Contents

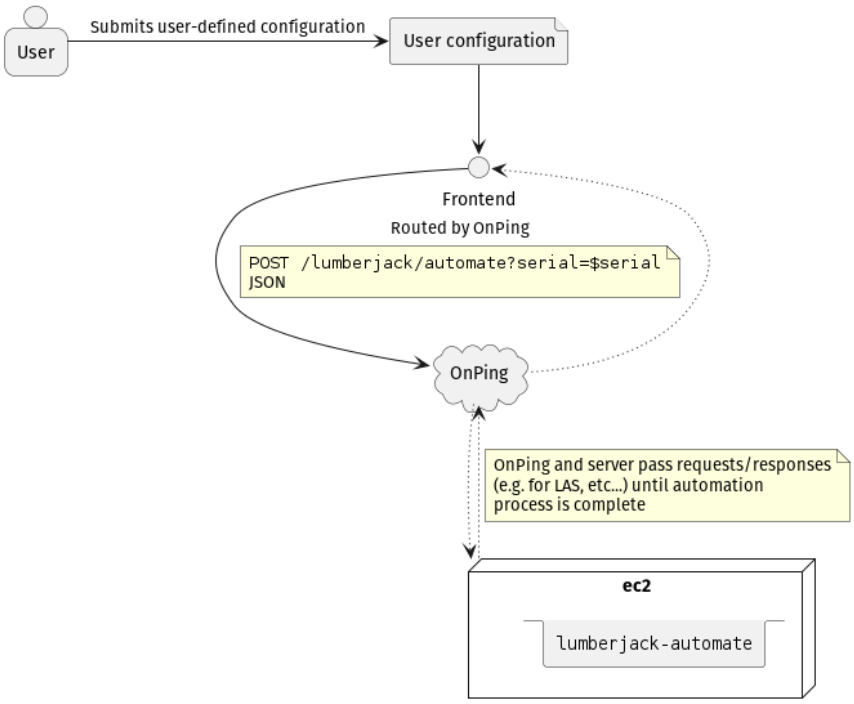
This is a simplified example of an automated lumberjack configuration process with example Nix code. It illustrates how a user could install and configure the `single-well-controllogix` driver. The workflow and example code make the following assumptions:

- The lumberjack to be configured has already been created (and has a serial number)
- We have a method of reusing existing `onping-api` (or other OnPing) routes
- The only driver configuration option for `single-well-controllogix` corresponds to calling OnPing's `/control/logix/location/add` route
  - Note that this simplifies the example modules presented, because we do not need to deal with any impure intermediate results
- The Haskell executable (referred to as `lumberjack-automate`) has been deployed on a server

Diagrams

Deployment

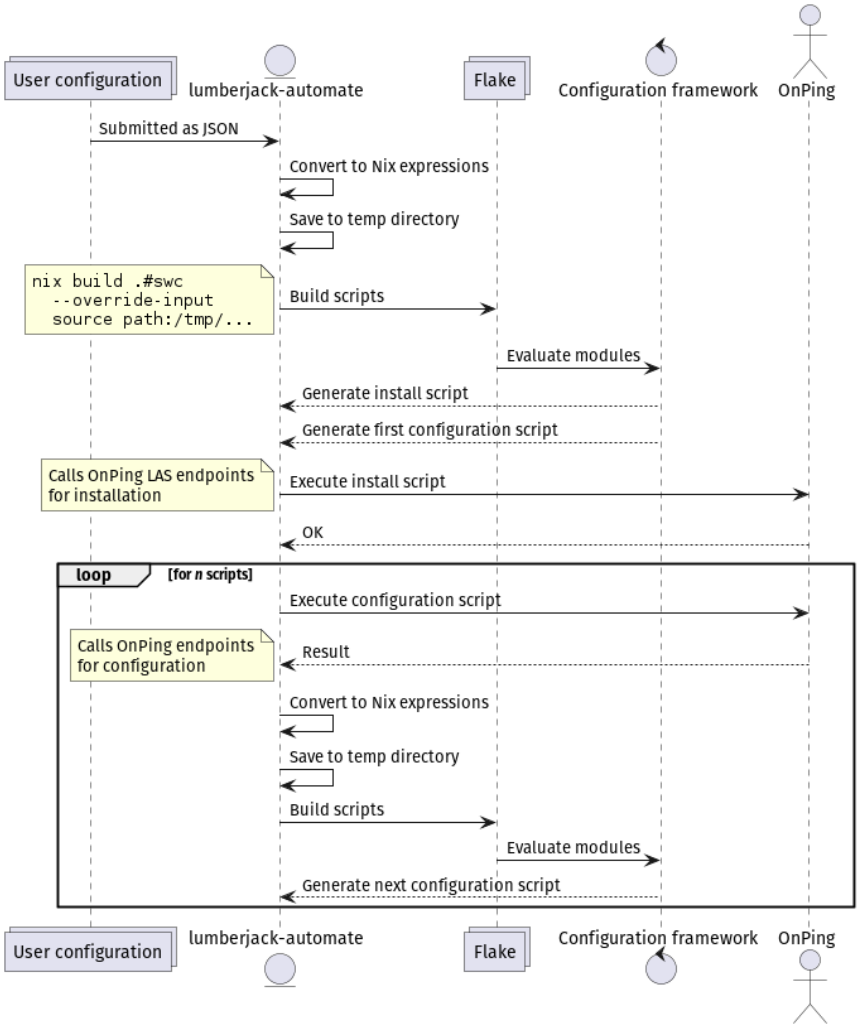
This illustrates a simplified deployment. `lumberjack-automate` is assumed to be deployed on a server having been configured to interact with the flake (i.e. SSH authentication and binary cache setup):



The operations of `lumberjack-automate` are illustrated below.

`lumberjack-automate`

This illustrates the processes and interfaces necessary for the driver installation and configuration to occur:



Examples

Note that this example is simplified and assumes that we can generate the installation script and a single configuration script for `single-well-controllogix` at the same time. This makes the example more concise but in practice, each driver will have several separate modules, which is not illustrated below.

Flake

As described in the original outline, the configuration framework would be exposed as applied functions from a dedicated flake. The flake would use an input to parameterize functions that generate the configuration scripts using the Nix module system.

```
{
  inputs = {
    nixpkgs.follows = "all/nixpkgs";
    all.url = "ssh+git://git@github.com:plow-technologies/all";
    # By default, this is an empty Nix expression. It will
    # be overriden by `lumberjack-automate` to evaluate the
    # modules
    source.url = "path:./noop.nix";
    source.flake = false;
  };
  outputs = { self, nixpkgs, all, source, ... }:
    let
      system = "x86_64-linux";
      pkgs = all.legacyPackages.${system};
      # These are functions that evaluate the Nix
      # expression using the modules illustrated
      # below, ultimately generating the configuration
      # scripts
      modules = import ./modules {
        inherit pkgs;
      };
    in
    {
      # We can use `legacyPackages` instead of
      # `packages` in order to nest the several
      # configuration scripts
      legacyPackages = {
        single-well-controllogix =
          modules.single-well-controllogix
            (import source);
        ;
      };
    };
}
```

Configuration modules

The configuration framework modules would be defined in the same repository as the flake. For this example, a single module is illustrated that would generate an installation script and a configuration script (to add a location) for `single-well-controllogix`. In reality, each driver will almost certainly have separate modules for generating different configuration scripts.

```
# In modules/single-well-controllogix.nix
let
  cfg = config.drivers.single-well-controllogix;
in
{
  options = {
    enable = lib.mkEnableOption
      (lib.mdDoc "single-well-controllogix");

    configuration = lib.mkOption {
      description = lib.mdDoc "...";
      type = lib.submodule {
        options = {
          name = { /* ... */ };
          plcPort = { /* ... */ };
          pollTimeSeconds = { /* ... */ };
          companyName = { /* ... */ };
          siteId = { /* ... */ };
          groupId = { /* ... */ };
          # Optional
          latitude = { /* ... */ };
          longitude = { /* ... */ };
        };
      };
    };
  };

  config =
    let
      # We can also use common options for all modules
      inherit (config.onping)
        url port;

      conf = builtins.toJSON
        (
          cfg.configuration // {
            # This would also be a common option
```

```
            lumberjackID = config.lumberjack.id;
          }
        );
      # We can use the package set from `all` to create the package set
      # to install. This could also be a single installation script for
      # all of the modules (instead of per driver). This is one reason
      # that the flake needs to interface directly with `all` (so we
      # know which Nix paths each package corresponds to, etc...)
      packages = utils.mkPackageSet { /**/ };
    in
    lib.mkIf cfg.enable {
      scripts = {
        install = pkgs.writeShellApplication {
          name = "install-swc";
          runtimeInputs = [ pkgs.curl ];
          text = ''
            curl -X POST \\
             ${url}:${toString port}/lumberjack/deploy/package/install \\
              -d '${packages}'
          '';
        };
        # This uses the generated controllogix configuration to add a
        # location with the user input
        configure = pkgs.writeShellApplication {
          name = "configure-swc";
          runtimeInputs = [ pkgs.curl ];
          text = ''
            curl -X POST ${url}:${toString port}/control/logix/location/add \\
              -d '${conf}'
          '';
        };
      };
    };
}
```