Building a Workflow System

Or how to automate with safety

Copyright John Doak

Prerequisites

- Go 1.18 or later installed
- A git repository to store your code
- A cloned copy of gitHub.com/johnsiilver/classes

What is a Workflow system

- A centralized service that all automation goes through
- The service receives a definition of work to be done and executes it on behalf of a client
- Note: sometimes called a Workflow Orchestration System

Why not just build an app/script?

- Applications/scripts that run on local machines loose centralized control
  - How do you stop a script running on someone's machine that is causing problems? Now think about this problem in the WFH world!
- Where do you put policy enforcement?
  - Its hard to enforce policy in lots of apps/scripts written by different engineers
- Where do you have a centralized view of your automations?
  - If you have to look at a log file, then you don't really have a centralized view

Components of a Workflow

A few definitions first

- Workflow - an entire body of work that is related
  - Examples:
    - Deploy service @version
    - Update certs on all machines
    - Update new telemetry endpoint addresses
- Node - a unit that represents a single collection of work in the Workflow that will be changed
  - Examples:
    - Modifying a single server state
    - Modifying a single process state
    - Modifying a group of “objects” that act as one

Change state actions

- An action that changes the state of something, somewhere
- Examples:
  - Run a command on a machine
  - Change data in a data store
  - Execute an RPC against a service that changes something
  - Run a CLI tool

Workflow Pre-checks

- Validates health parameters before starting work, failure prevents execution
- Examples:
  - Check service health
  - Check machine health
  - Check network health

Work Node Check

- Can be a pre-checks that verifies the Node can be worked on
- Can be a post-check that verifies the Node change was successful
- Failure causes that work unit to fail and moves on

Basic System Characteristics

- Sequential execution of actions on a node
- Concurrent execution of node work
- Maximum tolerated failure detection
- Three-way handshake

Defining our Service Proto

Service Definition

```
service Workflow {
    // Submit the work to the server. This will not execute the work, it will
    // simply verify it against policy and store it for execution.
    rpc Submit(WorkReq) returns (WorkResp) {};
    // Tell the service to execute a WorkReq submitted earlier.
    rpc Exec(ExecReq) returns (ExecResp) {};
    // Get the status of a WorkReq.
    rpc Status(StatusReq) returns (StatusResp) {};
}
```

A Work Request Message

```
// WorkReq is the definition of some work to be done by the system.
message WorkReq {
    // This is used to describe the work to be done. This name
    // must be authorized by having a policy with the same name
    // in the server's policies.json fie.
    string name = 1;
    // A description of what this is doing.
    string desc = 2;
    // These are groupings of Jobs. Each block is executed one at
    // a time.
    repeated Block blocks = 3;
}
```

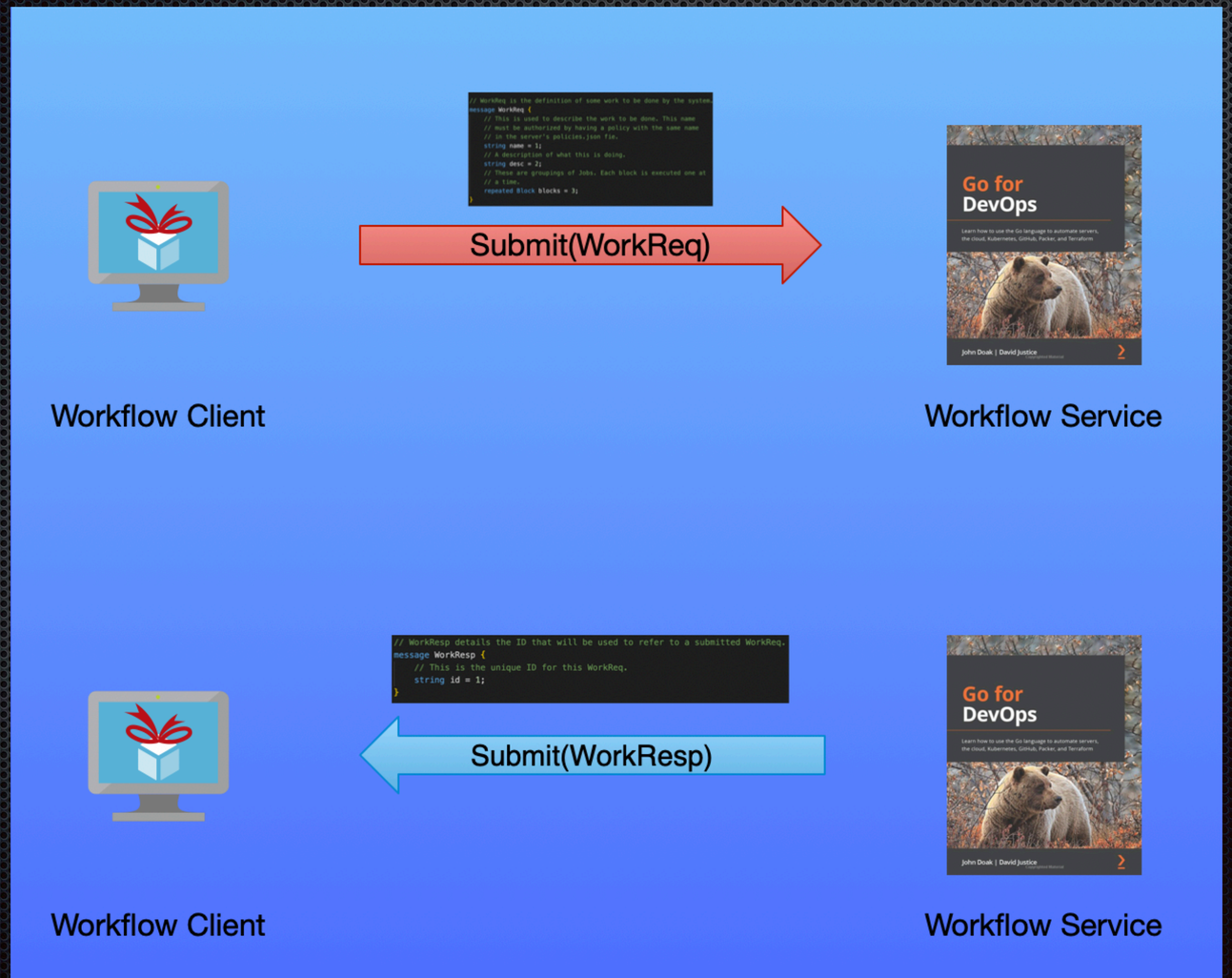
A Block Message

```
// Block is a grouping of Jobs that will be executed concurrently
// at some rate.
message Block {
    // This describes what the Block is doing.
    string desc = 1;
    // The amount of concurrency executions. < 1 will default to 1.
    int32 rate_limit = 2;
    // The Jobs to to execute in this Block.
    repeated Job jobs = 3;
}
```

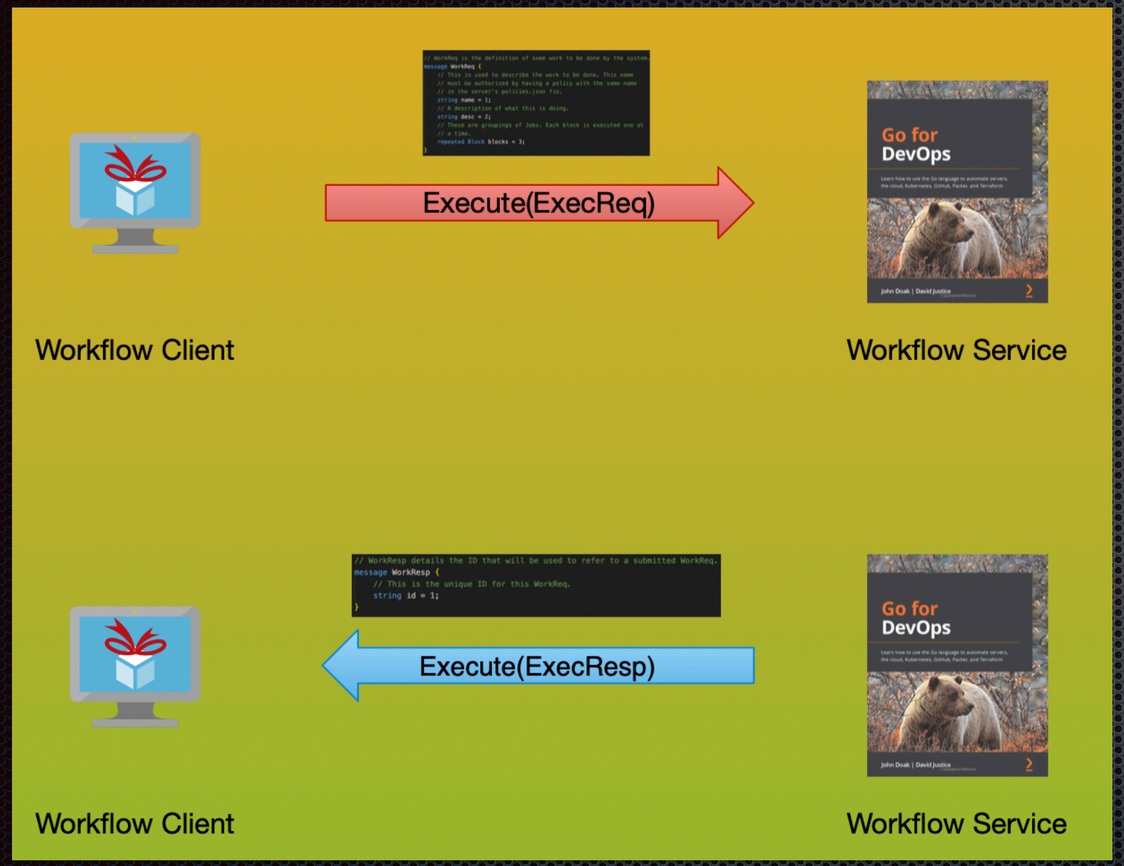
A Job Message

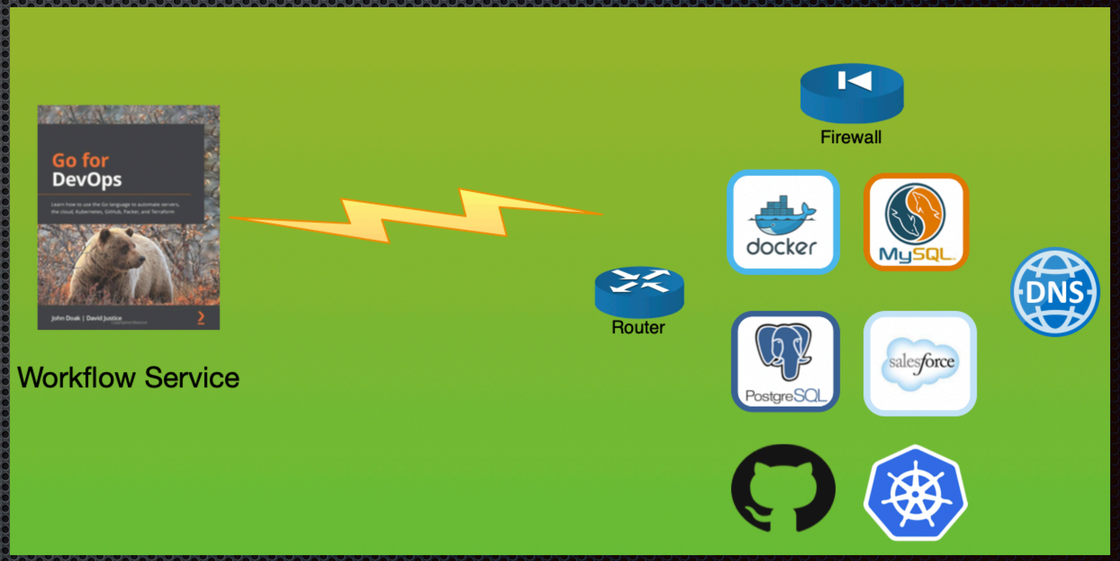
```
// Job refers to a Job action that is defined on the server.
message Job {
    // This is the name of the Job, which must be registered on
    // the server.
    string name = 1;
    // This is a description of what the job is doing.
    string desc = 2;
    // A mapping of key/value arguments. While the value is a string,
    // it can represent non-string data and will be converted by the
    // Job on the server. See the Job definition for a list of arguments
    // that are mandatory and optional.
    map<string, string> args = 3;
}
```

Workflow Submission



Workflow Execution





Work Response

```
// WorkResp details the ID that will be used to refer to a submitted WorkReq.
message WorkResp {
    // This is the unique ID for this WorkReq.
    string id = 1;
}
```

Execute Request/Response

```
// ExecReq is used to tell the server to execute a WorkReq
// that was previously submitted.
message ExecReq {
    // This is the unique ID of the WorkReq given back
    // by WorkResp.
    string id = 1;
}

// ExecResp is the response from an ExecReq.
message ExecResp {}
```

Status Types

```
// Status details the status of a Block or Job.
enum Status {
    // Indicates that there is some bug
    // as the Status for the object was not set.
    StatusUnknown = 0;
    // The WorkReq, Block or Job has not started execution.
    StatusNotStarted = 1;
    // The WorkReq, Block or Job is currently executing.
    StatusRunning = 2;
    // The WorkReq, Block or Job has failed.
    StatusFailed = 3;
    // The WorkReq, Block or Job has completed.
    StatusCompleted = 4;
}
```

Status Request/Response

```
// StatusReq requests a status update from the server.
message StatusReq {
    // The unique ID of the WorkReq.
    string id = 1;
}

// StatusResp is the status of WorkReq.
message StatusResp {
    // The name of the WorkReq.
    string name = 1;
    // The description of the WorkReq.
    string desc = 2;
    // The overall status of the WorkReq.
    Status status = 3;
    // The status information on the Blocks.
    repeated BlockStatus blocks = 4;
    // If we are SatusFailed or StatusCompleted, if
    // there were any errors when run.
    bool had_errors = 5;
    // If the WorkReq was stopped with emergency stop.
    bool was_es_stopped = 6;
}
```

Status Request/Response Cont

```
// BlockStatus holds the status of block execution.
message BlockStatus {
    // The description of the block.
    string desc = 1;
    // The status of the block.
    Status status = 2;
    // If there any errors in Jobs in the Block.
    bool has_error = 3;
    // The status of Jobs in the Block.
    repeated JobStatus jobs = 4;
}
```

Status Request/Response Cont

```
// JobStatus holds the status of the Jobs.
message JobStatus {
    // The name of the Job called.
    string name = 1;
    // The description of the Job.
    string desc = 2;
    // The args for the Job.
    map<string, string> args = 3;
    // The status of the Job.
    Status status = 4;
    // The error, if there was one.
    string error = 5;
}
```

Code Structure Layout

```
WORKFLOW
  client
    client.go
  internal/service
    executor
      executor.go
      jobs
        register
      jobs.go
    service.go
  proto
    buf.gen.yaml
    buf.yaml
    custom.go
    workflow_grpc.pb.go
    workflow.pb.go
    workflow.proto
```

- client.go contains the service client
- service.go implements the gRPC service
- executor.go handles workflow execution
- jobs.go handles registering a Job in the system
- register/ contains Job implementations

Implementing the Client

Needs of the client

- Build a workflow and send the service
- Execute a workflow
- Monitor a workflow

Basic Client Framework

```go
type Workflow struct {
    conn   *grpc.ClientConn
    client pb.WorkflowClient
}

func New(addr string) (*Workflow, error) {}

func (w *Workflow) Submit(ctx context.Context, req *pb.WorkReq) (string, error) {}

func (w *Workflow) Exec(ctx context.Context, id string) error {}

func (w *Workflow) Status(ctx context.Context, id string) (*pb.StatusResp, error) {}
```

New Constructor

```go
func New(addr string) (*Workflow, error) {
    conn, err := grpc.Dial(addr, grpc.WithInsecure())
    if err != nil {
        return nil, err
    }

    return &Workflow{
        conn:   conn,
        client: pb.NewWorkflowClient(conn),
    }, nil
}
```

Submit()

```go
func (w *Workflow) Submit(ctx context.Context, req *pb.WorkReq) (string, error) {
    resp, err := w.client.Submit(ctx, req)
    if err != nil {
        return "", err
    }
    return resp.Id, nil
}
```

Exec()

```go
func (w *Workflow) Exec(ctx context.Context, id string) error {
    _, err := w.client.Exec(ctx, &pb.ExecReq{Id: id})
    return err
}
```

Status()

- We leave this as an exercise for you to complete

Writing the Service

Needs of the Service

- Validate incoming workflows
- Read/Write workflows to storage
- Receive and validate workflow to be executed
- Execute workflow
- Report on the status of a workflow

Outline: jobs.go

```go
var jobs = map[string]Job{}

// Job executes some type of work.
type Job interface {
    // Validate validates that the Job settings sent to the server are valid.
    Validate(job *pb.Job) error
    // Run runs the Job settings.
    Run(ctx context.Context, job *pb.Job) error
}

// Register registers a Job so that it can be executed.
func Register(name string, job Job) {}

// GetJob returns a Job by its type from the registry.
func GetJob(name string) (Job, error) {}
```

Exercise: Build Registration functions

- Register() should register a Job with a unique name or panic
- Get() should retrieve a Job by name
- Write *WorkReq to storage using UUID as file name

Tips:

- Copy the following content into the file: https://go.dev/play/p/kDcZRfuQqDm

Exercise: Build a DiskErase Job

- jobs
  - register
    - diskerase
      - diskerase.go

- Note: For demo purposes, this will simply sleep and not erase a disk
- Use an init() statement to register your Job with the name "diskErase"
- Requires two arguments
  - machine: the name of the machine to erase
  - site: the name of the site the machine is at
- Job.Validate()
  - The machine name and site must both exist
  - Use the "sites" package to validate the site exists
  - Use the "sites" package to validate the machine exists
- Job.Run()
  - Simply sleep for 30 * time.Second

Tips:

- Get the "sites" package at:
  - github.com/johnsiilver/classes/workflow/data/sites

Solution Part I

```go
// This registers our Job on server startup.
func init() {
    jobs.Register("diskErase", newJob())
}

type args struct {
    machine string
    site    string
}
```

Solution Part II

```go
func (a *args) validate(args map[string]string) error {
    must := map[string]bool{"machine": false, "site": false}

    for k, v := range args {
        switch k {
        case "machine":
            must["machine"] = true
            a.machine = v
        case "site":
            if _, ok := sites.Data.Sites[v]; !ok {
                return fmt.Errorf("site(%s) arg was not a valid site", v)
            }
            must["site"] = true
            a.site = v

        default:
            return fmt.Errorf("invalid arg(%s)", k)
        }
    }

    for k, v := range must {
        if !v {
            return fmt.Errorf("missing required arg(%s)", k)
        }
    }

    fullName := fmt.Sprintf("%s.%s", a.machine, a.site)
    _, ok := sites.Data.Machines[fullName]
    if !ok {
        return fmt.Errorf("invalid arg(machine): machine(%s) does not exist", fullName)
    }
    return nil
}
```

Solution Part III

```go
// Job implements jobs.Job.
type Job struct {
    args args
}

func newJob() *Job {
    return &Job{}
}

// Validate implements jobs.Job.Validate().
func (j *Job) Validate(job *pb.Job) error {
    a := args{}
    if err := a.validate(job.Args); err != nil {
        return err
    }
    j.args = a
    return nil
}

// Run implements jobs.Job.Run().
func (j *Job) Run(ctx context.Context, job *pb.Job) error {
    time.Sleep(30 * time.Second) // A crude and inaccurate simulation of a disk erasure
    return nil
}
```

Exercise: Build a Sleep Job

- Register under “sleep”
- Jobs purpose is to sleep between actions
- Has argument, “seconds” which is the amount of time to sleep
  - Must be > 1 second

Exercise: Build a Decom Validator

- Register under “validateDecom”
- Arguments are:
  - site: The site name
  - type: The type of site
- Validate()
  - site must exist
  - site data must match the type
  - site status must be “decor”
  - Must be > 1 second
- Run() simply rechecks at runtime

Outline executor.go

```go
type Work struct {
    req *pb.WorkReq

    mu     sync.Mutex
    status *pb.StatusResp
    ch     chan *pb.StatusResp
}

// New is the constructor for Work.
func New(req *pb.WorkReq, status *pb.StatusResp) *Work {}

// Run validates that a WorkReq is correct and passed policy, then executes it.
func (w *Work) Run(ctx context.Context) chan *pb.StatusResp {}
func (w *Work) setWorkStatus(status pb.Status, esStopped bool) {}
func (w *Work) setBlockStatus(block *pb.BlockStatus, status pb.Status) {}
func (w *Work) setJobStatus(job *pb.JobStatus, status pb.Status, err string) {}
func (w *Work) sendStatus(status *pb.StatusResp) {}
func (w *Work) runJobs(ctx context.Context, block *pb.Block, blockStatus *pb.BlockStatus) error {}

func Validate(ctx context.Context, req *pb.WorkReq) error {}
```

Exercise: Build Executor

- Complete the runJobs() method
- Complete the Validate() method

Tips:

- Copy executor template from our directory workflow/internal/executor/template

Outline: service.Workflow

```go
// Workflow implements our gRPC service.
type Workflow struct {
    // storageDir is where we store workflow information.
    storageDir string

    // mu protects active
    mu sync.Mutex
    // active tracks all active work that is occuring.
    active map[string]*active

    // Required for gRPC to run, makes sure we have all the methods defined.
    pb.UnimplementedWorkflowServer
}

// New creates a new Workflow service.
func New(storageDir string) (*Workflow, error) {}

// Submit submits a request to run a workflow.
func (w *Workflow) Submit(ctx context.Context, req *pb.WorkReq) (*pb.WorkResp, error) {}

// Exec requests that the system execute a submitted workflow.
func (w *Workflow) Exec(ctx context.Context, req *pb.ExecReq) (*pb.ExecResp, error) {}

// Status is used to query for the status of a workflow.
func (w *Workflow) Status(ctx context.Context, req *pb.StatusReq) (*pb.StatusResp, error) {}
```

Define our active struct

```go
// active provides an entry for an actively executing workflow.
type active struct {
    work   *executor.Work
    status atomic.Value // *pb.StatusResp
}
```

Exercise: Build New() constructor

- Checks that storageDir exists
- Check that it can write to storageDir
- Cleans up anything it wrote
- Initialize the .active map
- Returns a *Workflow

Solution

```go
func New(storageDir string) (*Workflow, error) {
    stat, err := os.Stat(storageDir)
    if err != nil {
        return nil, fmt.Errorf("could not stat the workflow storage(%s): %w", storageDir, err)
    }
    if !stat.IsDir() {
        return nil, fmt.Errorf("storageDir(%s) is not a directory", storageDir)
    }
    u := "ping_" + uuid.NewString()
    p := filepath.Join(storageDir, u)
    f, err := os.OpenFile(p, os.O_CREATE+os.O_RDWR, 0600)
    if err != nil {
        return nil, fmt.Errorf("could not open a file in storage(%s) for RDWR: %w", storageDir, err)
    }
    f.Close()
    if err := os.Remove(p); err != nil {
        return nil, fmt.Errorf("could not remove ping file(%s) in storage(%s)", p, storageDir)
    }
    return &Workflow{storageDir: storageDir, active: map[string]*active{}}, nil
}
```

Exercise: Build .Submit()

- Calls executor.Validate() on the *WorkReq, timeout of 30 seconds
- Creates a UUID and assigns it to the *WorkReq
- Write *WorkReq to storage using UUID as file name
- Return a *WorkResp

Tips:

- Use gRPC status codes when returning errors (status.Errorf(codes.[code])
- Use path/filepath to manipulate file names
- Use Google's uuid package with UUIDv1 IDs (uuid.NewUUID)

Solution Part I

```go
func (w *Workflow) Submit(ctx context.Context, req *pb.WorkReq) (*pb.WorkResp, error) {
    validateCtx, cancel := context.WithTimeout(ctx, 30*time.Second)
    defer cancel()

    if err := executor.Validate(validateCtx, req); err != nil {
        if errors.Is(err, context.Canceled) {
            return nil, status.Error(codes.DeadlineExceeded, err.Error())
        }
        return nil, status.Error(codes.InvalidArgument, err.Error())
    }

    // Generate a unique ID.
    u, err := uuid.NewUUID()
    if err != nil {
        return nil, status.Errorf(codes.Internal, "problem getting UUIDv1; %s", err.Error())
    }
    id := u.String()
    p := filepath.Join(w.storageDir, id)

    resp := &pb.WorkResp{Id: u.String()}

    b, err := proto.Marshal(req)
    if err != nil {
        return nil, status.Errorf(codes.InvalidArgument, "could not marshal the request: %s", err)
    }
```

Solution Part II

```go
    f, err := os.OpenFile(p, os.O_CREATE+os.O_WRONLY, 0600)
    if err != nil {
        return nil, status.Errorf(codes.Internal, "could not open file at(%s): %s", w.storageDir, err)
    }
    defer f.Close()

    if _, err := f.Write(b); err != nil {
        return nil, status.Errorf(codes.Internal, "problem writing request to storage: %s", err)
    }

    return resp, nil
}
```

Exercise: Build .Exec()

- Error on call with running workflow
- Use UUIDv1 identifier to reject calls to execute a workflow older than 1 hour
- Writes status files containing a pb.StatusResp to: req.Id+"_status")
- Creates a new executor.Work object
  - Adds it to the .active map
  - Stores the status in the active entry
- Runs the Work object
- Spins off goroutine to monitor/update status of the worker until is completes

Tips:

- Use the two methods here to help with status messages:
  - https://go.dev/play/p/P81noP_Yd6P
- Use path/filepath to manipulate file names
- Use Google's uuid package with UUIDv1 IDs (uuid.NewUUID)

Solution Part I

```go
func (w *Workflow) Exec(ctx context.Context, req *pb.ExecReq) (*pb.ExecResp, error) {
    p := filepath.Join(w.storageDir, req.Id)
    statP := filepath.Join(w.storageDir, req.Id+"_status")

    w.mu.Lock()
    defer w.mu.Unlock()

    _, ok := w.active[req.Id]
    if ok {
        return nil, status.Errorf(codes.AlreadyExists, "Workflow(%s) is already running", req.Id)
    }

    _, err := os.Stat(statP)
    if err == nil {
        return nil, status.Errorf(codes.AlreadyExists, "Workflow(%s) already executing or executed", req.Id)
    }

    u, err := uuid.Parse(req.Id)
    if err != nil {
        return nil, status.Errorf(codes.InvalidArgument, "Id(%s) is not a valid value: %s", req.Id, err)
    }

    t := time.Unix(u.Time().UnixTime())
    if time.Since(t) > 1 * time.Hour {
        return nil, status.Errorf(codes.FailedPrecondition, "Id(%s) is older than 1 hour and cannot be started'
    }
```

Solution Part II

```go
b, err := os.ReadFile(p)
if err != nil {
    return nil, status.Errorf(codes.NotFound, "Workflow(%s) not found", req.Id)
}

workReq := &pb.WorkReq{}
if err := proto.Unmarshal(b, workReq); err != nil {
    return nil, status.Errorf(codes.Internal, "Workflow(%s) could not be unmarshalled: %s", req.Id, err)
}

// Write our status file to indicate we have started working on this.
statusResp := statusFromWork(workReq)
statusB, err := proto.Marshal(statusResp)
if err != nil {
    return nil, status.Errorf(codes.InvalidArgument, "could not marshal the work's status proto: %s", err)
}

if err := os.WriteFile(statP, statusB, 0600); err != nil {
    return nil, status.Errorf(codes.Internal, "problem writing status to storage: %s", err)
}

work := executor.New(workReq, statusResp)
active := &active{work: work}
active.status.Store(proto.Clone(statusResp).(*pb.StatusResp))
w.active[req.Id] = active
```

Solution Part III

```go
// Run our work and get the first state change.
ch := work.Run(context.Background())
active.status.Store(<-ch)
writeIn := statusWriter(statP)

// Update our status as it changes in memory and on disk.
// Cleanup our list of active work when we are done.
go func() {
    for status := range ch {
        // Record our status in memory
        active.status.Store(status)

        // Record our status on disk. If there is an entry pending,
        // remove it for the latest entry.
        select {
        case writeIn <- status:
        default:
            select {
            case <-writeIn:
            default:
            }
            writeIn <- status
        }
    }
    w.mu.Lock()
    delete(w.active, req.Id)
    w.mu.Unlock()
}()

return &pb.ExecResp{}, nil
```

Exercise: Build .Status()

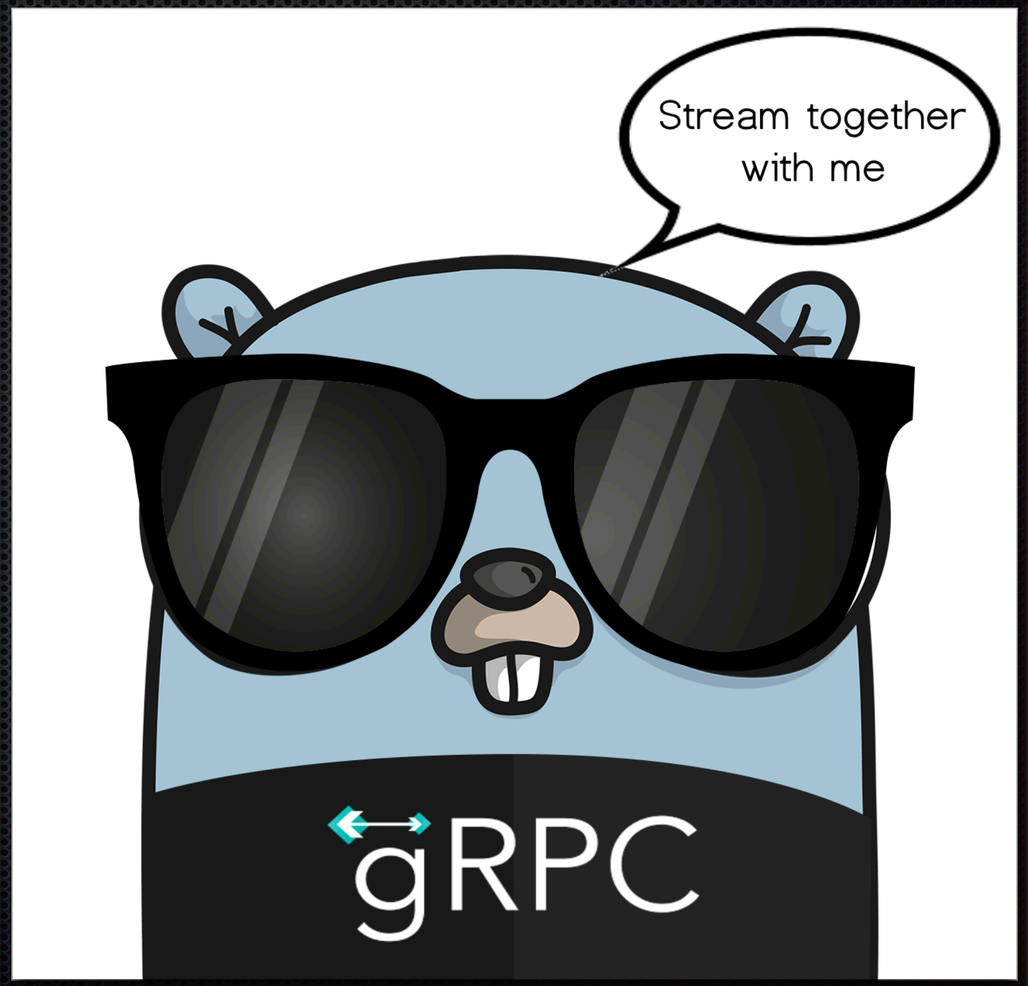
- Return status of any matching active workflow
- If not an active workflow, find a [uuid]_status file that matches
  - Decode it
  - Return its status
- Otherwise, return an error

Solution

```go
func (w *Workflow) Status(ctx context.Context, req *pb.StatusReq) (*pb.StatusResp, error) {
    w.mu.Lock()
    a := w.active[req.Id]
    w.mu.Unlock()
    if a != nil {
        return a.status.Load().(*pb.StatusResp), nil
    }
    // This ID is not currently running, so look in storage.
    p := filepath.Join(w.storageDir, req.Id+"_status")
    b, err := os.ReadFile(p)
    if err != nil {
        return nil, status.Errorf(codes.InvalidArgument, "work ID(%s) was not found", req.Id)
    }
    resp := &pb.StatusResp{}
    if err := proto.Unmarshal(b, resp); err != nil {
        return nil, status.Errorf(codes.Internal, "work ID(%s) data was corrupted on disk", req.Id)
    }
    return resp, nil
}
```

Tying it all together with workflow.go

- Must import the Jobs we created via “side effect” import
- Creates our “workflows” directory
- Starts the service



Running a sample disk erase workflow

- App that erases satellite disks
- Divides erasures into blocks of 5 machines
- Executes in alphabetical order
- Waits 1 minute between executing each block
- Each block runs at concurrency 5

```
Updates every 10 seconds
Time: Wed, 03 Aug 2022 09:17:23 PDT
Workflow: aa353b96-1347-11ed-b56d-6a8086e3250c
Name: SatelliteDiskErase
Description: Erasing disks in datacenter satellite aap

Running Block(1): disk erase machines aa00-aa04
Job Number  Desc                                        Status
0           Erase satellite(aap) machine(aa00) disk     StatusCompleted
1           Erase satellite(aap) machine(aa01) disk     StatusCompleted
2           Erase satellite(aap) machine(aa02) disk     StatusCompleted
3           Erase satellite(aap) machine(aa03) disk     StatusCompleted
4           Erase satellite(aap) machine(aa04) disk     StatusCompleted
5           Wait 1 minute between disk erasures         StatusRunning

Block Summaries
Block Number  Desc                          Status
0             Check pre-conditions          StatusCompleted
1             disk erase machines aa00-aa04 StatusRunning
2             disk erase machines aa05-aa09 StatusNotStarted
3             disk erase machines ab00-ab04 StatusNotStarted
4             disk erase machines ab05-ab09 StatusNotStarted
5             disk erase machines ac00-ac04 StatusNotStarted
6             disk erase machines ac05-ac09 StatusNotStarted
7             disk erase machines ad00-ad04 StatusNotStarted
8             disk erase machines ad05-ad09 StatusNotStarted
9             disk erase machines ae00-ae04 StatusNotStarted
10            disk erase machines ae05-ae09 StatusNotStarted
```

Code Walkthrough

Next Class

- Add an emergency stop capability to the system
- Build a policy engine to enforce system wide policies