the developer's handbook

Abstract

i (mark), heavily use nix to manage my projects, either with devbox or flakes

if you are going to develop for surplus or its' sibling projects (except surplus on wheels, which only needs `shfmt` and `shellcheck`), i would recommend you install Nix using Determinate Systems' Nix Installer:

```
curl --proto '=https' --tlsv1.2 -sSf -L https://install.determinate.systems/nix | sh -s -- install
```

if i, a very very inexperienced Nix and NixOS user to give a rundown on why to use it:

1. **environments and builds are reproducible**

   nix is part package manager, operating system (as NixOS), and functional programming language. because it's functional, having X as an input will always produce Y as an output, no matter what. even in the event of environmental, social, economic, or structural collapse

   what does this mean for you?
   when i use `nix develop` and you use `nix develop` to start a development environment, your version of python will be the same as my version of python. your version of go will be same as my version of go, etc

   if i can build it, and the locked inputs of the nix flake are still available on the internet, you can build it too

2. **the nix store**, located literally at `/nix/store`
   it's where it stores every package you need, separate and isolated from other packages. lets say you have a tool that needs python (3.8), and another tool that needs python (3.11). nix store will download and store the binaries for both python installations, instead of sharing the earliest downloaded python version for both tools

   what does this mean for you?
   whatever project you're working on that can use nix for development environments and builds will not dirty anything else on your system. any build dependencies of surplus provided with `nix develop` will **not** mess up software installed for other projects or even the system. neat, if you ask me tbh

tl;dr: things will just werk with nix. but if you see all of this and go, "eh. i can manage what i install.", then power to you! i list down exactly what prerequisite software needs to be installed for each project anyways, so have fun! (●'◡'●)

surplus (and Documentation)

environment setup

> **Note**
>
> all prerequisite software are available in a nix flake. if you want a reproducible environment, run `nix develop --impure` in the repository root
>
> - for NixOS users, nix-ld is needed. if you use flakes to declare your system, follow accordingly. else, use `/etc/nixos/configuration.nix`
>
>   for NixOS-on-WSL users, use nix-ld-rs
>
>   once you're done installing `nix-ld` or `nix-ld-rs`, don't forget to run `sudo nixos-rebuild switch`

prerequisite software:

- Python, 3.11 or newer
- Hatch: dependency management and build tool

to start a development environment:

```
hatch shell
```

for docs:

```
hatch -e docs shell
```

workflow for python code

TODO

workflow for markdown documentation

run the documentation server with:

```
hatch run docs:serve
```

i personally don't use a linter for markdown files, if it looks good on my code editor, then whatever. if you're going to contribute back, i ask for three things:

- run it through a spell checker or something similar
- line limit of 100
- should be readable as-is on a code editor, **not the markdown preview pane**.

  my stance is, if you can afford a fancy preview of the markdown file, use the nice-ned documentation website. else, read it as a plaintext file

  (make it look pretty on the doc site and in plaintext)

---

surplus on wheels

environment setup

> **Note**
>
> all prerequisite software are available in a nix flake. if you want a reproducible environment, run `nix develop` in `src/surplus-on-wheels`

prerequisite software:

- shfmt: formatter
- ShellCheck: static analyser

workflow

> **Note**
>
> alternatively, run `check.sh` inside `src/surplus-on-wheels`

- formatting s+ow:
  - run `shfmt s+ow > s+ow.new`
  - mv `s+ow.new` into `s+ow`
    sometimes when piping shfmt's output immediately into the same file results in the file being empty :(
- checking s+ow:
  - run `shellcheck s+ow`
    if there's no output, that means it passed :)
- if commiting back into the repository, try it out on your Termux system for a day or two, just to make sure it runs correctly

---

surplus on wheels: Telegram Bridge

environment setup

> **Note**
>
> all prerequisite software are available in a nix flake. if you want a reproducible environment, run `nix develop` in `src/spow-telegram-bridge`. it uses poetry2nix, so you won't need to run `poetry shell` afterwards. if you've changed the `pyproject.toml` file, just exit and re-run `nix develop`

prerequisite software:

- Python, 3.11 or newer
- Poetry: dependency management and build tool

to start a development environment:

```
poetry shell
```

workflow

after modifying,

1. check the source code:
    a. `mypy bridge.py`
    b. `ruff format bridge.py`
    c. `ruff check bridge.py`

    > **Note**
    >
    > alternatively, run `check.sh` inside `src/spow-telegram-bridge`

2. and then test the binary

if the bridge behaves nominally, bump the version and commit!

---

surplus on wheels: WhatsApp Bridge

environment setup

> **Note**
>
> all prerequisite software are available in a nix flake. if you want a reproducible environment, run `nix develop` in `src/spow-whatsapp-bridge`
>
> the flake will pull in the Android SDK and NDK for building on Termux, and as such can only be ran on `x86_64-linux` and `x86_64-darwin`

prerequisite software:

- Go: 1.22 or newer
- Android NDK, if building for Termux

workflow for modifying bridge code

the bridge's code is just modified mdtest code, and as such, whenever in doubt, do a diff between mdtest and the bridge code

after modifying,

1. check the source code:
    a. `go fmt bridge.go`
    b. `go vet bridge.go`
    c. `golint bridge.go`

    > **Note**
    >
    > alternatively, run `check.sh` inside `src/spow-whatsapp-bridge`

2. build a binary
3. test the binary
4. and if all goes well, bump the version and commit!

workflow for bumping dependencies

- check with your editor, plugin, or online if there's newer patch/minor (see semantic versioning) versions to update to
- change the `go.mod` accordingly

after bumping,

1. build a binary
2. test the binary

3. and if all goes well, bump the version and commit!

workflow for building a binary

ensure you already have c compiler on the system (if you're using `nix develop` then yes you do), then run:

```
CGO_ENABLED=1 go build
```

nix users can alternatively run:

```
nix build
```

instructions to build a Termux build are located at the bridges' documentation page, however nix users can run the following instead for a reproducible, deterministic and hermetic build command:

```
nix build .#termux
```

the resulting build will be in `result/spow-whatsapp-bridge`

workflow for testing the binary

- test it out, making sure that you write dummy test text to `~/.cache/s+ow/message` before running the binary
    a. run `s+ow-whatsapp-bridge login` first
    b. run `s+ow-whatsapp-bridge list` if you don't already have a chat ID to send the test message to
    c. run `s+ow-whatsapp-bridge` type or copy and paste in a `wa:`-prefixed chat ID after it logs in, and verify it sends

if the bridge behaves nominally, bump the version and commit!

workflow for versioning and tagging releases

versioning surplus

format: `YEAR.MAJOR.MINOR[-PRERELEASE]` (semantic versioning)
example: `2024.0.0`, `2024.0.0-beta`
change: update the `__version__` variable in `src/surplus/surplus.py`

versioning surplus on wheels

i've tried to make surplus on wheels as reliable as it could be given a POSIX compliant shell and commands you'd find available on virtually every linux system, Termux included

as such, it doesn't really follow a versioning scheme as it doesn't need to. also there's no automatic updater for it, which would be overkill anyway

versioning surplus on wheels: Telegram Bridge

format: `REVISION.YYYY.WW[+BUILD]` (calendar versioning)
example: `2.2024.24`, `2.2024.24+1`
change: `version` key in `src/spow-telegram-bridge/pyproject.toml`

`REVISION` here meaning any general revision/change

the Telegram Bridge relies on Telethon, which also follows semantic versioning. so, as long as major isn't bumped, or as long as Telegram doesn't become Discord, the MTProto APIs to talk to Telegram should be stable.

however because Telethon also relies on a bunch of networking libraries, it made some sense to still do weekly builds to bump dependencies, getting pipx to download the newest compatible dependencies as compared to dubiously running some sort of script to `pipx inject` dependencies

under normal circumstances, a non-working version of the bridge would and **should not have a version bump**. but for any reason if an already tagged bridge is faulty and/or erroneous in normal/expected usage, add a revision number to the end after a period (see example above)

versioning surplus on wheels: WhatsApp Bridge

format: `REVISION.YYYY.WW[+BUILD]` (calendar versioning)
example: `2.2024.25`, `2.2024.25+1`
change: `version` attribute of `bridge` attribute set in `src/spow-whatsapp-bridge/flake.nix`

`REVISION` here meaning any general revision/change

the WhatsApp Bridge relies on whatsmeow, a rolling release library due to the volatile, undocumented nature of WhatsApp's multidevice API and also directly and indirectly relies on a bunch of networking libraries:

src/spow-whatsapp-bridge/go.mod

```
module forge.joshwel.co/mark/surplus/src/spow-whatsapp-bridge

go 1.22.3

require (
    github.com/mattn/go-sqlite3 v1.14.22
    github.com/mdp/qrterminal/v3 v3.2.0
    go.mau.fi/whatsmeow v0.0.0-20240603101645-64bc969fbe78
    google.golang.org/protobuf v1.34.2
)

require (
    filippo.io/edwards25519 v1.1.0 // indirect
    github.com/google/uuid v1.6.0 // indirect
    github.com/gorilla/websocket v1.5.3 // indirect
    github.com/mattn/go-colorable v0.1.13 // indirect
    github.com/mattn/go-isatty v0.0.20 // indirect
    github.com/rs/zerolog v1.33.0 // indirect
    go.mau.fi/libsignal v0.1.0 // indirect
    go.mau.fi/util v0.4.2 // indirect
    golang.org/x/crypto v0.24.0 // indirect
    golang.org/x/net v0.26.0 // indirect
    golang.org/x/sys v0.21.0 // indirect
    golang.org/x/term v0.21.0 // indirect
    rsc.io/qr v0.2.0 // indirect
)
```

as such, it uses a calendar versioning scheme and is built weekly

under normal circumstances, a non-working version of the bridge would and **should not have a version bump**. but for any reason if an already tagged bridge is faulty and/or erroneous in normal/expected usage, add a revision number to the end after a period (see example above)

---

i've made my changes. what now?

if you're contributing back to surplus and/or the sibling projects, firstly, thanks! see the contributor's handbook for what's next