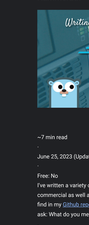
Writing Server Software With Go | by Jan Kammerath



~7 min read

June 25, 2023 (Updated: June 25, 2023)

Free: No
I've written a variety of different server software products. Both commercial as well as Open Source. Some of the examples you can find in my Github repositories. Before we dive in, you might rightfully ask: What do you mean by "server software"?

> Server software refers to the programs and applications installed on a server computer that enable it to provide specific services or perform certain functions. It is designed to manage and respond to client requests, handle data storage and retrieval, facilitate communication between different devices or applications, and perform other tasks related to server operations.

Server software is software like Apache, MySQL, MongoDB, Postfix, Redis, Memcache or any other software that provides network services and runs as a daemon process under systemd on Linux. I'll specifically focus on Linux, because the vast majority of servers out there on the Internet run on Linux.

Why write server software in the first place

The approach to many software solutions today is using standard stacks of software products like Linux, Apache, MySQL, Nginx, Redis or MongoDB. These are often applied as universal "tech stacks" for any solution. This may work for basic business applications. With more complex applications that involve audio, video, instant messaging, complex data structures or algorithms, you'll often find yourself having to write server software as system applications.

With the rise of Go and its simplicity it has also gotten significantly easier and faster to write system software and thus server applications with very good runtime performance. Hence, you may often need to rethink the approach of using a standard tech stack and may consider writing a bespoke server software application.

What Linux server software is made of

Linux has a very strict approach to how system services are managed with systemd. Before systemd, there was the SysV init system to manage initialization and system services. Further, the Linux specifications are also straightforward on where to put your files.

- Config files shall all go under */etc/<myservice>*
- Log files shall be written to */var/log/<myservice>*
- The binary shall reside in */usr/bin*
- Any data stored should be in */var/<myservice>*

For programming languages that compile into a single binary like C, C++, Rust and Go, the binary would go into the */usr/bin* path. That's the standard approach for Linux ever since. With interpreted languages like JavaScript or Python, it may become complicated. Since you're shipping multiple JavaScript or Python files, you may want to put them into the */opt/* path. An example of a daemon written in Node.js is my HTTP Time Travel Proxy.

Your server software would ideally be packaged into *'deb' (Debian, Ubuntu, Linux Mint and Linux, Elementary OS, Raspberry Pi OS)* or *'rpm' (Red Hat, Fedora, CentOS, SUSE)* packages for the different Linux distributions. The packages allow for an easy and quick installation on the target system. In addition you might also want to provide your server software as a docker image quickly allowing people to spin up a container to run the application.

Servers with Go compared to C++ and Node.js

Most of the well known server products were written in either C or C++. Popular examples are the Apache Webserver, Nginx, MySQL, Postgres, Memcache or Redis. The C source code of Memcache can be found in the Memcache repository as a reference.

The challenge of writing servers in C or C++

C and C++ have been the first choice for writing any system applications, especially servers. The main reason is performance and portability. The latter means that a server can also be compiled for a variety of different Linux distributions. The downside is development speed, complexity and a high error rate. Memory management in C and C++ can become extremely complex and you'll find yourself relying on a number of libraries to achieve reasonable developer ergonomics.

You shouldn't be writing servers with Node.js

As explained, it is absolutely possible to write server software in JavaScript or TypeScript for Node.js. There are however significant disadvantages. First of all, you will need to ship your entire source code to the target system. If your application is marketed through places like the AWS marketplace or any Application Service Providers (ASPs), you're running the risk of having your code exposed to your users. Your application also comes with Node.js as a significant and sizeable dependency.

Go is a good alternative to C and C++

Most server software does not need to extreme runtime performance that can be achieved with C or C++. Considering that systems like Kubernetes are written in Go, it may very well be the perfect choice for your server software. Be it a commercial or Open Source server software application. Go's concurrency, memory management and compiler allow creating a result as good as that or C++ with an acceptable compromise in runtime performance.

Interacting with the system in Go

You can easily "daemonize" any Go application without any further work on the code. However, your Go application should be able to handle the signals from the system.

- Reload the config when **SIGHUP** is received
- Shut down when **SIGINT** or **SIGTERM** is received

Users will control your server or daemon using the systemd tools available on their system. In generale the "service" command is used to control daemons or system services on Linux.

```
Copy # start the system service
sudo service myserver start

# stop the system service (SIGTERM signal)
sudo service myserver stop

# stop and start the system service (SIGTERM signal)
sudo service myserver restart

# reload the configuration (SIGHUP signal)
sudo service myserver reload
```

The below code shows a very basic Go application handling a **SIGHUP** system call (abbr. "syscall") to reload the configuration of the service.

```
Copy package main

import "os/signal"
import "os"
import "fmt"
import "syscall"
import "time"

func main() {
  c := make(chan os.Signal, 1)
  signal.Notify(c, syscall.SIGHUP)

  go func(){
    for sig := range c {
      fmt.Println("SIGHUP. Reloading config...")
    }
  }()

  for {
    time.Sleep(5000 * time.Millisecond)
  }
}
```

Once SIGHUP is received, the application shall reload its configuration file. The reason most likely is that the user changed the configuration file in */etc/myserver/* and wants the application to reload it.

Setting up the system service

Managing system services with systemd has become relatively simple. You need to carefully watch that your service has sufficient privileges while still following the least privilege paradigm for security. Other than that you only need a *.service* file in */etc/systemd/system/*.

```
Copy # Service file under
/etc/systemd/system/myserver.service
[Unit]
Description=My Server
After=network.target

[Service]
Type=simple
User=daemon
ExecStart=/usr/bin/myserver
Restart=on-failure

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

Once you've stored the service file in the path, you can run **"sudo systemctl daemon-reload"** to reload all daemon configurations and your service file. Afterwards you'll be able to start your service with **"sudo service myserver start"**. With these relatively small additions to your Go application, you successfully daemonized your application already.

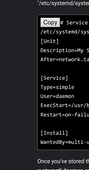
Compiling for different platforms

With the above sorted, you already have all the requirements to compile for the various different platforms such as x86_64 (e.g. Intel and AMD) and arm64 (e.g. Ampere, Graviton). The following commands

```
Copy # build for Linux with ARM64
env GOOS=linux GOARCH=arm64 go build -o /bin/arm64-
linux/myserver

# build for Linux with Intel/AMD x86_64/amd64
env GOOS=linux GOARCH=amd64 go build -o /bin/amd64-
linux/myserver
```

It is absolutely manageable to create your own build pipeline for the desired architectures. However, I can strongly recommend the openSUSE Build Service (abbr. "OBS") for automated build of your applications. I have used it myself (see my repository on Github that uses OBS). I also recommend the Beginner's Guide for OBS and the Packaging Go article in the help section. You do not need OBS for the start, but you may want to later switch from building "manually" to using OBS.



Have an awesome app for the Raspberry Pi? Just package and publish it for Raspberry Pi OS.

Packaging the server software for Linux

Once you're software is ready to be released, you may want to package it as a deb and rpm package. You may also want to release it as a Docker image. Regardless of whether you want to publish your server software as an Open Source project or a commercial solution, I highly recommend using GoReleaser. Creating packages and images can be cumbersome since every package manager uses its own formats and toolsets. GoReleaser simplifies the approach by allowing you to create all these packages in a single process.

A note on commercial licences

If your server software is intended to be a commercial product, you may want to allow adding a serial number or activation code to the config file. This is the usual way to freely distribute the binaries and then requiring the customer to add the licence information to a configuration file.

The return of server software

Writing server software was long the domain of C and C++. With the rise of Go and Rust, simpler and easier alternatives have made writing server software faster, easier and more economical. This also starts to mark the end of general purpose software solutions such as relational databases, search engines and document stores.

The last years have already seen the rise of new database systems, networking services, container services and many others. Kubernetes, written in Go, has proven that complex performance-related software can be written in Go efficiently.

Thank you for reading, Jan

Have you written server software in Go? What was your experience?