Fleet Automation, The Hard Way

Why take the easy route?

John Doak and David Justice, Gophercon 2023

Because

The easy way can be way harder

What’s in this section?

- Using os/exec for automating local changes
- Using SSH for automating remote changes
- Writing a system agent to do local work at scale
- (Bonus) Designing safe, concurrent change automations

Let’s talk about local automation

Your new best friend

os.Exec

- Provides access to command execution in your local OS
- Not as friendly as the command line, unless you also invoke a shell
- You must be careful to either invoke trusted code or make sure to use a more secure version: https://golang.org/x/sys/execabs
- Can be used with other techniques to automate at scale

A few gotcha’s to look for

Because it can’t be that easy

- Output from a command line program is not always coming from where it seems:
  - Stdout for some
  - Stderr for others
  - You have to pay attention to capture the right thing
- Programs that you don’t just pass bytes around usually have to have output interpreted
  - Try to use programs with —json like flags to give digestible output
  - Otherwise, you have to parse the output (yuck)
- Program errors are not standardized
  - A program might exit with status 0, but still have an error
    - So checking the error returned by a Cmd is not always good enough
- Security must be paid attention to
  - What type of environment is this running in?
  - Are the programs trusted?
  - Do you need to cleanse the user input?

Validating program availability

Or how to make sure your program doesn’t crash

```go
const (
    kubectl = 0
    git     = 1
)

// program is a program name and path. path is setup by exec.Lookup().
type program struct {
    name string
    path string
}

var requiredList = []program{{name: "kubectl"}, {name: "git"}}

func programValidation() error {
    var err error
    for i, p := range requiredList {
        p.path, err = exec.LookPath(p.name)
        if err != nil {
            return fmt.Errorf("cannot find %s in our PATH", p)
        }
        requiredList[i] = p
    }
    return nil
}
```

Sometimes you might want to check the version installed

Could be changed to install missing programs or could embed them with “embed"

Bonus content: How to support Universal binaries with shell script

Introducing Command() & CommandContext()

For local forking

Command()

```
// kubectl has a familiar syntax because it is written with Cobra!
cmd := exec.Command(requiredList[kubectl].path, "apply", "-f", *config)
```

- Takes the program name (aka path) and arguments to run
- Returns a *Cmd object that can be used to execute the program
- You can gain access to the input and output streams of the program
- Can signal the program via cmd.Process.Signal() want to be more gentle and allow the program to react to a signal:
  - cmd.Process.Signal()

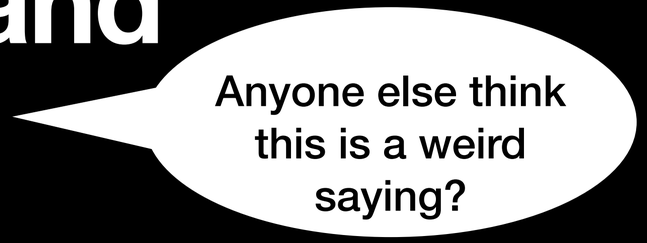
CommandContext()

```
// kubectl has a familiar syntax because it is written with Cobra!
cmd := exec.CommandContext(ctx, requiredList[kubectl].path, "apply", "-f", *config)
```

- The same as Command, but takes a Context
- The Context can be used to interrupt the program via os.Process.Kill()

Running the command

Many ways to skin a cat?



- Cmd.CombinedOutput()
  - Returns the output of both stderr and stdout
  - Blocks until completed
- Cmd.Output()
  - Runs the command and returns stdout
  - Blocks until completed
- Cmd.Run()
  - Runs the program and returns an error only
  - Blocks until completed
- Cmd.Start()
  - Runs the program and returns nothing
  - DOES NOT block
  - Use when you need to stream output from an application

What if I need to respond on stdin to stdout

Because sometimes it is not just some flags

- For example, this can happen with interactive command prompts
- Use either:
  - Google’s goexpect*: github.com/google/goexpect
  - Netflix goexpect* (my preferred choice): github.com/Netflix/go-expect

*These continue the proud tradition of TCL expect which has automated interactive tasks for network engineers forever

Simple example

Automate the ping command

```go
// hostalive returns true if the host is alive. You require special privileges to send
// ICMP packets on most devices. This bypasses that requirement by using the ping command.
func hostAlive(ctx context.Context, host netip.Addr) bool {
    // -c 1: send 1 ping packet
    // -t 2: timeout after 2 seconds
    // Note: to work on all platforms (like windows), you might need to use
    // build tags around this function or use os detection via runtime.GOOS.
    cmd := exec.CommandContext(ctx, ping, "-c", "1", "-t", "2", host.String())

    if err := cmd.Run(); err != nil {
        return false
    }
    return true
}
```

An error will mean that the ping failed, as we already evaluated the address

Full example: github.com/gofordevopsclass/automation_the_hard_way/ping

A CLI App using the ping command

Using os.Args

```
ElephantInTheRoom:ping jdoak$ go run . 127.0.0.1 www.news.com www.google.com www.microsoft.com 10.0.0.85
host(127.0.0.1) is alive: true
host(www.google.com) is alive: true
host(www.news.com) is alive: true
host(www.microsoft.com) is alive: true
host(10.0.0.85) is alive: false
```

Using stdin

```
ElephantInTheRoom:ping jdoak$ go run .
127.0.0.1
host(127.0.0.1) is alive: true
www.news.com
host(www.news.com) is alive: true
www.google.com
host(www.google.com) is alive: true
^Csignal: interrupt
```

Remotely run uname -a on machines

Objectives

- Get a list of CIDR addresses
- Ping all addresses in the range
- If they are alive connect to them via the “ssh” command
  - Note: we will talk about using SSH directly in code later on
- If SSH succeeds, run uname -a on the remote box to get kernel information

Running uname over SSH

```go
// runUname will attempt to use the "ssh" binary to log into a host and run "uname -a".
// This will return the output of that command.
func runUname(ctx context.Context, host net.IP, user string) (string, error) {
    if _, ok := ctx.Deadline(); !ok {
        var cancel context.CancelFunc
        ctx, cancel = context.WithTimeout(ctx, 5*time.Second)
        defer cancel()
    }

    login := fmt.Sprintf("%s@%s", user, host)
    cmd := exec.CommandContext(
        ctx,
        requiredList[ssh].path,
        "-o StrictHostKeyChecking=no",
        "-o BatchMode=yes",
        login,
        "uname -a",
    )
    out, err := cmd.CombinedOutput()
    if err != nil {
        return "", err
    }
    return string(out), nil
}
```

Prevents needing the host key and if the key doesn't work, don't fallback to password

SSH can send a single command

Full code at: gofordevopsclass/automation_the_hard_way/remoteUname

Example output

Scans the
127.0.0.0/24 network

```
ElephantInTheRoom:remoteUname jdoak$ go run . 127.0.0.0/24
{"Uname":"Darwin ElephantInTheRoom.local 22.6.0 Darwin Kernel Version 22.6.0: Wed Jul  5 22:22:05 PDT 2023; root:xnu-8796.141.3~6/RELEASE_ARM64_T60
00 arm64\n","Host":"127.0.0.1","Reachable":true,"LoginSSH":true}
{"Uname":"","Host":"127.0.0.2","Reachable":false,"LoginSSH":false}
{"Uname":"","Host":"127.0.0.8","Reachable":false,"LoginSSH":false}
```

Note on exec use

- Use Go packages over exec'ing binaries, if available
  - The code is more portable between systems
  - If you can't, think about wrapping command line capabilities in re-usable packages
- Don't exec user supplied input
  - Its the same as taking SQL input from a user, good way to find a dead or hacked system

Exercise

- Run the “top” command for 1 second on your machine
- Get the output and return it

Code at: github.com/johnsiilver/gofordevopsclass/tree/main/automation_the_hard_way/top

Let’s talk about remote automation

The ssh package

Your remote connection friend

- Don’t think of SSH as a terminal connection
  - Think of it as a way of connecting software securely
- With SSH, you can do things like:
  - Securely connect to a gRPC service without TLS
  - Connect to HTTP or HTTPS services
  - Connect to X11 services
  - Write your own remote terminal, similar to network devices
  - And yes, a standard terminal connection

SSH authorization methods

Gotta have choices

- Username/password
- Public key
- Challenge-response authentication

Password and public key authentication

```go
auth := ssh.Password("password")
```

```go
func publicKey(privateKeyFile string) (ssh.AuthMethod, error) {
    k, err := os.ReadFile(privateKeyFile)
    if err != nil {
        return nil, err
    }
    signer, err := ssh.ParsePrivateKey(k)
    if err != nil {
        return nil, err
    }
    return ssh.PublicKeys(signer), nil
}
```

Making the SSH connection

- We can assign the auth method we created to an SSH config
- You can have multiple auth methods to attempt
- We have a dial timeout of 5 seconds
- We are ignoring the host key for simplicity here

```
config := &ssh.ClientConfig{
    User:            user,
    Auth:            []ssh.AuthMethod{auth},
    HostKeyCallback: ssh.InsecureIgnoreHostKey(),
    Timeout:         5 * time.Second,
}
```

```
conn, err := ssh.Dial("tcp", host, config)
if err != nil {
    fmt.Println("Error: could not dial host: ", err)
    os.Exit(1)
}
defer conn.Close()
```

Sending a command

- You need a session per command you send
- Returns the stdout and stderr from the command run
- Avoids []byte to string conversion cost

```go
func combinedOutput(conn *ssh.Client, cmd string) (string, error) {
    sess, err := conn.NewSession()
    if err != nil {
        return "", err
    }
    defer sess.Close()

    b, err := sess.Output(cmd)
    if err != nil {
        return "", err
    }
    var s string
    if len(b) > 0 {
        s = unsafe.String(&b[0], len(b))
    }
    return s, nil
}
```

Full code at: gofordevopsclass/automation_the_hard_way/remoteCmd

Caution - Ignoring Host Keys

- Ignoring a host key with ssh.InsecureIgnoreHostKey() is not secure.
  - A typo in the host can send information to a system outside your control.
- Any date entered into a terminal or data sent could be captured by a third party.
- When working in a production environment, it is critical not to ignore the host key and store a valid list of host keys that can be checked.

Caution!!!
Leaky passwords

It is not safe to retrieve a password using flags or arguments.

- It is possible they can be seen in process managers, over someone's shoulder, etc…

Safe ways to get a password:

- Reading it from a secure file
- Using the terminal package to prevent terminal echo
  - Package can be found at: golang.org/x/term

```
fmt.Printf("Enter password: ")
b, err := term.ReadPassword(int(os.Stdin.Fd()))
```

Using expect for complex interactions

- Some systems (such as network devices) often need complex interactions
- On Unix systems, this can be anything that prompts a user for input when running a program
- Not all programs will allow input via command arguments
  - Which is why any program you write should take input via arguments and config files
  - Allow output in easily digestible formats such as JSON

Using expect to install a package with APT

- We will use the most popular: github.com/google/goexpect
  - This project is archived, but is feature complete
  - Can be used with terminal apps, os.Exec as well as SSH
  - The documentation is not stellar
  - May not work on non-Linux systems
- The other popular package: github.com/Netflix/go-expect
  - Not archived, but also not updated for the same reason

```
e, _, err := expect.SpawnSSH(conn, 5*time.Second)
if err != nil {
    fmt.Println("Error: could not spawn ssh: ", err)
    os.Exit(1)
}
defer e.Close()
```

Uses the SSH connection

Annoyingly has package name different from package directory

Checking for the shell prompt and sending a command

- Compile some regexes that represents output we will look for
- Wait to see the prompt
- Send a command to use apt-get to install the TCL expect package

```
var (
    promptRE    = regexp.MustCompile(`\$ `)
    aptContRE     = regexp.MustCompile(`Do you want to continue\? \[Y/n\] `)
    aptAtNewestRE = regexp.MustCompile(`is already the newest`)
)

_, _, err = e.Expect(promptRE, 10*time.Second)
if err != nil {
    fmt.Println("did not get shell prompt")
    os.Exit(1)
}

if err := e.Send("sudo apt-get install expect\n"); err != nil {
    fmt.Println("error on send command: %s", err)
    os.Exit(1)
}
```

Handling the responses

- This handles the case where we are prompted to continue
- Also handles the case where APT says we have the newest version
- Will wait up to 10 seconds for these
- There are other tags:
  - expect.Fail()
  - expect.Continue()
  - expect.LogContinue()
  - expect.Next()
- You can use “S:” to send a string when the case hits
- Can use “Rt:” to indicate how many times to retry if tag is generated from expect.Continue() or expect.LogContinue()

```
_, _, ecase, err := e.ExpectSwitchCase(
    []expect.Caser{
        &expect.Case{
            R: aptContRE,
            T: expect.OK(),
        },
        &expect.Case{
            R: aptAtNewestRE,
            T: expect.OK(),
        },
    },
    10*time.Second,
)
if err != nil {
    fmt.Println("apt-get install did not send what we expected")
    os.Exit(1)
}
```

OK()
returns a tag that indicates we found what we want

Handling the responses (continued)

- If we receive the “continue” prompt, we send “Y” with a carriage return
- Otherwise we can continue
- Wait for the prompt to indicate that we are done
  - Note: 10 seconds might be too low for some installations

```
switch ecase {
case 0:
    if err := e.Send("Y\n"); err != nil {
        return err
    }
}

_, _, err = e.Expect(promptRE, 10*time.Second)
if err != nil {
    fmt.Println("did not get shell prompt")
    os.Exit(1)
}
```

Full code at: gofordevopsclass/automation_the_hard_way/expect

Writing a System Agent

Why a System Agent?

- Provides a more robust system for controlling a fleet of machines
- Change control can be localized
  - Work can survive network issues or service problems
  - State can be recorded locally for resuming after an unexpected reboot
- In essence, this is how large orchestration systems like Kubernetes control machines

What can you do with a System Agent

- Install and run services, even containerize them!
  - See talk by Liz Rice on: Writing a container from scratch in Go
    - She is the author of O'Reilly's "Container Security"
- Gather machine stats and program stats
- Gather machine inventory
- By defining a common RPC interface, you can control machines with different OS types in the same uniform way

Writing a simple System Agent
Requirements

- Download and run a program sent via a REST interface
- Provide system stats over HTTP
- We will use the popular “gin” package for our REST server

Note: A more complex version using gRPC and Systemd that containerizes applications can be found here:
github.com/PacktPublishing/Go-for-DevOps/tree/rev0/chapter/8/agent

Our main file

- Simply creates our agent from the service package
- Starts the agent

```go
package main

import (
    "log"

    "github.com/gin-gonic/gin"
    "github.com/johnsiilver/gofordevopsclass/automation_the_hard_way/agent/service"
)

func main() {
    agent, err := service.New(gin.Default())
    if err != nil {
        log.Fatalf("unable to create agent: %s", err)
    }
    if err := agent.Start(); err != nil {
        log.Fatalf("unable to start agent: %s", err)
    }
}
```

The Agent

```
const (
    // pkgDir is the directory in the Agent user's home where we are installing and
    // running packages. A more secure version would be to have the agent do this
    // in individual user directories that match some user on all machines. However
    // this is for illustration purposes only.
    pkgDir = "sa/packages/"

    maxInstallSize = 1 * 1024 * 1024 * 1024 // 1GB
)

// Agent provides a simple System Agent using REST to install and run programs.
type Agent struct {
    // homePath is the path to the user's home directory.
    homePath string
    // addr is the address to listen on.
    addr   string
    router *gin.Engine
}
```

The Agent (continued)

```go
// New creates a new Agent. If addr is empty, it will default to localhost:8080.
func New(router *gin.Engine, addr string) (*Agent, error) {
    u, err := user.Current()
    if err != nil {
        return nil, err
    }
    if addr == "" {
        addr = "localhost:8080"
    }

    homePath := ""
    switch runtime.GOOS {
    case "linux":
        homePath = filepath.Join("/home", u.Username)
    case "darwin":
        homePath = filepath.Join("/Users", u.Username)
    default:
        return nil, fmt.Errorf("unsupported OS: %s", runtime.GOOS)
    }

    agent := &Agent{
        homePath: homePath,
        router:   router,
        addr:     addr,
    }

    router.POST("/api/v1.0.0/install", agent.Install)
    return agent, nil
}
```

Start()

- Serves HTTP
  - No authentication
  - No transport security
- This is why we are serving it on localhost and not “:8080”

```
//go:build !ds

package service

// Start starts the agent.
func (a *Agent) Start() error {
    return a.router.Run(a.addr)
}
```

Alternate Start()

- Serves HTTP on a Unix socket
- You can use github.com/johnsiilver/serveonssh to allow connecting to the service over SSH
- Moves authentication to SSH and provides transport security

```go
//go:build ds

package service

import (
    "fmt"
    "net"
    "os"
    "path/filepath"
)

// Start starts the agent on a domain socket instead of an IP. To use this method,
// you must compile with "go build -tags ds".
func (a *Agent) Start() error {
    var sockAddr = filepath.Join(a.homePath, "/sa/socket/sa.sock")
    if err := os.MkdirAll(filepath.Dir(sockAddr), 0700); err != nil {
        return fmt.Errorf("could not create socket dir path: %w", err)
    }
    // Remove old socket file if it exists.
    os.Remove(sockAddr)

    l, err := net.Listen("unix", sockAddr)
    if err != nil {
        return fmt.Errorf("could not connect to socket: %w", err)
    }

    a.router.Run(a.addr)

    return a.router.RunListener(l)
}
```

The Agent (continued)

```go
// Install installs a package on the machine and starts it.
func (a *Agent) Install(c *gin.Context) {
    req, err := a.getInstallReq(*c.Request)
    if err != nil {
        sendInstallError(c, http.StatusBadRequest, err)
        return
    }

    from, err := a.unpack(req)
    if err != nil {
        sendInstallError(c, http.StatusBadRequest, err)
        return
    }
    if err := a.migrate(req, from); err != nil {
        sendInstallError(c, http.StatusBadRequest, err)
        return
    }
    if err := a.startProgram(c.Request.Context(), req); err != nil {
        sendInstallError(c, http.StatusBadRequest, err)
        return
    }

    c.IndentedJSON(http.StatusOK, msgs.InstallResp{})
}
```

The Agent (continued)

```go
// getInstallReq gets the msgs.InstallReq from the request body. It will return
// an error if the body is larger than maxInstallSize (1 GiB). Also runs the
// validation on the message.
func (a *Agent) getInstallReq(r http.Request) (*msgs.InstallReq, error) {
    lr := io.LimitedReader{
        R: r.Body,
        N: maxInstallSize,
    }
    defer r.Body.Close()

    b, err := io.ReadAll(&lr)
    if err != nil {
        return nil, fmt.Errorf("unable to read message body: %s", err)
    }

    req := &msgs.InstallReq{}
    if err := json.Unmarshal(b, req); err != nil {
        return nil, fmt.Errorf("unable to unmarshal message body: %s", err)
    }

    return req, req.Validate()
}
```

The Agent (continued)

```go
// unpack unpacks the zip file into a temp directory and returns the directory location.
func (a *Agent) unpack(req *msgs.InstallReq) (string, error) {
    dir, err := os.MkdirTemp("", fmt.Sprintf("sa_install_%s_*", req.Name))
    if err != nil {
        return "", err
    }
    r, err := zip.NewReader(bytes.NewReader(req.Package), int64(len(req.Package)))
    if err != nil {
        return "", err
    }

    // Iterate through the files in the archive, writing the files into our
    // temp directory.
    for _, f := range r.File {
        if err := a.writeFile(f, dir); err != nil {
            return "", err
        }
    }
    return dir, nil
}
```

The Agent (continued)

```go
// writeFile writes a zip file under the root directory dir.
func (a *Agent) writeFile(z *zip.File, dir string) error {
    if z.FileInfo().IsDir() {
        err := os.Mkdir(
            filepath.Join(dir, filepath.FromSlash(z.Name)),
            z.Mode(),
        )
        return err
    }

    rc, err := z.Open()
    if err != nil {
        return fmt.Errorf("could not open file %q: %w", z.Name, err)
    }
    defer rc.Close()

    nf, err := os.OpenFile(
        filepath.Join(dir, filepath.FromSlash(z.Name)),
        os.O_CREATE|os.O_WRONLY,
        z.Mode(),
    )
    if err != nil {
        return fmt.Errorf("could not open file in temp diretory: %w", err)
    }
    defer nf.Close()

    _, err = io.Copy(nf, rc)
    if err != nil {
        return fmt.Errorf("file copy error: %w", err)
    }
    return nil
}
```

The Agent (continued)

```go
// migrate shuts down any existing job that is running and migrates our files
// from the temp location to the final location.
func (a *Agent) migrate(req *msgs.InstallReq, from string) error {
    to := filepath.Join(a.homePath, pkgDir, req.Name)
    // We can only have one program running at a time.
    // Note: I did not implement something to stop any existing programs that are running.
    // This is trivial to implement.
    if _, err := os.Stat(to); err == nil {
        os.RemoveAll(to)
    }
    log.Println("from: ", from)
    log.Println("to: ", to)
    if err := os.Rename(from, to); err != nil {
        return err
    }
    return nil
}
```

The Agent (continued)

```
// startProgram starts our program manually.
// Note: You generally want to use a process manager like systemd, upstart, etc to manage
//
//  your programs. This is for illustration purposes only. You would also want something
//  to restart the program if it dies by monitoring the process.
func (a *Agent) startProgram(ctx context.Context, req *msgs.InstallReq) error {
    p := filepath.Join(a.homePath, pkgDir)
    cmdStr := fmt.Sprintf(
        "%s %s &",
        filepath.Join(p, req.Name, req.Binary),
        strings.Join(req.Args, " "),
    )
    cmd := exec.CommandContext(ctx, "/bin/sh", "-c", cmdStr)

    cmd.Stderr = nil
    cmd.Stdout = nil
    if err := cmd.Start(); err != nil {
        return err
    }
    log.Println("Started program: ", cmd.Process.Pid)
    log.Println("waiting: ", cmd.Wait())
    return nil
}
```

& indicates to run in background

Executing using shell

Give it a try

Start the agent

- cd [your path]/gofordevopsclass/automation_the_hard_way/agent
- go run agent.go

```
ElephantInTheRoom:agent jdoak$ go run agent.go
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.

[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] POST   /api/v1.0.0/install       --> github.com/johnsiilver/gofordevopsclass/automation_the_hard
(3 handlers)
[GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to set a value.
Please check https://pkg.go.dev/github.com/gin-gonic/gin#readme-don-t-trust-all-proxies for details.
[GIN-debug] Listening and serving HTTP on localhost:8080
2023/09/20 13:13:35 from:  /var/folders/rd/hbhb8s197633_f8ncy6fmpqr0000gn/T/sa_install_qotd_3717677974
2023/09/20 13:13:35 to:  /Users/jdoak/sa/packages/qotd
2023/09/20 13:13:35 Started program:  17795
2023/09/20 13:13:35 waiting:  <nil>
[GIN] 2023/09/20 - 13:13:35 | 200 |   73.637083ms |       127.0.0.1 | POST     "/api/v1.0.0/install"
```

Install the QOTD service

- cd [your path]/gofordevopsclass/automation_the_hard_way/agent/client/cli
- go run . install 127.0.0.1:8080 qotd ./qotd.zip server
    - qotd.zip has been provided for you in this directory.

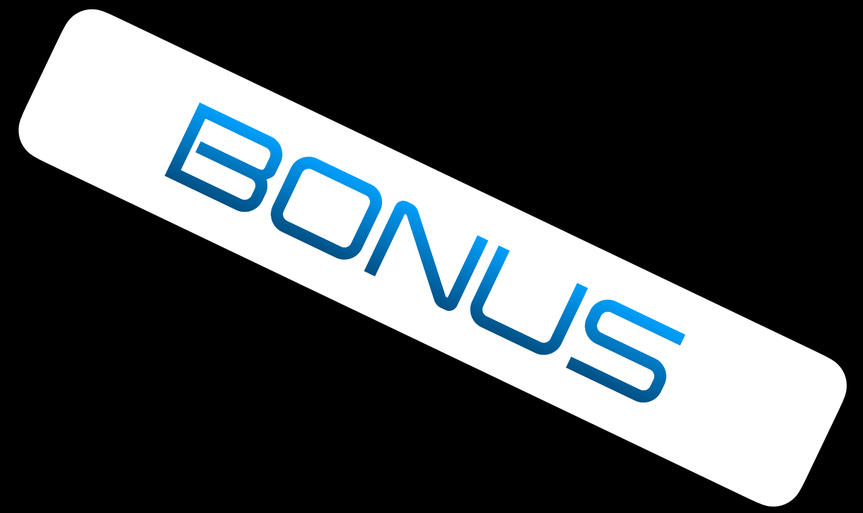
Get a quote

- if you have telnet installed:
    - telnet 127.0.0.1 17
- If not:
    - cd [your path]/gofordevopsclass/cli_apps/qotd/client/usingflags/
    - go run .

```
ElephantInTheRoom:usingflags jdoak$ go run .
Golf is a good walk spoiled
```

If you want to try the domain socket over SSH version

- run/build the binary with “-tags ds”
  - Both for the cli app and the agent
- The agent machine must have OpenSSH running
- Must either:
  - provide a private key for logging in via —key
  - The SSH agent must be able to log you in
  - Sorry, no passwords (but you could add it easily)



Exposing System Stats

Let’s get some system stats

- We are going to publish some CPU stats via expvar
  - You could publish via Prometheus or OpenTelemetry metrics (discussed later)
- You could also publish the stats for any program running through the agent
- Or whatever information you would like to scrape for data
- We are going to use the package github.com/shirou/gopsutil/cpu
  - And some custom code if you are on OSX
  - Sorry, no Windows (this irony that we work for Microsoft is not lost on us)

Update our Agent attributes

- Use atomic.Pointer for fast updates
  - Useful when you change an entire type out
  - Not great if you want to update a value in a slice or map

```go
// Agent provides a simple System Agent using REST to install and run programs.
type Agent struct {
    // homePath is the path to the user's home directory.
    homePath string
    // addr is the address to listen on.
    addr   string
    router *gin.Engine

    // cpuData is the atomic pointer to the CPU data.
    cpuData atomic.Pointer[msgs.CPUPerfs]
    // memData is the atomic pointer to the memory data.
    memData atomic.Pointer[msgs.MemPerf]
}
```

Collect CPU stats

- resolution is how often we collect
- Gather's per CPU stats
- Stores it with our atomic.Pointer

```go
func (a *Agent) collectCPU(ctx context.Context, resolution int32) error {
    stats, err := cpu.TimesWithContext(ctx, true)
    if err != nil {
        return err
    }

    v := &msgs.CPUPerfs{
        ResolutionSecs: resolution,
        UnixTimeNano:   time.Now().UnixNano(),
    }

    for _, stat := range stats {
        c := msgs.CPUPerf{
            ID:     stat.CPU,
            User:   stat.User,
            System: stat.System,
            Idle:   stat.Idle,
            IOWait: stat.Iowait,
            IRQ:    stat.Irq,
        }
        v.CPU = append(v.CPU, c)
    }
    a.cpuData.Store(v)
    return nil
}
```

Collect Memory Stats

- resolution is how often we collect
- Stores it with our atomic.Pointer

```go
func (a *Agent) collectMem(ctx context.Context, resolution int32) error {
    stats, err := mem.VirtualMemoryWithContext(ctx)
    if err != nil {
        return err
    }

    v := &msgs.MemPerf{
        ResolutionSecs: resolution,
        UnixTimeNano:   time.Now().UnixNano(),
        Total:          stats.Total,
        Free:           stats.Free,
        Avail:          stats.Available,
    }

    a.memData.Store(v)
    return nil
}
```

Gather our stats

- Collect stats every 10 seconds
- Cancel stat collection if it takes more than 10 seconds
- Publish via expvar

```
func (a *Agent) perfLoop() error {
    const resolutionSecs = 10

    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()
    if err := a.collectCPU(ctx, resolutionSecs); err != nil {
        return err
    }
    if err := a.collectMem(ctx, resolutionSecs); err != nil {
        return err
    }

    expvar.Publish(
        "system-cpu",
        expvar.Func(
            func() interface{} {
                return a.cpuData.Load()
            },
        ),
    )
    expvar.Publish(
        "system-mem",
        expvar.Func(
            func() interface{} {
                return a.memData.Load()
            },
        ),
    )
```

Gather our stats
(Continued)

- Sleep for our resolution time
- Concurrently gather all stats
- Don’t take longer than our resolution time

```
    go func() {
        wg := sync.WaitGroup{}
        for {
            time.Sleep(resolutionSecs * time.Second)

            ctx, cancel := context.WithTimeout(context.Background(), resolutionSecs)
            wg.Add(1)
            go func() {
                defer wg.Done()
                if err := a.collectCPU(ctx, resolutionSecs); err != nil {
                    log.Println(err)
                }
            }()
            wg.Add(1)
            go func() {
                defer wg.Done()
                if err := a.collectMem(ctx, resolutionSecs); err != nil {
                    log.Println(err)
                }
            }()
            wg.Wait()
            cancel()
        }
    }()
    return nil
}
```

Finish up

- Import the github.com/gin-contrib/expvar package
- Edit New() to start our perfLoop()
- Also add a debug route for expvar

```
"github.com/gin-contrib/expvar"
```

```
if err := agent.perfLoop(); err != nil {
    return nil, fmt.Errorf("problem starting perf loop: %w", err)
}

router.GET("/debug/vars", expvar.Handler())
router.POST("/api/v1.0.0/install", agent.Install)
```

Try it out

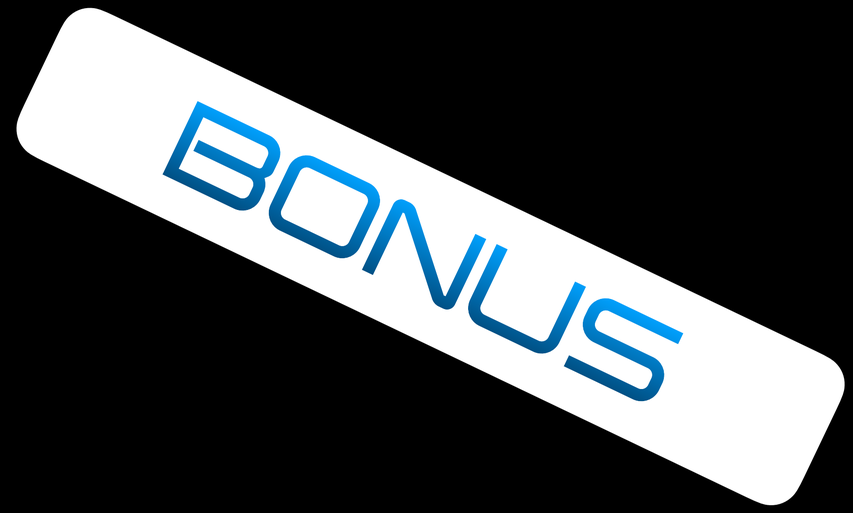
- http://127.0.0.1:8080/debug/vars

OSX Output


```
    "system-cpu": {
        "ResolutionSecs": 10,
        "UnixTimeNano": 1695324540709173000,
        "CPU": [
            {
                "ID": "CPU",
                "User": 11.93,
                "System": 14.82,
                "Idle": 73.23,
                "IOWait": 0,
                "IRQ": 0
            }
        ]
    },
    "system-mem": {
        "ResolutionSecs": 10,
        "UnixTimeNano": 1695324539580684000,
        "Total": 68719476736,
        "Free": 607436800,
        "Avail": 0
    }
```

Summary

- We've used os/exec to automate local changes
  - Automate using the "ping" command
  - Use the "ssh" command to log into a machine and run "uname"
- Using SSH for automating remote changes
  - How to authenticate with Google's SSH package
  - Issuing commands via SSH
  - Using Google's Expect package to handle interactive prompts
- Writing a system agent to do local work at scale
  - Automating a remote package install
  - Collecting system stats and exposing them



Designing safe, concurrent change automations

Aka: Building a small work orchestrator

Couldn’t I just use…

- Ansible
- Jenkins
- Insert other…..

Components of a change

- Global Preconditions - conditions of service or set of services required for an automation to run
- Local Preconditions - conditions of a running service on node, the node itself or whatever is the unit of change required for the automation to run on the work unit
- Actions - a set of work to execute on a unit
- Action validation - checks to make sure the action was successful
- Local post conditions - checks to make sure the unit of work is in a working state
- Global post conditions - checks to make sure the global state of a service or set of services is still in a good state

Example work

- Rollout a set of jobs to a set of Virtual Machines
- Jobs are behind a load-balancer
- We need to canary the rollout
- Validate health before and after

A version using SFTP and multiple machines can be found at: github.com/PacktPublishing/Go-for-DevOps/tree/rev0/chapter/8/rollout

Steps required

Workflow Steps

- Check our load balancer state (local pre condition)
- Run canary actions (make sure it works at all)
- Concurrently execute actions
- Validate our load balancer state (local post condition)

Action Steps

- Remove our job from a load balancer
- Kill the job on the VM or server
- Copy our new software to the server
- Start our service
- Check the service is reachable
- Add the job back to the load balancer

Overall Structure of a Machine Action

```go
// StateFn is a function that represents a state in the workflow.
type StateFn func(ctx context.Context) (StateFn, error)

// Actions is the set of actions to take on a single endpoint, one at a time.
type Actions struct {
    // endpoint is the machine to connect to.
    endpoint string
    // backend is the backend configuraiton in the load balancer.
    // This is used to remove and add the backend to the load balancer.
    backend client.IPBackend
    // config is the configuration for the workflow.
    config *config.Config
    // srcf is the file to copy to the remote machine.
    srcf *os.File
    // dst is the destination path on the remote machine to copy the file to.
    dst string
    // lb is the load balancer client.
    lb *client.Client

    // started indicates if the workflow has started.
    started bool
    // failedState is the state to start at if we have a failure.
    failedState StateFn
    // err is the error that caused the failure.
    err error
}
```

Overall Structure of a Machine Action (Continued)

```go
// Run runs the workflow.
func (a *Actions) Run(ctx context.Context) (err error) {
    a.srcf, err = os.Open(a.config.Src)
    if err != nil {
        a.err = fmt.Errorf("cannot open binary to copy(%s): %w", a.config.Src, err)
        return a.err
    }

    fn := a.findAppLocal
    if a.failedState != nil {
        fn = a.failedState
    }

    a.started = true
    for {
        if ctx.Err() != nil {
            a.err = ctx.Err()
            return ctx.Err()
        }
        fn, err = fn(ctx)
        if err != nil {
            a.failedState = fn
            a.err = err
            return err
        }
        if fn == nil {
            return nil
        }
    }
}
```

Example StateFn

```go
func (a *Actions) rmBackend(ctx context.Context) (StateFn, error) {
    err := a.lb.RemoveBackend(ctx, a.config.Pattern, a.backend)
    if err != nil {
        return nil, fmt.Errorf("problem removing backend from pool: %w", err)
    }

    return a.jobKill, nil
}
```

Other states defined

- findAppLocal(ctx context.Context) (StateFn, error)
- jobKill(ctx context.Context) (StateFn, error)
- cp(ctx context.Context) (StateFn, error)
- jobStart(ctx context.Context) (StateFn, error)
- reachable(ctx context.Context) (StateFn, error)
- addBackend(ctx context.Context) (StateFn, error)

Config file

```
{
    "Concurrency": 2,
    "CanaryNum": 1,
    "MaxFailures": 2,
    "Src": "/Users/jdoak/trees/...",
    "LB": "127.0.0.1:9091",
    "Pattern": "/",
    "Backends": [
            "127.0.0.1:9092",
            "127.0.0.1:9093",
            "127.0.0.1:9094",
            "127.0.0.1:9095",
            "127.0.0.1:9096",
            "127.0.0.1:9097",
            "127.0.0.1:9098",
            "127.0.0.1:9099"
    ]
}
```

```
// Config represents the configuration file that details the work to be done.
type Config struct {
    // Concurrency is the number of servers that can be upgraded at a time.
    Concurrency int32
    // CanaryNum is the number of canaries to do before proceeding with a general rollou
    // Any canary failure fails the workflow. Canaries execute one at a time.
    CanaryNum int32
    // MaxFailures is the maximum number of failures to tolerate before stopping.
    // You can have more failures than MaxFailures due to concurrency settings.
    MaxFailures int32
    // Src is the path on disk to the binary to push.
    Src string
    // LB is the host:port of the load balancer.
    LB string
    // Pattern is the load balancer's Pool pattern.
    Pattern string
    // Backends are the backends that need to be updated, simply the host in IP:Port fo
    Backends []string
}
```

Config file (Continued)

```go
// Validate does basic validation of the config.
func (s Config) Validate() error {
    if _, _, err := CheckIPPort(s.LB); err != nil {
        return fmt.Errorf("LB(%s) is not correct: %w", s.LB, err)
    }
    if len(s.Backends) < 1 {
        return fmt.Errorf("must specify some Backends")
    }
    for _, b := range s.Backends {
        _, _, err := CheckIPPort(b)
        if err != nil {
            return fmt.Errorf("Backend(%s) is not correct: %w", b, err)
        }
    }
    if strings.TrimSpace(s.Pattern) == "" {
        return fmt.Errorf("Pattern(%s) is invalid", s.Pattern)
    }
    if s.Concurrency < 1 {
        return fmt.Errorf("Concurrency(%d) is invalid", s.Concurrency)
    }
    return nil
}
```

The Workflow package

States

```
// EndStates are the final states after a run of a workflow.
//
run go generate ./... | run go generate
//go:generate stringer -type=endState
type EndState int8

const (
    // ESUnknown indicates we haven't reached an end state.
    ESUnknown EndState = 0
    // ESSuccess means that the workflow has completed successfully. This
    // does not mean there haven't been failurea.
    ESSuccess EndState = 1
    // ESPreconditionFailure means no work was done as we failed on a precondition.
    ESPreconditionFailure EndState = 2
    // ESCanaryFailure indicates one of the canaries failed, stopping the workflow.
    ESCanaryFailure EndState = 3
    // ESMaxFailures indicates that the workflow passed the canary phase, but failed
    // at a later phase.
    ESMaxFailures EndState = 4
)
```

The Workflow package (continued)

Workflow object definition

```go
// Workflow represents our rollout Workflow.
type Workflow struct {
    config *config.Config
    lb     *client.Client

    failures int32
    endState EndState

    actions []*actions.Actions
}

// New creates a new workflow.
func New(config *config.Config, lb *client.Client) (*Workflow, error) {
    wf := &Workflow{
        config: config,
        lb:     lb,
    }
    if err := wf.buildActions(); err != nil {
        return nil, err
    }
    return wf, nil
}
```

The Workflow package (continued)

Workflow runner

```go
// Runs runs our workflow on the supplied "actions" doing "canaryNum" canaries,
// then running "concurrency" number of actions that will stop at "maxFailures" number of
// failurea.
func (w *Workflow) Run(ctx context.Context) error {
    // Run a local precondition to make sure our load balancer is in a healthy state.
    preCtx, cancel := context.WithTimeout(ctx, 30*time.Second)
    err := w.checkLBState(preCtx)
    cancel()
    if err != nil {
        w.endState = ESPreconditionFailure
        return fmt.Errorf("checkLBState precondition fail: %s", err)
    }

    // Run our canaries one at a time. Any problem stops the workflow.
    for i := 0; i < len(w.actions) && int32(i) < w.config.CanaryNum; i++ {
        color.Green("Running canary on: %s", w.actions[i].Endpoint())
        ctx, cancel := context.WithTimeout(ctx, 10*time.Minute)
        err := w.actions[i].Run(ctx)
        cancel()
        if err != nil {
            w.endState = ESCanaryFailure
            return fmt.Errorf("canary failure on endpoint(%s): %w", w.actions[i].Endpoint(), err)
        }
        color.Yellow("Sleeping after canary for 1 minutes")
        time.Sleep(1 * time.Minute)
    }
```

The Workflow package (continued)

Workflow runner (continued)

```
    // Run the rest of the actions, with a limit to our concurrency.
    for i := w.config.CanaryNum; int(i) < len(w.actions); i++ {
        i := i
        limit <- struct{}{}
        if atomic.LoadInt32(&w.failures) > w.config.MaxFailures {
            break
        }
        wg.Add(1)
        go func() {
            defer func() { <-limit }()
            defer wg.Done()
            ctx, cancel := context.WithTimeout(ctx, 10*time.Minute)

            color.Green("Upgrading endpoint: %s", w.actions[i].Endpoint())
            err := w.actions[i].Run(ctx)
            cancel()
            if err != nil {
                color.Red("Endpoint(%s) had upgrade error: %s", w.actions[i].Endpoint(), err)
                atomic.AddInt32(&w.failures, 1)
            }
        }()
    }
    wg.Wait()

    if atomic.LoadInt32(&w.failures) > w.config.MaxFailures {
        w.endState = ESMaxFailures
        return errors.New("exceeded max failures")
    }
    w.endState = ESSuccess
    return nil
}
```

Run our example workflow

- Note: This example is going to use localhost ports 9090 - 9099
- cd to gofordevopsclass/automation_the_hard_way/workflow
- ./test.sh
- At the end, you should see:

```
Pool   Status
/      PS_FULL
Backend           Status
127.0.0.1:9092    BS_HEALTHY
127.0.0.1:9093    BS_HEALTHY
127.0.0.1:9094    BS_HEALTHY
127.0.0.1:9095    BS_HEALTHY
127.0.0.1:9096    BS_HEALTHY
127.0.0.1:9097    BS_HEALTHY
127.0.0.1:9098    BS_HEALTHY
127.0.0.1:9099    BS_HEALTHY
127.0.0.1:9100    BS_HEALTHY
```

Run our example workflow
(Continued)

- Browse to 127.0.0.1:9090
  - Should see something similar to:
    - Hello web from node d287f81f-8627-4b6a-a1ac-9f0decb84b59
- cd to: gofordevopsclass/automation_the_hard_way/orchestration
- go run . service.json

Run our example workflow
(Continued)

- Should see:

```
ElephantInTheRoom:orchestration jdoak$ go run . service.json
2023/09/25 12:59:00 READY
Starting Workflow
Running canary on: 127.0.0.1:9092
Sleeping after canary for 1 minutes
Upgrading endpoint: 127.0.0.1:9093
Upgrading endpoint: 127.0.0.1:9094
Upgrading endpoint: 127.0.0.1:9096
Upgrading endpoint: 127.0.0.1:9095
Upgrading endpoint: 127.0.0.1:9097
Upgrading endpoint: 127.0.0.1:9098
Upgrading endpoint: 127.0.0.1:9099
Workflow Completed with no failures
```

- Refresh browser at: 127.0.0.1:9090
  - You should see:
    - Hello web from replaced node 52a30a65-661d-4493-930b-92ea9d89a1f8

Run our example workflow
(Continued)

- Copy “pkill” line from test.sh
- Copy “rm -rf” line from test.sh
  - WARNING, double check the “rm -rf” for your environment