of the demo dervice (ansible)
- `hosts`: the inventory list of hosts to which the demo service will be deployed
- `akefile`: targets for building and deploying the service

The Guide

Writing the service

Our service is a very simple one: it accepts http requests and responds with a greeting to the client based on the contents of the url path. The code is dead simple: package main

```
import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        var name string
        if name = r.URL.Path[1:]; name == "" {
            name = "stranger"
        }
        fmt.Fprintf(w, "hello %s!", name)
    })
    log.Fatal(http.ListenAndServe(":9999", nil))
}
```

The code above starts an http server that listens on port 9999. If the url path is the root path ("/") then the service greets the "stranger", otherwise it greets whoever is mentioned in the url path (e.g. GET /world will return "hello world!").

This file is placed under cmd/demo as main.go in our working directory and can be built in executable form (under bin/) as follows:

```
$ go build -o ./bin/demo ./cmd/demo/...
```

OK, so now we have our service - how about we deploy it?

Deploying the service

We will use ansible to deploy our service to our remote server as a systemd service unit. As mentioned before, the remote server can be any linux system with ssh and systemd. If you don't have access to such a system, you can use a tool such as virtual box in order to setup a debian buster system.

We will specify our remote server in our inventory (file hosts) for use by ansible:

```
[myservers]
harpo
```

As you can see, this file can declare multiple named server groups (names in [] brackets can be referenced in other ansible files). We have specified the section myservers which contains the name of our single server harpo. In this case, harpo is an alias defined in our .ssh/config file as follows:

```
Host harpo
     HostName 12.34.56.789
     User USERNAME
     IdentityFile ~/.ssh/harpo
```

This configuration facilitates ansible's access to the remote server (as mentioned before) and assumes that we have correctly set up access for user USERNAME in the server located in the address 12.34.56.789 (replace this with your own server's IP).

Now that we have specified our remote server, we need to define a role (workbook in ansible terminology) for our server as follows:

```
$ mkdir roles
$ cd roles
$ ansible-galaxy init demo
```

The above command will generate a file/directory structure under roles/demo of which the following are relevant to our guide:

- `roles/demo/tasks/main.ym`: the sequence of tasks to execute on the server
- `roles/demo/handlers/main.yml`: actions to execute when a task is completed
- `roles/demo/files/`: contains the files that we will need to copy to the remote server Let's start with the latter and define our systemd unit in file roles/demo/files/demo.service:

```
[Unit]
Description=Demo service

[Service]
User=demo
Group=demo
```

```
ExecStart=/usr/local/bin/demo

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

As you can see, systemd units are defined simply using a declarative language. In our case, we declare our service executable (ExecStart) that will run under user demo. The [Install] section specifies that our service requires a system state in which network is up and the system accepts logins.

Now, that we have our systemd unit, let's define our ansible playbook, starting from file roles/demo/tasks/main.yml:

```
---
- name: create demo group
  group:
    name: demo
    state: present

- name: create demo user
  user:
    name: demo
    groups: demo
    shell: /sbin/nologin
    append: yes
    state: present
    create_home: no

- name: Copy systemd service file to server
  copy:
    src: demo.service
    dest: /etc/systemd/system
    owner: root
    group: root
  notify:
    - Start demo

- name: Copy binary to server
  copy:
```

```
    src: demo
    dest: /usr/local/bin
    mode: 0755
    owner: root
    group: root
  notify:
    - Start demo
```

The task file is mostly self-explanatory but a few items need clarifications:

each task has a name and references an ansible module that accepts parameters

- ansible's group module creates the specified group if it does not exist
- ansible's user module creates users
- ansible's copy module copies files that exist locally under roles/demo/files (such as demo.service that we created previously) to the remote server Ansible's notify directive enqueues a particular handler (Start demo) to be executed after the completion of all tasks. All handlers are defined in file roles/demo/handlers/main.yml:

```
---
- name: Start demo
  systemd:
    name: demo
    state: started
    enabled: yes
```

This notification uses ansible's systemd module and requires the service to be started and enabled (i.e. started every time the remote server boots).

Finally, we complete our ansible configuration by combining our inventory and roles in file demo.yml:

```
---
- hosts: myservers
  become: yes
  become_user: root
  roles:
    - demo
```

Here, we declare that we would like to apply the role demo that we just defined to the specified host group (myservers as specified in our inventory file).

Wrap up

We're almost there! Let's wrap up the whole thing in a Makefile that contains the two targets of interest, build and deploy our service, as follows:

```
.PHONY: build
build:
    env GOOS=linux go build -o ./bin/demo ./cmd/demo/...

.PHONY: deploy
deploy: build
    cp ./bin/demo ./roles/demo/files/demo
    ansible-playbook -i hosts demo.yml
```

The build action compiles our service (for linux) and outputs the executable under bin/. The deploy target first builds the service, then copies the executable under the demo role's files and executes the entire ansible playbook by using the demo.yml spec.

Now, we can deploy our service by issuing:

```
$ make deploy
```

The output of this command on my machine was as follows:

```
make deploy
env GOOS=linux go build -o ./bin/demo ./cmd/demo/...
cp ./bin/demo ./roles/demo/files/demo
ansible-playbook -i hosts demo.yml

PLAY [home] ********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [harpo]
```

```
TASK [demo : create demo group] ************************************************
changed: [harpo]

TASK [demo : create demo user] *************************************************
changed: [harpo]

TASK [demo : Copy systemd service file to server] ******************************
changed: [harpo]

TASK [demo : Copy binary to server] ********************************************
changed: [harpo]

RUNNING HANDLER [demo : Start demo] ********************************************
changed: [harpo]

PLAY RECAP *********************************************************************
harpo                      : ok=6    changed=5    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

We can now test our service using curl:

```
$ curl 12.34.56.789:9999/world
```

where 12.34.56.789 needs to be replaced by your remote server's actual IP. If you see the output "hello world!", then you made it!

Status & Monitoring

We can also have a look on how the demo process is doing on our remote server by logging in (via ssh) and using the systemd commands systemctl (control and status) and journalctl (logs) as follows:

```
# check the status of our service
$ sudo systemctl status demo
# tail our service's logs
$ sudo journalctl -f -u demo
```

Further Work

This approach can be used to do pretty much anything on one or more remote servers in a consistent and robust manner. Beyond process management, systemd can also be used to schedule events (ala cronjobs) using timer units and manage logs using its own binary journal files and syslog.

Ansible's apt, shell and copy modules also facilitate the automated installation and configuration of standard software packages, even on the local machine using the "[local]" group name in the inventory file:

```
[local]
127.0.0.1
```

and executing any playbook using ansible-playbook's --connection=local command argument.

Epilogue

ansible and systemd are two fantastic tools that allow one to build automated, simple and reproducible operational pipelines quickly and efficiently.

All the contents of the service and the deployment code are in this repo.

I hope that you enjoyed this guide and found it useful! Please feel free to leave your comments or ask your questions.

docker-compose как сервис

Создаем папку с docker-compose в дирректории `/projectdirectory`.

Создаем файл /etc/systemd/system/НАЗВАНИЕСЕРВИСА.service:

```
[Unit]
Description=Docker Compose Service
Requires=docker.service
After=docker.service
[Service]
Type=oneshot
RemainAfterExit=true
WorkingDirectory=/projectdirectory
ExecStart=/usr/local/bin/docker-compose up -d --remove-orphans
ExecStop=/usr/local/bin/docker-compose down
[Install]
WantedBy=multi-user.target
```

Используем `systemctl` для управления:

Запуск/остановка:

```
systemctl start/stop НАЗВАНИЕСЕРВИСА
```

Включение/отключение автозапуска:

```
systemctl enable/disable НАЗВАНИЕСЕРВИСА
```

Run a Java Application as a Service on Linux

Введение

Любое Java приложение с точки зрения системы это просто объект JVM. В этом коротком руководстве мы увидим, как мы можем сделать наше приложение сервисом.

Мы будем истользовать удобства пакета system, systemd это сервис управления системой в современных дистрибутивах Linux.

Здесь вы найдёте две реализации: одна для простого случая, другая - расширенная.

Простой сервис

В мире systemd, для создания системного сервиса, нам нужно подготовить файл и зарегистрировать его определённым способом. Начнём с содержания файла:

```
[Unit]
Description=My Java driven simple service
After=syslog.target network.target

[Service]
SuccessExitStatus=143

User=appuser
Group=appgroup

Type=simple
```

```
Environment="JAVA_HOME=/path/to/jvmdir"
WorkingDirectory=/path/to/app/workdir
ExecStart=${JAVA_HOME}/bin/java -jar javaapp.jar
ExecStop=/bin/kill -15 $MAINPID

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

Мы узнали тип сервиса простым потому что система начинает JVM процесс напрямую без создания дочернего процесса.

ExecStop указывает команду завершения, и systemd достаточно умен чтобы выяснить PID начального процесса. он автоматически создаёт MAINPID переменные окружения.

После, мы указываем systemd посылать 15 (SIGTERM) системный сигнал, чтобы завершить процесс.

Java создатели спроектировали его таким образом, чтобы он возвращал не нулевой код в случае если он завершён системным сигналом. Так как сигнал не нулевой, то ответ будет `128 + числовое значение сигнала`.

У казывая SuccessExitStatus как 143, мы говорим systemd отловить это значение(128+15) как нормальное завершение

Форкаем сервис

Простой файл описания сервиса выше может быть довольно эффективным для простого приложения. Однако, во многих практических случаях, будут возможно включены дополнительные настройки.

Это может быть JVM параметры так же как любой другой параметр приложения, для примера, файл конфигурации или данных. Это модно свести к написанию обёртки для shell скрипта, где мы можем настроить все требуемые параметры прежде чем запустить JVM.

Предоставим, мы уже имеем обертку для скрипта, и теперь просто хотим включить это в сервис систем:

```
#!/bin/bash

JAVA_HOME=/path/to/jvmdir
WORKDIR=/path/to/app/workdir
JAVA_OPTIONS=" -Xms256m -Xmx512m -server "
APP_OPTIONS=" -c /path/to/app.config -d /path/to/datadir "

cd $WORKDIR
"${JAVA_HOME}/bin/java" $JAVA_OPTIONS -jar javaapp.jar $APP_OPTIONS
```

Так как мы используем shell скрипт чтобы запустить сервис, JVM будет запущена с помощью shell скрипта. Эта операция известна как `fork`, и поэтому мы указали тип как `forking`.

Перенесём опеределения переменных в тело скрипта:

```
[Unit]
Description=My Java forking service
After=syslog.target network.target
[Service]
SuccessExitStatus=143
User=appuser
Group=appgroup

Type=forking

ExecStart=/path/to/wrapper
ExecStop=/bin/kill -15 $MAINPID

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

Регистрируем и запускаем сервис

Не важно какой тип сервиса выбран, для выполнения задачи, мы должны знать, как настроить и запустить сам systemd сервис.

First, we need to name the unit file after the service name we want to have. In our examples, that could be javasimple.service or javaforking.service.

Then, we put the unit file under one of the locations where systemd can find it. For an arbitrary service, /etc/systemd/system is a good choice.

The full path to our system units, in that case, will be:

```
/etc/systemd/system/javasimple.service
/etc/systemd/system/javaforking.service
```

Another possible path to place system units is /usr/lib/systemd/system. This is typically the location used by the system installation packages.

However, we should consider it more appropriate when we develop our own .rpm or .deb installation packages containing system services.

In either case, we'll control the service using the systemctl utility and pass either the start, stop, or status command.

Before that, however, we should notify systemd that it has to rebuild its internal service database. Doing this will make it aware of the new system unit we introduced. We can do this by passing the daemon-reload command to systemctl.

Now, we're ready to run all the commands we mentioned:

```
sudo systemctl daemon-reload

sudo systemctl start javasimple.service
sudo systemctl status javasimple.service

● javasimple.service - My Java driven simple service
Loaded: loaded (/etc/systemd/system/javasimple.service; disabled; vendor preset: disabled)
Active: active (running) since Sun 2021-01-17 20:10:19 CET; 8s ago
Main PID: 8124 (java)
CGroup: /system.slice/javasimple.service
```

```
└─8124 /path/to/jvmdir/bin/java -jar javaapp.jar
```

We'll need to run the daemon-reload command each time we modify the unit file.

Next, we notice the system reports our service running but disabled. Disabled services will not start automatically when the system boots.

Of course, we can configure it to start up automatically along with the system. This is where we use another systemctl command — enable:

```
sudo systemctl enable javasimple.service
Created symlink from /etc/systemd/system/multi-user.target.wants/javasimple.service to
/etc/systemd/system/javasimple.service
```

Now, we can see that it's enabled:

```
sudo systemctl status javasimple.service
● javasimple.service - My Java driven simple service
Loaded: loaded (/etc/systemd/system/javasimple.service; enabled; vendor preset: disabled)
Active: active (running) since Sun 2021-01-17 20:10:19 CET; 14min ago
Main PID: 8124 (java)
....
```

5. Conclusion

In this article, we looked at two possible ways of turning Java applications into system service by means of systemd.

Java is still one of the most popular programming languages. A lot of Java applications are designed to run non-interactively for a variety of tasks, such as processing data, providing an API, monitoring events, and so on. Thus, they all are good candidates to become system services.

Simple vs Oneshot - Выбираем тип systemd сервиса

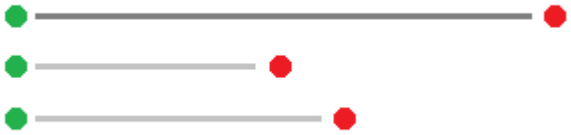
Этот пост довольно подробный, но если вы просто ищите общую информацию когда и какие типы сервсов использовать, читайте под катом.

Когда вы создаете свой `systemd` сервис, выбор типа сервиса может быть довольно сложен. Есть множество доступных и полезных типов сервисов, но этот пост сконцентрирован вокруг разниц между `oneshot` и `simple` простого сервисов. Возможно вас смущает какой из них использоваться.

Время запуска последующей единицы

Это наибольшая разница между `oneshot` и `simple` сервисами, когда стартует слудующая единица. Как указано в `man`: следующая единица простого сервиса стартует сразу же. На картинке ниже можете посмотреть:

Простой сервис и следующие за ним



Напротив же в `oneshot` сервисе, все последующие единицы дождутся заверешения сервиса прежде чем они стартанут.

`Oneshot` сервис и следующие за ним



Давайте рассмотрим простой пример сервиса и последующего за ним:

simple-test.service

```
[Unit]
Description=Simple service test

[Service]
Type=simple
ExecStart=/bin/bash -c "echo Simple service - start && sleep 60 && echo Simple service - end"
```

И зависимый сервис:

dep-simple-test.service

```
[Unit]
Description=Dependent service
After=simple-test.service
Requires=simple-test.service

[Service]
ExecStart=/bin/bash -c "echo Dependent service - running"
```

Запуск зависимого сервиса `dep-simple-test.service` запустит `simple-test.service` сначала(из-за `After/Requires` директив), а логи выведут следующее:

```
Jun 19 20:28:16 thstring20200619162314 systemd[1]: Started Simple service test.
Jun 19 20:28:16 thstring20200619162314 systemd[1]: Started Dependent service.
Jun 19 20:28:16 thstring20200619162314 bash[1238]: Simple service - start
Jun 19 20:28:16 thstring20200619162314 bash[1239]: Dependent service - running
Jun 19 20:28:16 thstring20200619162314 systemd[1]: dep-simple-test.service: Succeeded.
Jun 19 20:29:16 thstring20200619162314 bash[1238]: Simple service - end
Jun 19 20:29:16 thstring20200619162314 systemd[1]: simple-test.service: Succeeded.'
```

Простой пример(как и множество дальше) просто используют `sleep` для имитации работы сервиса. Так как `simple-test.service` это просто сервис, сразу за ним следует запуск `dep-simple-test.service`, и можно увидеть как оба сервиса стартуют в одно и то же время.

Но если мы сделаем тоже самое для `oneshot` сервиса, давайте посмотрим как различия выглядят.

oneshot-test.service

```
[Unit]
Description=Oneshot service test

[Service]
Type=oneshot
ExecStart=/bin/bash -c "echo Oneshot service - start && sleep 60 && echo Oneshot service - end"
```

dep-oneshot-test.service

```
[Unit]
Description=Dependent service
After=oneshot-test.service
Requires=oneshot-test.service

[Service]
ExecStart=/bin/bash -c "echo Dependent service - running"
```

Логирование для этих двух единиц(после запуска `dep-oneshot-test.service`) показывает разницу:

```
Jun 19 20:31:46 thstring20200619162314 systemd[1]: Starting Oneshot service test...
Jun 19 20:31:46 thstring20200619162314 bash[1420]: Oneshot service - start
Jun 19 20:32:46 thstring20200619162314 bash[1420]: Oneshot service - end
Jun 19 20:32:46 thstring20200619162314 systemd[1]: oneshot-test.service: Succeeded.
Jun 19 20:32:46 thstring20200619162314 systemd[1]: Started Oneshot service test.
Jun 19 20:32:46 thstring20200619162314 systemd[1]: Started Dependent service.
Jun 19 20:32:46 thstring20200619162314 bash[1440]: Dependent service - running
Jun 19 20:32:46 thstring20200619162314 systemd[1]: dep-oneshot-test.service: Succeeded.
```

Вы можете видеть как зависимый сервис не запускается пока `oneshot` сервис не завершится.

Состояния активации

Состояния активации различных типов сервисов управляют множеством взаимодействия с другими единицами.

Тип	До	Во время	После
Simple	inactive (dead)	active (running)	inactive (dead)
Oneshot	inactive (dead)	activating (start)	inactive (dead)
Oneshot (RemainAfterExit)	inactive (dead)	activating (start)	active (exited)

Состояние **Во время** различного состояния между `simple` и `oneshot` это причина почему следующая единица ждет завершения `oneshot` сервиса и почему не ждет завершения `simple` сервиса.

RemainAfterExit (oneshot)

Вы можете заметить такую директиву выше, `RemainAfterExit` меняет поведение `oneshot` сервиса не много. Это просто способ сказать `systemd` что после того как он выходит, он долже держать активное состояние. Для понимания, рассмотрим пример:

oneshot-remainafterexit.service

```
[Unit]
Description=Oneshot service test with RemainAfterExit

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/bin/bash -c "echo Oneshot service - start && sleep 60 && echo Oneshot service -
end"
```

Запустив `systemctl status` для этого сервиса во время работы, мы можем увидеть различия:

```
● oneshot-remainafterexit.service - Oneshot service test with RemainAfterExit
   Loaded: loaded (/etc/systemd/system/oneshot-remainafterexit.service; static; vendor preset:
enabled)
   Active: active (exited) since Fri 2020-06-19 20:55:14 UTC; 7s ago
  Process: 1174 ExecStart=/bin/bash -c echo Oneshot service - start && sleep 60 && echo
Oneshot service - end (code=exited, status=0/SUCCESS)
 Main PID: 1174 (code=exited, status=0/SUCCESS)

Jun 19 20:54:14 thstring20200619162314 systemd[1]: Starting Oneshot service test with
RemainAfterExit...
Jun 19 20:54:14 thstring20200619162314 bash[1174]: Oneshot service - start
Jun 19 20:55:14 thstring20200619162314 bash[1174]: Oneshot service - end
```

```
Jun 19 20:55:14 thstring20200619162314 systemd[1]: Started Oneshot service test with
RemainAfterExit.
```

Заметим что сервисв в `active(exited)` состоянии. вместо `inactive(dead)`(который должен быть, в случае если `RemainAfterExit` был отключен). Но если мы это хотим сохранить, что он делает на самом деле? Давайте посмотрим на пример, который использует `ExecStop` директиву. `ExecStop` запустится когда сервис остановится.

oneshot-execstop.service

```
[Unit]
Description=Oneshot service test with ExecStop

[Service]
Type=oneshot
RemainAfterExit=no
ExecStart=/bin/bash -c "echo Oneshot service - start && sleep 60 && echo Oneshot service -
end"
ExecStop=/bin/bash -c "echo Oneshot service - stop"
```

В этом сервисе `RemainAfterExit` отключен(это по-умолчанию, но добвлен для наглядности)

```
● oneshot-execstop.service - Oneshot service test with ExecStop
   Loaded: loaded (/etc/systemd/system/oneshot-execstop.service; static; vendor preset:
enabled)
   Active: inactive (dead)

Jun 19 21:04:10 thstring20200619162314 systemd[1]: Starting Oneshot service test with
ExecStop...
Jun 19 21:04:10 thstring20200619162314 bash[1480]: Oneshot service - start
Jun 19 21:05:10 thstring20200619162314 bash[1480]: Oneshot service - end
Jun 19 21:05:10 thstring20200619162314 bash[1604]: Oneshot service - stop
Jun 19 21:05:10 thstring20200619162314 systemd[1]: oneshot-execstop.service: Succeeded.
Jun 19 21:05:10 thstring20200619162314 systemd[1]: Started Oneshot service test with ExecStop.
```

Теперь видно, что `ExecStop` запускается сразу когду `ExecStart` выполнился, так как сервис перешел в состояние `inactive(dead)`. Теперь взглянем что случится с установленным

`RemainAfterExit`:set:

oneshot-execstop-remainafterexit.service

```
[Unit]
Description=Oneshot service test with ExecStop and RemainAfterExit

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/bin/bash -c "echo Oneshot service - start && sleep 60 && echo Oneshot service - end"
ExecStop=/bin/bash -c "echo Oneshot service - stop"
```

Вывод systemctl будет таков:

```
● oneshot-execstop-remainafterexit.service - Oneshot service test with ExecStop and RemainAfterExit
   Loaded: loaded (/etc/systemd/system/oneshot-execstop-remainafterexit.service; static; vendor preset: enabled)
   Active: active (exited) since Fri 2020-06-19 21:07:54 UTC; 8s ago
  Process: 1708 ExecStart=/bin/bash -c echo Oneshot service - start && sleep 60 && echo Oneshot service - end (code=exited, status=0/SUCCESS)
 Main PID: 1708 (code=exited, status=0/SUCCESS)

Jun 19 21:06:54 thstring20200619162314 systemd[1]: Starting Oneshot service test with ExecStop and RemainAfterExit...
Jun 19 21:06:54 thstring20200619162314 bash[1708]: Oneshot service - start
Jun 19 21:07:54 thstring20200619162314 bash[1708]: Oneshot service - end
Jun 19 21:07:54 thstring20200619162314 systemd[1]: Started Oneshot service test with ExecStop and RemainAfterExit.
```

Так как сервис до сих пор активен(даже не смотря на то завершился его `ExecStart`), `ExecStop` до сих пор не запущен. Теперь если вы запустите `systemctl stop oneshot-execstop-remainafterexit.service`, посмотрим на вывод:

```
● oneshot-execstop-remainafterexit.service - Oneshot service test with ExecStop and
RemainAfterExit
   Loaded: loaded (/etc/systemd/system/oneshot-execstop-remainafterexit.service; static;
vendor preset: enabled)
   Active: inactive (dead)

Jun 19 21:06:54 thstring20200619162314 systemd[1]: Starting Oneshot service test with ExecStop
and RemainAfterExit...
Jun 19 21:06:54 thstring20200619162314 bash[1708]: Oneshot service - start
Jun 19 21:07:54 thstring20200619162314 bash[1708]: Oneshot service - end
Jun 19 21:07:54 thstring20200619162314 systemd[1]: Started Oneshot service test with ExecStop
and RemainAfterExit.
Jun 19 21:08:58 thstring20200619162314 systemd[1]: Stopping Oneshot service test with ExecStop
and RemainAfterExit...
Jun 19 21:08:58 thstring20200619162314 bash[1900]: Oneshot service - stop
Jun 19 21:08:58 thstring20200619162314 systemd[1]: oneshot-execstop-remainafterexit.service:
Succeeded.
Jun 19 21:08:58 thstring20200619162314 systemd[1]: Stopped Oneshot service test with ExecStop
and RemainAfterExit.
```

Теперь видно, что `ExecStop` запущен так как сервис теперь неактивен. Это все, конечно, интересно, но systemctl не часто останавливает сервис. Вопрос, когда это будет полезно? Смотрим ниже...

Запуск сервиса при выключении

Создавая `oneshot` сервис с `ExecStop` и `RemainAfterExit`, это лучший способ для того, чтобы запустить что-то при включении. Посмотрим как выглядит на практике:

oneshot-execstop-remainafterexit-install.service

```
[Unit]
Description=Oneshot service test with ExecStop and RemainAfterExit

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/bin/bash -c "echo Oneshot service - start && sleep 60 && echo Oneshot service - end"
ExecStop=/bin/bash -c "echo Oneshot service - stop"

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

Затем запускаем `systemctl enable` чтобы включить сервис. Если мы запустим сервис, или перезагрузимся, то увидим:

```
● oneshot-execstop-remainafterexit-install.service - Oneshot service test with ExecStop and RemainAfterExit
   Loaded: loaded (/etc/systemd/system/oneshot-execstop-remainafterexit-install.service; enabled; vendor preset: enabled)
   Active: active (exited) since Fri 2020-06-19 21:14:02 UTC; 5s ago
 Main PID: 366 (code=exited, status=0/SUCCESS)
    Tasks: 0 (limit: 4087)
   Memory: 0B
   CGroup: /system.slice/oneshot-execstop-remainafterexit-install.service

Jun 19 21:13:02 thstring20200619162314 systemd[1]: Starting Oneshot service test with ExecStop and RemainAfterExit...
Jun 19 21:13:02 thstring20200619162314 bash[366]: Oneshot service - start
Jun 19 21:14:02 thstring20200619162314 bash[366]: Oneshot service - end
Jun 19 21:14:02 thstring20200619162314 systemd[1]: Started Oneshot service test with ExecStop and RemainAfterExit.
```

Как указано выше, наш `ExecStop` не запущен. Теперь перезапускаемся и сморим на логи:

```
-- Logs begin at Fri 2020-06-19 21:14:50 UTC, end at Fri 2020-06-19 21:18:47 UTC. --
Jun 19 21:14:51 thstring20200619162314 systemd[1]: Starting Oneshot service test with ExecStop and RemainAfterExit...
Jun 19 21:14:51 thstring20200619162314 bash[337]: Oneshot service - start
Jun 19 21:15:51 thstring20200619162314 bash[337]: Oneshot service - end
```

```
Jun 19 21:15:51 thstring20200619162314 systemd[1]: Started Oneshot service test with ExecStop
and RemainAfterExit.
Jun 19 21:17:48 thstring20200619162314 systemd[1]: Stopping Oneshot service test with ExecStop
and RemainAfterExit...
Jun 19 21:17:48 thstring20200619162314 bash[681]: Oneshot service - stop
Jun 19 21:17:49 thstring20200619162314 systemd[1]: oneshot-execstop-remainafterexit-
install.service: Succeeded.
Jun 19 21:17:49 thstring20200619162314 systemd[1]: Stopped Oneshot service test with ExecStop
and RemainAfterExit.
```

Что будет если, машина была выключена в 9 часов вечера, и это приведет к остановке сервиса, который заставит выключиться машину чуть позже из-за запуска команды из `ExecStop`. Это довольно простой способ для запуска чего-то во время выключения(например процесса очистки). А что еще лучше, это то что у вас нет `ExecStart` с `oneshot` сервисом. Дальше больше.

Множественные `ExecStarts`

Простой сервис, может только иметь один `Execstart` директиву. Но `oneshot` сервис может иметь один или больше, или вообще не иметь `ExecStart`. Если у вас нет `ExecStart`, тогда необходимо обязательно указать `ExecStop`(так же указать `RemainAfterExit`). Это будет сервис который запускается при выключении, и ни в какое другое время. Он напоминает `oneshot-execstop-remainaftgerexit-install.service` но с удаленным `ExecStart`.

Как сказано выше, `oneshot` сервис может иметь множество `ExecStarts`. Выглядить это буде так:

oneshot-multiple-execstart.service

```
[Unit]
Description=Oneshot service test with multiple ExecStart

[Service]
Type=oneshot
```

```
ExecStart=/bin/bash -c "echo First"
ExecStart=/bin/bash -c "echo Second"
ExecStart=/bin/bash -c "echo Third"
```

Как ожидали, лог будет следующим:

```
-- Logs begin at Mon 2020-06-22 13:24:01 UTC, end at Mon 2020-06-22 13:33:16 UTC. --
Jun 22 13:33:02 thstring20200622092223 systemd[1]: Starting Oneshot service test with multiple
ExecStart...
Jun 22 13:33:02 thstring20200622092223 bash[1316]: First
Jun 22 13:33:02 thstring20200622092223 bash[1317]: Second
Jun 22 13:33:02 thstring20200622092223 bash[1318]: Third
Jun 22 13:33:02 thstring20200622092223 systemd[1]: oneshot-multiple-execstart.service:
Succeeded.
Jun 22 13:33:02 thstring20200622092223 systemd[1]: Started Oneshot service test with multiple
ExecStart.
```

Объединим цепочку в `Execstart` действия, позволит нам создать мощный рабочий процесс прям внутри systemd единицы. Но что будет, если упадет один из `ExecStarts`?

oneshot-multiple-execstart-failure.service

```
[Unit]
Description=Oneshot service test with multiple ExecStart and failure

[Service]
Type=oneshot
ExecStart=/bin/bash -c "echo First"
ExecStart=/bin/bash -c "false && echo Second"
ExecStart=/bin/bash -c "echo Third"
```

Пытаясь запустить этот сервис, мы получим следующую ошибку:

```
$ sudo systemctl start oneshot-multiple-execstart-failure.service
Job for oneshot-multiple-execstart-failure.service failed because the control process exited
```

```
with error code.
See "systemctl status oneshot-multiple-execstart-failure.service" and "journalctl -xe" for
details.

$ sudo journalctl -u oneshot-multiple-execstart-failure.service
-- Logs begin at Mon 2020-06-22 13:24:01 UTC, end at Mon 2020-06-22 13:37:16 UTC. --
Jun 22 13:36:53 thstring20200622092223 systemd[1]: Starting Oneshot service test with multiple
ExecStart and failure...
Jun 22 13:36:53 thstring20200622092223 bash[1441]: First
Jun 22 13:36:53 thstring20200622092223 systemd[1]: oneshot-multiple-execstart-failure.service:
Main process exited, code=exited, status=1/FAILURE
Jun 22 13:36:53 thstring20200622092223 systemd[1]: oneshot-multiple-execstart-failure.service:
Failed with result 'exit-code'.
Jun 22 13:36:53 thstring20200622092223 systemd[1]: Failed to start Oneshot service test with
multiple ExecStart and failure.
```

Сервис падает и прерывает выполнение. Но, что если вы не хотите чтобы падение остановило сервис на середине? Мы можете добавить `-` символ перед выполнением команды

oneshot-multiple-execstart-failure-success.service

```
[Unit]
Description=Oneshot service test with multiple ExecStart and failure

[Service]
Type=oneshot
ExecStart=/bin/bash -c "echo First"
ExecStart=-/bin/bash -c "false && echo Second"
ExecStart=/bin/bash -c "echo Third"
```

Это не очевидно, но отметим, во втором `Exectart` что перед `/bin/bash` стоит `-`. Теперь посмотрим на вывод:

```
-- Logs begin at Mon 2020-06-22 13:24:01 UTC, end at Mon 2020-06-22 13:39:04 UTC. --
Jun 22 13:38:59 thstring20200622092223 systemd[1]: Starting Oneshot service test with multiple ExecStart and failure...
Jun 22 13:38:59 thstring20200622092223 bash[1553]: First
Jun 22 13:38:59 thstring20200622092223 bash[1555]: Third
Jun 22 13:38:59 thstring20200622092223 systemd[1]: oneshot-multiple-execstart-failure-success.service: Succeeded.
Jun 22 13:38:59 thstring20200622092223 systemd[1]: Started Oneshot service test with multiple ExecStart and failure.
```

Второй `ExecStart` упал как и ожидали, но это не уронило в целом сервис или остановило выполнение третей стадии.

Podman systemd

```
podman generate systemd --new --files --name pod-name
systemctl --user enable container-pod-name.service
systemctl --user  daemon-reload
systemctl --user start container-pod-name.service
systemctl --user status container-pod-name.service
```