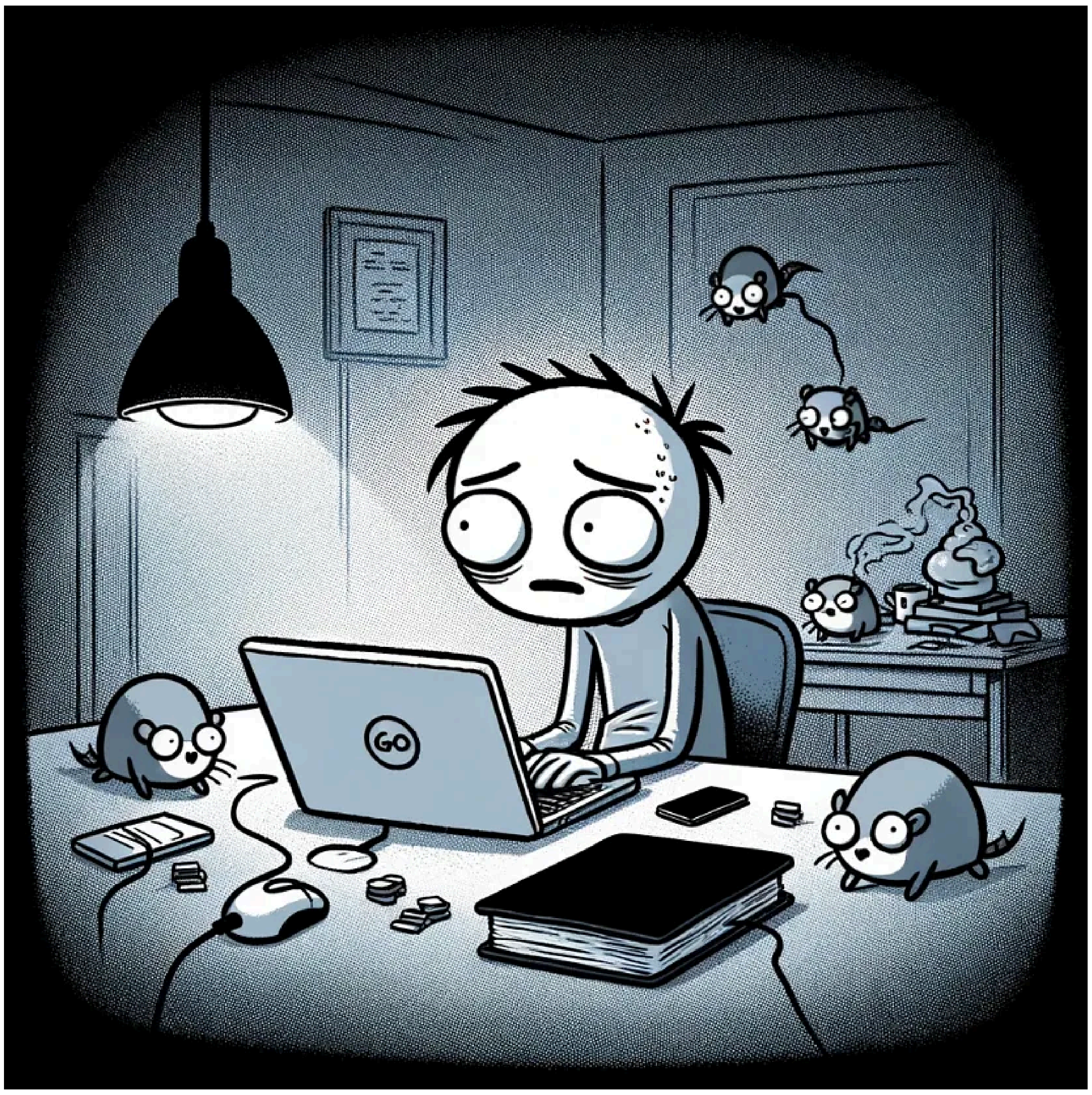
My Golang Experience: Managing Multiple Private Repositories with GitHub and Docker

Jan Seidl · Follow
8 min read · May 22, 2024



ChatGPT plus: a tired software engineer, hunched over a laptop in a dimly lit room, drawn in the style of XKCD comics, monochrome with simple line art, featuring an exhausted expression, cluttered desk, and flying small gophers (golang logo) around their head, adding a touch of humor and wit, artistically created to capture the essence of a late-night coding session

Working with Go, GitHub, and multiple private repositories can be challenging, especially when it comes to managing SSH keys and ensuring secure, smooth deployments. In this blog post, I'll share my journey, including the obstacles I've encountered and the solutions I've found effective. We'll cover GitHub Actions, Docker, and Makefiles to create a seamless workflow.

Setting Up GOPRIVATE

To manage multiple private repositories in Go, the first step is to configure *GOPRIVATE*. This is straightforward:

```
export GOPRIVATE=github.com/project/repo-a,github.com/project/repo-b
```

or it's possible either

```
export GOPRIVATE=github.com/project/*
```

when everything is in one project.

This tells the Go toolchain to treat these repositories as private.

Managing SSH Keys for Multiple Repositories

One of the main obstacles I faced was dealing with SSH keys for multiple repositories. GitHub only allows one deployment key per repository, which makes it difficult to work with multiple private repositories.

> *You can launch projects from a repository on GitHub.com to your server by using a deploy key, which is an SSH key that grants access to a single repository.*
>
> *https://docs.github.com/en/authentication/connecting-to-github-with-ssh/managing-deploy-keys#deploy-keys*

To be fair, there is a workaround described. You need multiple deployment keys. Which conflicts with ssh-agent. Ouch.

A great help here was the command:

```
ssh -T git@github.com
# Hi JaSei! You've successfully authenticated, but GitHub does not provide shell
```

It depends on which key you are using, you can see your account after *Hi*. In the case of the deploy key, it is the name of the repo.

But with combination of ssh-agent, which (probably?) give you first "match" key, you are lost. For *repo-a* and *repo-b* it will be the first key, but the first key will only match one of those repos.

Here's a really great github gist that describes how to do this: https://gist.github.com/jrapoport/d12f60029eef017354d0ec982b918258 so I won't duplicate that and let's jump to the example.

Initial solution using GitHub actions

A solution that works well within GitHub Actions. It involves setting up SSH keys and configuring SSH for multiple repositories.

```
name: service
on:
 push:

concurrency:
 group: ${{ github.workflow }}-${{ github.ref }}
 cancel-in-progress: true

env:
 GO_VERSION: "1.22"
 GOPRIVATE: "github.com/project/repo-a,github.com/project/repo-b"

jobs:
 build_and_test:
 runs-on: ubuntu-latest

steps:
 - name: Checkout repository
   uses: actions/checkout@v4
 - name: Setup SSH
   run: |
     mkdir -p ~/.ssh
     ssh-keyscan github.com >> ~/.ssh/known_hosts
     echo "${{ secrets.REPO_A_PRIVATE_KEY }}" | tr -d '\r' > ~/.ssh/repo_a
     chmod 600 ~/.ssh/repo_a
     echo "${{ secrets.REPO_B_PRIVATE_KEY }}" | tr -d '\r' > ~/.ssh/repo_b
     chmod 600 ~/.ssh/repo_b
     cat <<EOF > ~/.ssh/config
     Host github.com-repo-a
       HostName github.com
       User git
       IdentityFile ~/.ssh/repo_a
       IdentitiesOnly yes
     Host github.com-repo-b
       HostName github.com
       User git
       IdentityFile ~/.ssh/repo_b
       IdentitiesOnly yes
     EOF

 - name: Add SSH Keys to SSH Agent
   run: |
     eval "$(ssh-agent -s)"
     ssh-add ~/.ssh/repo_a
     ssh-add ~/.ssh/repo_b
     echo "SSH_AUTH_SOCK=$SSH_AUTH_SOCK" >> $GITHUB_ENV

 - name: Setup access for private go modules
   run: |
     git config – global url."git@github.com-repo-a:project/repo-a".insteadOf "h
     git config – global url."git@github.com-repo-b:project/repo-b".insteadOf "h

 - name: Test GitHub SSH connection
   run: |
     if ssh -T git@github.com-repo-a 2>&1 | grep -q "repo-a"; then
       echo "SSH connection to GitHub successful"
     else
       echo "SSH connection to GitHub failed"
     exit 1
     fi
     if ssh -T git@github.com-repo-b 2>&1 | grep -q "repo-b"; then
```

```
        echo "SSH connection to GitHub successful"
      else
        echo "SSH connection to GitHub failed"
        exit 1
      fi

  - name: Test
    run: go test ./…
```

This setup works well within GitHub Actions. Even though it's complicated and not universal, I accept that.

Unfortunately, we use Docker (probably like everyone else right now) and you need to replicate the same logic for Docker. Let's dive into handling multiple SSH keys within a Docker build.

Docker: Handling Multiple SSH Keys

When building Docker images, managing SSH keys securely and effectively is critical and a multi-step Docker build is a must. Below is a Docker file that handles multiple private repositories:

```
FROM golang:1.22-bookworm AS base

ARG REPO_A_PRIVATE_KEY
ARG REPO_B_PRIVATE_KEY

COPY go.mod go.sum /src/

WORKDIR /src

ENV GOPRIVATE=github.com/project/repo-a,github.com/project/repo-b

RUN mkdir -m 0600 ~/.ssh && \
 ssh-keyscan github.com >> ~/.ssh/known_hosts && \
 if [ -n "$REPO_A_PRIVATE_KEY" ] && [ -n "$REPO_B_PRIVATE_KEY" ]; then \
 echo "$REPO_A_PRIVATE_KEY" > ~/.ssh/repo_a && \
 chmod 600 ~/.ssh/repo_a && \
 echo "$REPO_B_PRIVATE_KEY" > ~/.ssh/repo_b && \
 chmod 600 ~/.ssh/repo_b && \
 printf "Host github.com-repo-a\n User git\n HostName github.com\n IdentityFile
 printf "Host github.com-repo_b\n User git\n HostName github.com\n IdentityFile
 fi

RUN if [ -n "$REPO_A_PRIVATE_KEY" ] && [ -n "$REPO_B_PRIVATE_KEY" ]; then \
 git config - global - add url."git@github.com-repo-a:project/repo-a".insteadOf
 git config - global - add url."git@github.com-repo-b:project/repo-b".insteadOf
 fi && \
 git config - global - add url."git@github.com:".insteadOf "https://github.com/"

RUN if [ -n "$REPO_A_PRIVATE_KEY" ] && [ -n "$REPO_B_PRIVATE_KEY" ]; then \
 (ssh -T git@github.com-repo-a 2>&1 | grep -q "repo-a") && \
 (ssh -T git@github.com-repo-b 2>&1 | grep -q "repo-b"); \
 fi

RUN -mount=type=ssh go mod download -x && go mod verify

COPY . /src

RUN -mount=type=ssh go build -o /src/build/service

# only if you have to tests from docker (e.g. if you don't use testcontainers)
WORKDIR /src/tests
```

```
RUN go test -c -o /src/build/service_tests


FROM debian:bookworm-slim

# Add the executable into the image.
COPY - from=base /src/build/service /
# temporary solution: add tests to the image (we can remove with testcontainers)
COPY - from=base /src/build/service_tests /
# In case the service contains an HTTP server,
# we want the service to listen on all interfaces.
ENV HTTP_LISTEN_HOST 0.0.0.0
ENTRYPOINT ["/service"]
HEALTHCHECK CMD ["/service", "healthcheck"]
```

Nothing strange here, right? It's almost the same as what we've seen in the github action workflow. Although *-mount=type=ssh* and *if*s there would make question. The good thing about this setup is that it's the same dockerfile for deploy keys and either for user ssh agent provided key.

. . .

To build the docker image, you need to pass the SSH keys as build arguments. Here's how you can do it:

```
export REPO_A_PRIVATE_KEY=$(cat ~/.ssh/repo_a)
export REPO_B_PRIVATE_KEY=$(cat ~/.ssh/repo_b)
docker build \
 - ssh default \
 - build-arg REPO_A_PRIVATE_KEY=$REPO_A_PRIVATE_KEY \
 - build-arg REPO_B_PRIVATE_KEY=$REPO_B_PRIVATE_KEY \
 -t service:latest \
 .
```

This method ensures that the keys are handled securely and are only used during the build process. And as I said, it works for CI/CD (you set the `*_PRIVATE_KEY` envs) as well as for your private key in the ssh agent on local.

Nice. But what if you use Makefile like we unfortunately do.

Using Makefile for Docker Builds

If your project uses Makefiles, here's an snippet to integrate the Docker build process:

```
docker/build: docker/base
  @docker build \
    -ssh default \
    ... \
    \
    -tag "$(DOCKER_IMAGE):$(VERSION)" \
    -tag "$(DOCKER_IMAGE):latest" \
    \
    -build-arg REPO_A_PRIVATE_KEY="$$REPO_A_PRIVATE_KEY" \
    -build-arg REPO_B_PRIVATE_KEY="$$REPO_B_PRIVATE_KEY" \
    .
```

That's easy, right? But what if you have shared Makefile for all service and some needs different repos — unfortunately we do.

My idea was to have a special env variable that contains "references" to the environment variables.

```
export REPO_A_PRIVATE_KEY=$(cat ~/.ssh/repo_a)
export REPO_B_PRIVATE_KEY=$(cat ~/.ssh/repo_b)
BUILD_ARGS="REPO_A_PRIVATE_KEY REPO_B_PRIVATE_KEY" make docker/build.
```

Isn't that elegant? I thought it was, but I wasn't able to pass multiline private keys through makefile/shell — you can see the appendix at the end where my attempts are described.

And finally, I'm convinced that it's better and more reliable to use the "machine user" account described here:

> *If your server needs to access multiple repositories, you can create a new account on GitHub.com and attach an SSH key that will be used exclusively for automation.*
>
> *https://docs.github.com/en/authentication/connecting-to-github-with-ssh/managing-deploy-keys#machine-users.*

You have to pay +1 seat, but all this described bullshit is gone and everything is solved by ssh-agent.

Simplifying with a Machine User

Dockerfile with Machine User

Here's a simplified Dockerfile using a machine user:

```
FROM golang:1.22-bookworm AS base

COPY go.mod go.sum /src/

WORKDIR /src

ENV GOPRIVATE=github.com/project/*

RUN mkdir -m 0600 ~/.ssh && \
 ssh-keyscan gitlab.com >> ~/.ssh/known_hosts && \
 ssh-keyscan github.com >> ~/.ssh/known_hosts

RUN git config -global -addurl."git@github.com:".insteadOf "https://github.com/"

RUN -mount=type=ssh \
 -mount=type=bind,source=go.sum,target=go.sum \
 -mount=type=bind,source=go.mod,target=go.mod \
 go mod download -x && go mod verify

COPY . /src

RUN go build -o build/service

# temporary solution: add tests to the image (we can remove with testcontainers)
WORKDIR /src/tests
RUN go test -c -o /src/build/service_tests


FROM debian:bookworm-slim
# Add the executable into the image.
COPY -from=base /src/build/service /
# temporary solution: add tests to the image (we can remove with testcontainers)
COPY -from=base /src/build/service_tests /
# In case the service contains an HTTP server,
# we want the service to listen on all interfaces.
ENV HTTP_LISTEN_HOST 0.0.0.0
ENTRYPOINT ["/service"]
HEALTHCHECK CMD ["/service", "healthcheck"]
```

GitHub Actions with Machine User

Here’s how you can set up GitHub Actions using a machine user:

```
name: service
on:
 push:

concurrency:
 group: ${{ github.workflow }}-${{ github.ref }}
 cancel-in-progress: true

env:
 GO_VERSION: "1.22"
 GOPRIVATE: "github.com/project/*"

jobs:
 build_and_test:
 runs-on: ubuntu-latest

steps:
 - uses: actions/checkout@v4

 - name: Add SSH Go Module Private Key
   env:
     SSH_AUTH_SOCK: /tmp/ssh_agent.sock
   run: |
     mkdir -p ~/.ssh
     ssh-keyscan github.com >> ~/.ssh/known_hosts
     ssh-agent -a $SSH_AUTH_SOCK > /dev/null
     ssh-add - <<< "${{ secrets.MACHINE_USER_PRIVATE_KEY }}"
     echo "SSH_AUTH_SOCK=$SSH_AUTH_SOCK" >> $GITHUB_ENV

 - name: Setup access for private go modules
   run: |
     git config - global url."ssh://git@github.com/".insteadOf https://github.co

- name: Set up Go
  uses: actions/setup-go@v4
  with:
   go-version: "${{ env.GO_VERSION }}"

- name: Test
  run: go test ./…

- name: Docker build
  run: make docker/build
```

Conclusion

Managing multiple private repositories with Go, GitHub, and Docker can be challenging, but by using GitHub actions, Docker build stages, and potentially a machine user account, you can do it.

I still think there's a way with deploy keys and relatively solid Makefile + shell magic, but I don't want to spend more time on it. And maybe the Docker/Makefile/GithubAction complexity will never pay off in comparison to the price of machine user account simplicity.

I hope this story helps you avoid spending hours in the same hell I did.

Appendix — the Makefile troubles

I've tried everything I could find on Google or LLMit (ChatGPT 4, 4o, Groq lama3–70b-8192, Gemini,…). And at the end I've made a good (I think) minimal example that just tries to propagate environment variables as I described above.

. . .

Dockerfile is not important, but to have something, let's say:

```
FROM debian:bookworm-slim

ARG ENV1
ARG ENV2

RUN echo "$ENV1" > env1
RUN echo "$ENV2" > env2
```

The goal have in files env1 and end2 in docker the same content as in $ENV1 and $ENV2, right?

. . .

I've made a shell script, that works with charm.

```
#!/bin/sh
set -x
echo "Running tests"

ARGS=()
for var in $ENVS; do
    if [ -z "${!var}" ]; then
        echo "Warning: Environment variable $var is not set" >&2
    else
        value=${!var}
        ARGS+=("--build-arg" "$var=$value")
    fi
done

echo "ENVS: $ENVS"
echo "Build arguments: ${ARGS[@]}"

docker build "${ARGS[@]}" -t test .
```

```
ENVS="ENV1 ENV2" ENV1=$(printf "Hello\nworld\nI'm\ncomming") ENV2="Some other te
```

And the equivalent Makefile is not equivalent, because it doesn’t work.

```
.PHONY: test
ENVS ?= ""

test:
 @echo "Building docker image"; \
    ARGS=(); \
 set -x; \
    for var in $(ENVS); do \
        if [ -z "$${!var}" ]; then \
            echo "Warning: Environment variable $$var is not set" >&2; \
        else \
   value=$${!var}; \
            ARGS+=("--build-arg" "$$var=$$value"); \
  fi; \
    done; \
    echo "ENVS: $(ENVS)"; \
    echo "Build arguments: $${ARGS[@]}"; \
    docker build $${ARGS[@]} -t test .
```

```
ENVS="ENV1 ENV2" ENV1=$(printf "Hello\nworld\nI'm\ncomming") ENV2="Some other te
```

I’ve tried all possible variants, encodings, etc. and nothing. Maybe I should go back to the base64 variant, but that also caused some problems. If you have an idea, don’t hesitate to share it.

Golang Docker Github Actions Experience

Written by Jan Seidl

4 Followers · 21 Following

Follow

No responses yet

What are your thoughts?

Respond

Recommended from Medium

Yash

Go 1.24: The Ultimate Update That Makes Development Faster and…

Go 1.24 (scheduled for release in February 2025) brings a variety of powerful new…

Jan 12 · 197

In Level Up Coding by Jacob Bennett

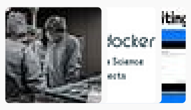
The 5 paid subscriptions I actually use in 2025 as a Staff Software…

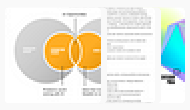
Tools I use that are cheaper than Netflix

Jan 7 · 5.7K · 121

Lists

General Coding Knowledge
20 stories · 1879 saves


Coding & Development
11 stories · 981 saves


Business
41 stories · 172 saves

Natural Language Processing
1889 stories · 1548 saves

In The Ordinary Programmer by huizhou92

Error Handling in Go: The New ? operator

It's Not A Silver Bullet.

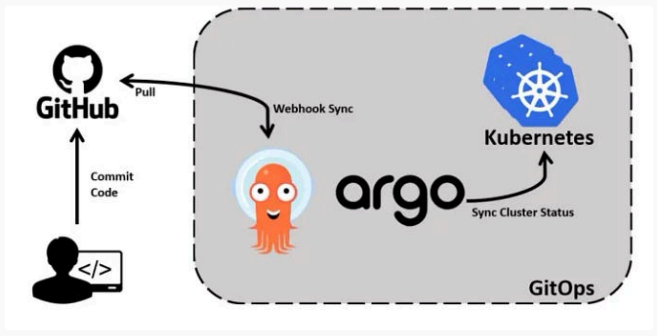
Jan 11 123 16

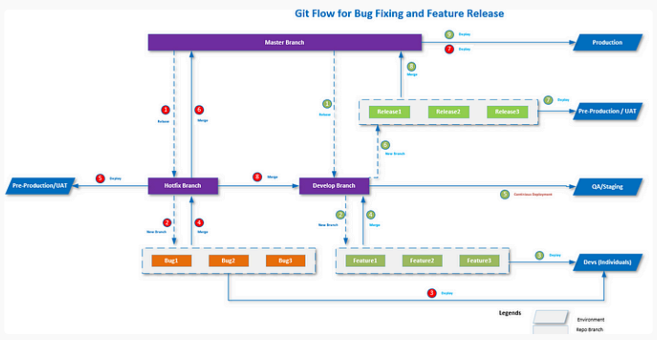
In Stackademic by Dylan Cooper

Zig's New Masterpiece: HashiCorp Co-founder's Latest Creation Tak...

Recently, a terminal emulator written in Zig reached its official 1.0 release, making wave...

Jan 13 750 10





Vijay Mehrotra

End-to-End DevOps on a Golang Web Application: From Code to...

Introduction

Aug 13, 2024 27

Bibhu Mishra

Git 3-Branch Strategy

A Git branching strategy is crucial for managing development, bug fixes, and...

Sep 14, 2024 40 1

See more recommendations

Help Status About Careers Press Blog Privacy Terms Text to speech Teams