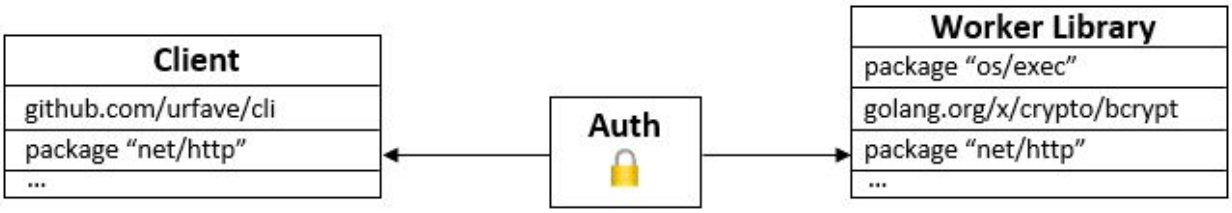
Linux Job Worker

Overview

`worker` will be a command-line service for running arbitrary Linux processes, written in the Go programming language.



The Worker Library

The worker library will handle the execution of jobs passed by the client. It will execute these jobs concurrently, keeping track of each one in a log stored in memory. In the log, each job will be represented by a unique ID (an auto-incrementing integer), its status, and its output. Jobs requiring standard input will block, but still log any output that occurs up to that point.

The API

A locally-hosted RESTful API will be responsible for making the functionality of the server available to the client, and it will provide the following endpoints:

POST **/jobs/run**
Create a job for the server to execute.
Body parameters: { job **object** }
job: { command **string**, args **[]string** }
name: the name of the job command (e.g. “echo”)
args: the arguments for the job (e.g. [“Hello,” “world!”]
Success response: { id **integer** }

GET **/jobs/{job_id}/status**
Get the status of a job.
Success response: { id **integer**, status **string** }

GET **/jobs/{job_id}/out**
Get the output of a job.
Success response: { id **integer**, output **string** }

PUT **/jobs/{job_id}/kill**
Terminate a job running on the server.
Success response: { id **integer**, status **string** }

The CLI

The user must first run the server, and then may use the CLI to schedule commands in a separate terminal instance. Usage of the CLI will involve specifying a worker action and any necessary parameters, for example:

```
$ ./worker help # view worker actions and usage examples
$ ./worker run echo hello # provide a job for the server to execute
$ ./worker status [job ID] # view the status of a given job
$ ./worker out [job ID] # view the output of a given job
$ ./worker kill [job ID] # stop a given job from executing
```

After scheduling a job, the client will not wait for it to finish execution before allowing the user to issue additional commands. It will merely wait for the server's response containing the job's assigned ID (or an error message) so it may be echoed to the user.

Error Handling

On the client side, providing `worker` with an invalid action will result in a Client Error and print the help text. There are, of course, syntactically correct worker actions that are destined to fail on the server, for example, `$ worker run echi hello`. In this scenario, the worker library will catch the error resulting from the attempt to execute the job. The job's log entry will have its status field updated to "failed" and its output field updated to a user-readable representation of the error.

Another situation that may raise an error is the attempted termination of a job that does not exist or has already completed execution. When this occurs, the API's response will indicate the reason that a job could not be terminated. A case worth mentioning is that terminating a process may fail if the process finishes execution in the small window of time between the user issuing the command and the API receiving the request.

Security

Encryption

To enable HTTPS for the API, a private key and self-signed certificate will be pre-generated and placed at the root of the repository. It should be noted that, while certificates are public-by-nature, a private key should never be exposed in a production environment. The client will be configured to trust only the CA used to create the pre-generated certificate. This way, another self-signed certificate with identical information will not be trusted by the client.

Authentication

The client will pass hard-coded credentials in each API request's Authorization header. The API will wrap its handler functions in authentication checks. These checks will consult a function that determines whether the HTTP request's Authorization header matches the correct username and password. It will validate the request's password by passing it through the **bcrypt** hashing function and checking it against the hashed version of the correct password.

Authorization

The server will implement a simple authorization system that only allows users to access endpoints for jobs they created. API requests must also pass a couple of additional authorization checks:

- *Rate-limiting*: requests from the same user must be separated by a number of milliseconds.
- *Request origin*: only local (not remote) requests will be accepted.