2025-07-01

Myriad Dreamin Blog 2025-06

Archive of Blog posts in June 2025.

Contents

2025-06-02T10:50:39+08:00

Hosting Multiple Websites using Caddy

To host multiple websites on a single server, I tried nginx, caddy, and traefik, and finally use caddy.

I bought a VPS to host my websites, a home page (i.myriad-dreamin.com) and a mirror site of my blog (cn.myriad-dreamin.com). Since Cloudflare is not available in my country, I'd better host them on my own server instead of proxying them through Cloudflare.

1.1 Directory Structure

The directory structure of the websites is as follows:

```
deployment
├── docker-compose.yml
├── caddy
│   ├── config
│   │   └── Caddyfile
│   ├── log
│   └── data
├── nginx
│   ├── conf
│   │   └── nginx.conf
│   └── log
├── dist
│   ├── i.myriad-dreamin.com
│   │   └── index.html
│   └── cn.myriad-dreamin.com
│       └── index.html
└── certbot
    ├── ssl
    └── www
```

The `docker-compose.yml` file contains all containers running for the websites The `dist` directory contains the static files for each website. The `caddy` or `nginx` have their owned directory to store the configuration files and logs. A `certbot` directory contains the SSL certificates and the webroot for certbot.

1.2 Serving `dist` through HTTP File Server

I don't want to use integrated file servers from `caddy` or `nginx`. I would like have some fine-grained control over the files. For example, I would like to cache fonts permanently. So I seek a simple HTTP file server implementation. As usual, I first tried to find one written in Rust, but failed.

I have to admit that Rust is not a good (or simple) choice to build web services. There are some heavy engine, but I don't want to use them. If I turn my eyes to lightweight ones, I find they are not well maintained or not feature complete. My last try was tiny-http, which deserves a look. It is almost great, but I'm still not satisfied with it.

If I'm going to build some network things, why not use Go? I had good memory of writing network tools and services in Go. It is an indisputable good start. I start it with less than 10 lines of code, and it works well:

```
package main

import (
  "log"
  "net/http"
  "os"
)

func main() {
  if len(os.Args) < 2 {
    log.Fatal("Usage: file-server <port> (:80)")
  }
  var port = os.Args[1]

  http.Handle("/", http.FileServer(http.Dir(".")))

  log.Println("Server listening on", port)
  log.Fatal(http.ListenAndServe(port, nil))
}
```

I also made some other improvments, like `gzip` compression:

```
// https://gist.github.com/bryfry/09a650eb8aac0fb76c24
import (
  "compress/gzip"
  "io"
  "strings"
)

type GzipResponseWriter struct {
  io.Writer
  http.ResponseWriter
}

func (w GzipResponseWriter) Write(b []byte) (int, error) {
  return w.Writer.Write(b)
}

func Gzip(handler http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    if !strings.Contains(r.Header.Get("Accept-Encoding"), "gzip") {
      handler.ServeHTTP(w, r)
      return
    }
    w.Header().Set("Content-Encoding", "gzip")
    gz := gzip.NewWriter(w)
    defer gz.Close()
    gzw := GzipResponseWriter{Writer: gz, ResponseWriter: w}
    handler.ServeHTTP(gzw, r)
  })
}
```

And change the main function to use the `Gzip` middleware:

```
 func main() {
   ...
-  http.Handle("/", http.FileServer(http.Dir(".")))
+  fs := http.FileServer(http.Dir("."))
+  http.Handle("/", Gzip(fs))
   ...
 }
```

Again, I only used standard libraries to build my custom tools. `gopls`, as one of my favorite language server, completed all of the package imports automatically.

1.3 HTTPS File Server?

About 4 years ago, I had experience to build a HTTPS file server using Go, but this is not a best practice in my view. Considering that I have to make an ingress controller, the SSL/TLS could be handled in middle. This mitigates both the complexity and attack surface of http services.

1.4 Building the HTTP File Server Container

It is not needed to build a custom image for the file server, if you use the following command to build the Go program:

```
CGO_ENABLED=0 go build -tags netgo -o target/file-server ./cmd/file-server
```

Simply start a `alpine` container with the file server binary mounted as a volume, and it will work well. The `docker-compose.yml` file is as follows:

```
services:
    homepage:
        container_name: homepage
        image: alpine:latest
        restart: unless-stopped
        environment:
            TZ : 'Asia/Shanghai'
        working_dir: /app
        volumes:
            - /usr/local/bin/file-server:/usr/local/bin/file-server:ro
            - ./dist/homepage/:/app/
        command: 'file-server :80'
```

1.5 Building Ingress using Nginx

I used both Caddy and Nginx. Both of them are good in my mind. Since it is not so disturbing to try both of them, I first tried Nginx, whose docker image is maintained by docker officially:

First, add a container for Nginx in `docker-compose.yml`:

```yaml
services:
  nginx:
      container_name: nginx
      image: nginx
      restart: unless-stopped
      ports:
          - "80:80"
          - "443:443"
      environment:
          TZ : 'Asia/Shanghai'
      volumes:
          - ./nginx/conf:/etc/nginx
          - ./nginx/web:/usr/share/nginx
          - ./nginx/log:/var/log/nginx
          - ./certbot/www:/usr/share/certbot/www:ro
          - ./certbot/ssl:/usr/share/certbot/ssl:ro
      command:  nginx -g 'daemon off;'
```

And add a configuration file `nginx.conf` in `nginx/conf` directory:

```
events {
    worker_connections  4096;
}
http {
    server {
        listen 80;
        listen [::]:80;

        server_name  orange.myriad-dreamin.com;
        server_tokens off;

        location /.well-known/acme-challenge/ {
            root /usr/share/certbot/www;
        }
        location / {
            return 301 https://orange.myriad-dreamin.com$request_uri;
        }
    }
}
```

Note that `location /.well-known/acme-challenge/` is intercepted for HTTP challenge from certbot, which is used to obtain SSL certificates. The `location /` block redirects all HTTP traffic to HTTPS.

Then, running `docker compose up -d nginx` to start the Nginx container. The Nginx will listen on port 80 and 443.

1.6 Making SSL Certificates using Certbot

Add a `certbot` container in `docker-compose.yml`:

```
services:
    certbot:
      container_name: certbot
      image: certbot/certbot
      volumes:
          - ./certbot/www:/usr/share/certbot/www:rw
          - ./certbot/ssl:/etc/letsencrypt:rw
```

Dry running the certbot to check if everything is fine:

```
docker compose run --rm  certbot certonly --webroot --webroot-path /usr/share/
certbot/www/ --dry-run -d orange.myriad-dreamin.com
```

And then remove the `--dry-run` flag to obtain the real certificates.

If everything is fine, the certificates will be stored in `certbot/ssl` directory.

1.7 Serving HTTPS using Nginx

The SSL certificates should be accessible in `/usr/share/certbot/ssl/live/orange.myriad-dreamin.com`. Let's add a server block in `nginx.conf` to serve the HTTPS traffic:

```
http {
    log_format main  '$remote_addr - $remote_user [$time_local] "$request" '
                     'status=$status body_bytes_sent=$body_bytes_sent
http_referer="$http_referer" '
                     'http_user_agent="$http_user_agent"
http_x_forwarded_for="$http_x_forwarded_for"';

    server {
        listen       443 ssl;
        listen [::]:443  ssl;
        server_name  orange.myriad-dreamin.com;

        access_log  /var/log/nginx/orange.myriad-dreamin.com.access.log  main;
        error_log  /var/log/nginx/orange.myriad-dreamin.com.error.log;

        ssl_certificate /usr/share/certbot/ssl/live/orange.myriad-dreamin.com/
fullchain.pem;
        ssl_certificate_key /usr/share/certbot/ssl/live/orange.myriad-dreamin.com/
privkey.pem;
        ssl_session_timeout 5m;
        ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:HIGH:!aNULL:!MD5:!RC4:!DHE;
        ssl_prefer_server_ciphers on;

        location / {
            proxy_pass http://homepage;
            proxy_set_header Host $http_host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header REMOTE-HOST $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
}
```

Since we use `docker compose`, The `http://homepage` is resolved by the Docker's internal DNS to the `homepage` container, which is running the HTTP file server we started earlier.

To support a new site, just copy the two server blocks (another one is in the previous section) about `orange.myriad-dreamin.com` and change the `server_name` to the new site name. I thing this is simple enough.

1.8 The Bad Guys are Accessing My Sites

From the logs, I found that there are some bad guys trying to access my site. They are trying to access many common paths, like `/admin`, `/login`, `/wp-login.php`, etc. That's interesting. Luckily, I only have read-only static files, and both Nginx and Golang HTTP file server are robust enough. But even if Nginx has been used for 20 years, we can usually see CVEs about it. Caddy does has slightly poorer performance, but my personal websites doesn't need to handle high traffic yet. `traefik` is another choice, but it is too complex and I might not use it for my personal websites. I think we can try Caddy next.

1.9 Serving HTTP using Caddy

First add a `caddy` container in `docker-compose.yml`:

```yaml
services:
    caddy:
        container_name: caddy
        image: caddy:latest
        restart: unless-stopped
        environment:
            TZ : 'Asia/Shanghai'
        ports:
        - "80:80"
        - "443:443"
        - "443:443/udp"
        volumes:
        - ./caddy/config:/etc/caddy
        - ./caddy/data:/data
        - ./caddy/log:/var/log/caddy
```

Then create a `Caddyfile` in `caddy/config` directory:

```
:80 {
  respond "Hello World!"
}
```

We should be able to get a response containing `"Hello World!"` from the Caddy server by running `docker compose up -d caddy` and visiting `http://localhost:80`.

1.10 Serving HTTPS using Caddy

Caddy can maintain the SSL certificates automatically, so we don't need to use `certbot` anymore. It will be pretty easy to set up a HTTPS server using Caddy. Just change the `Caddyfile` to:

```
orange.myriad-dreamin.com {
  tls x@email.com
  reverse_proxy homepage
}
```

Once again, `homepage` is the name of the HTTP file server container, which is resolved by Docker's internal DNS.

Execute the following command to ensure the configuration is hot reloaded:

```
docker compose exec caddy caddy reload --config /etc/caddy/Caddyfile
```

Looks even much simpler than Nginx, right? Besides, Caddy is written in Go, so no memory bug will be introduced.

1.11 Recording Access Logs

Caddy supports both Plaintext and JSON format for access logs. To enable access logs in Caddy, we can add the following snippet to the `Caddyfile`:

```
(subdomain-log) {
  log {
    hostnames {args[0]}
    format json
    output file /var/log/caddy/{args[0]}.jsonl {
      roll_size 100MiB
      roll_keep 3
      roll_keep_for 720h
    }
  }
}
```

And then include this snippet in each site block:

```
 orange.myriad-dreamin.com {
+  import subdomain-log orange.myriad-dreamin.com
   tls x@email.com
   reverse_proxy homepage
 }
```

I prefer JSON format, which is more structured and easier to parse. Among them, hl is a good tool to parse JSON logs.

```
$ hl caddy/log/orange.myriad-dreamin.com.jsonl
Jun 01 01:02:03.456 [INF] http.log.access.log0: handled request request.remote-
ip=a.b.c.d request.remote-port="xyz" request.client-ip=a.b.c.d ...
```

In fact, copilot helped me aggregate and display the access logs in a more readable way.

1.12 List of Code

`docker-compose.yml`:

```yaml
services:
    caddy:
        container_name: caddy
        image: caddy:latest
        restart: unless-stopped
        environment:
            TZ : 'Asia/Shanghai'
        ports:
        - "80:80"
        - "443:443"
        - "443:443/udp"
        volumes:
        - ./caddy/config:/etc/caddy
        - ./caddy/data:/data
        - ./caddy/log:/var/log/caddy
    homepage:
        container_name: homepage
        image: alpine:latest
        restart: unless-stopped
        environment:
            TZ : 'Asia/Shanghai'
        working_dir: /app
        volumes:
            - /usr/local/bin/file-server:/usr/local/bin/file-server:ro
            - ./dist/homepage/:/app/
        command: 'file-server :80'
```

caddy/config/Caddyfile:

```
(subdomain-log) {
  log {
    hostnames {args[0]}
    format json
    output file /var/log/caddy/{args[0]}.jsonl {
      roll_size 100MiB
      roll_keep 3
      roll_keep_for 720h
    }
  }
}

orange.myriad-dreamin.com {
  import subdomain-log orange.myriad-dreamin.com
  tls x@email.com
  reverse_proxy homepage
}
```

2025-06-17T11:11:54+08:00

Comment System

I built a simple comment system for my blog.

I would like to pick a suitable comment system for my blog. My considerations are:

1. It should have minimal backend requirements. If there is a backend, it should be reachable in global-wide.
2. It should be able to hide personal information like email address. I know that many email addresses are not a secret, but I don't want to give a change to reveal it from my blog.
3. It should be easy to use so that people will not stop commenting because of the complexity.
4. It should use JavaScript in frontend as little as possible.

This comment system supports:

- Markdown syntax and mathematical formulas.
- User mentions and comment replies.
- Email notifications.

2.1 Email the Old Fashion

I first investigate the mailto protocol. That is an actual old falsionm but I suspect its availablility. People rarely click `mailto:` (imo) and the remaining usage is leaking the email address. It is a simple and doesn't require any JavaScript and backend. But it has some problems:

- When the user clicks the link, it will open the user's email client, while people usually doesn't configure their email client, so they will go to the Outlook or Thunderbird and might exit it quickly. In worst case, they will not comment to my blog anymore. This breaks Point #3.
- The mailto link will reveal my email address. It is not a big problem, but it breaks Point #2.

2.2 Abusing GitHub

No backend is a lie. It just appears in another way. Another most popular comment system is utilizing GitHub issues. It is a good idea, but it also has some problems:

- It requires the user to have a GitHub account, which is not always the case. This breaks Point #3.
- The GitHub is not available in some countries. This breaks Point #1.
- No, I didn't consider Point #4, which is only a bonus, but such comment system usually requires JavaScript to render and process the comments.

2.3 My Home-made Comment System

Since we have served the static files in Golang's HTTP server, what about deploying a simple comment system on the same server? People who can access the frontend resources should be able to access the same server. This should have some downsides, but can be a easy start.

Should I use any backend framework? I bet this is not necessary at least we are not aiming to make a blog sites that handlers 100k of comment requests per second.

In go, this is easy to start:

```
package main

import (
  ...

  "database/sql"
  _ "github.com/mattn/go-sqlite3"
)

type Handler struct {
  db *sql.DB
}

func (h *Handler) makeTables() {
  h.db.Exec("CREATE TABLE IF NOT EXISTS comments (id INTEGER PRIMARY KEY
AUTOINCREMENT, article_id TEXT, email TEXT, content TEXT, authorized BOOLEAN NOT
NULL DEFAULT FALSE, created_at INTEGER)")
}

func (h *Handler) handleCommentPost(w http.ResponseWriter, r *http.Request) {
  articleId, content, email, createdAt := r.FormValue("article_id"),
r.FormValue("content"), r.FormValue("email"), time.Now().UnixMilli()
  _, err := mail.ParseAddress(email)

  // Validate the input
  if err != nil || len(content) > 4096 || len(email) > 128 || !
h.mustExistsArticle(articleId, w) {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }

  // Inserts comment into database
  _, err = h.db.Exec("INSERT INTO comments (article_id, content, email, authorized,
created_at) VALUES (?, ?, ?, ?, ?)", articleId, content, email, false, createdAt)
  if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }

  // Respond with success
}
```

Now, we can get comments periodically from the backend and render them in the static-site blog.

2.4 Comentario

I also surveyed some self-hosted comment systems, like Comentario. What I don't understand is what it is saying in Requirements: Sqlite:

- It's not scalable: it will probably be okay for up to a few thousand comments, but beyond that the performance will degrade.

That said, it's probably fine to use SQLite as a minimal option to try out Comentario, or even to use it for your (low traffic) personal blog.

I'm not offensive and respect that comentario has beautiful and out look and is totally free. Either my experience is not enough that I didn't ever handle a blog with thousands of comments, or comentario is too heavy to use Sqlite as a backend.

2.5 Continuing developing my Comment System

This doesn't mean that I will use my home-made comment system eventually. I continued to develop it a bit to be able to reply to my friends.

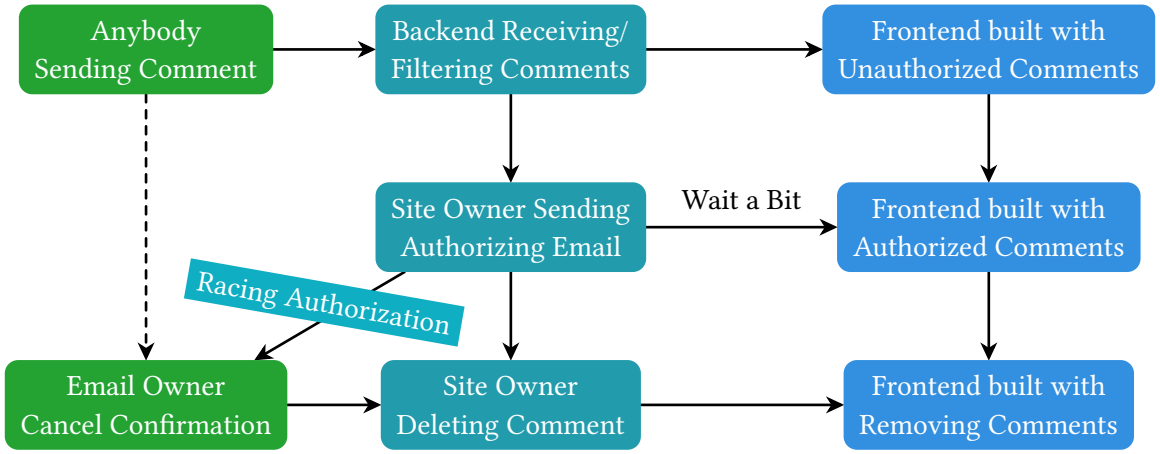

Figure 1: A simple comment system with minimal backend requirements.

2.5.1 Sending and Rendering Comment without Authorization

This minimalize the steps to send a comment. Registering or oauth is not required.

The backend only sends an email to the owner, so it can be easily deployed on cloud and distributed computing services like cloudflare workers.

- No, I still use the golang backend in the previous step because it works, but it can be easily ported the cloudflare workers. I may use the cloudflare workers in the future when I find it doesn't work perfectly in future.

2.5.2 Authorization Steps

The email can be confirmed by the owner of the email address in a racing manner:

1. I will send an email to notify the email owner.
2. The email owner doesn't have to send back a confirmation email. I will wait a bit to remove the `[Unauthorized]` tag from the comment.
   - It will be a day if the email owner doesn't continue comment on the blog site.
   - Otherwise, when we observe the activity of the email owner, we can remove the tag immediately.

2.6 Custom Markup made with Typst

The comment is in markdown format with extended syntax. It is rendered by Typst's `cmark` package, so it can be easily customized. Two custom syntax are extended.

2.6.1 `[user:name]`

People can mention by the name other people that "have been occurred in the current blog site". It is rendered as a hash link so no JavaScript is required to handle it.

- I don't know if there will be two "Steven" commenting on my blog, but I can think of it when it really happens.

2.6.2 `[comment:id]`

People can reply to a comment in the same article by its id. And it is rendered as a hash link along with the first line of content of the comment.

People who have been mentioned in the comment will receive an email notification.

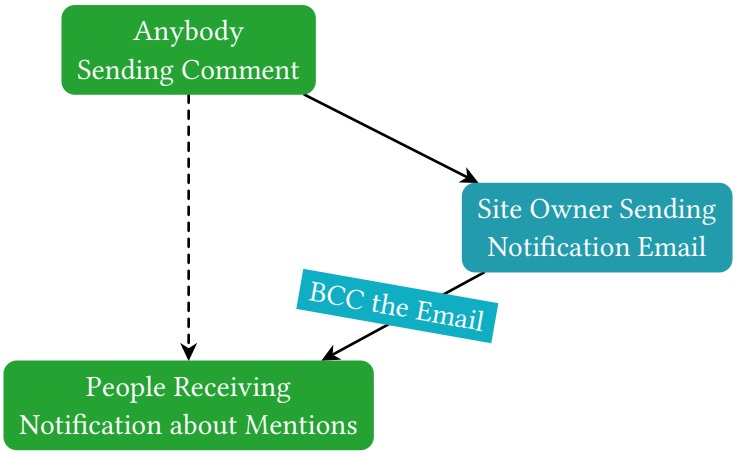

Figure 2: The notification of mentioned people.

The "BCC the Email" means that people won't see the email address of other people who have been mentioned in the comment, so it won't violate Point #2.

2.7 Post Story: gravatar

Should I use gravatar to show the avatar of the commenter? It is a good idea, but I didn't do it because it doesn't really protect the email address of the commenter.

We know that gravatar hashes the email address to avoid the exhibition the plain text email address occurring on the gravatar URL. However,

- Anybody can collect a list of email addresses and their gravatar hashes to find the real email address.
- Even if somebody doesn't try to reveal the email address, they can collect all the occurrences of the gravatar URLs to infer the activities of the commenter.

It is not that important to really protect the email address, but it is better to achieve that because we don't know whether anybody cares about that.