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

# 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.

---

Revision #1
Created 2022-11-08 07:14:51 UTC by gasick
Updated 2023-04-16 19:38:12 UTC by gasick