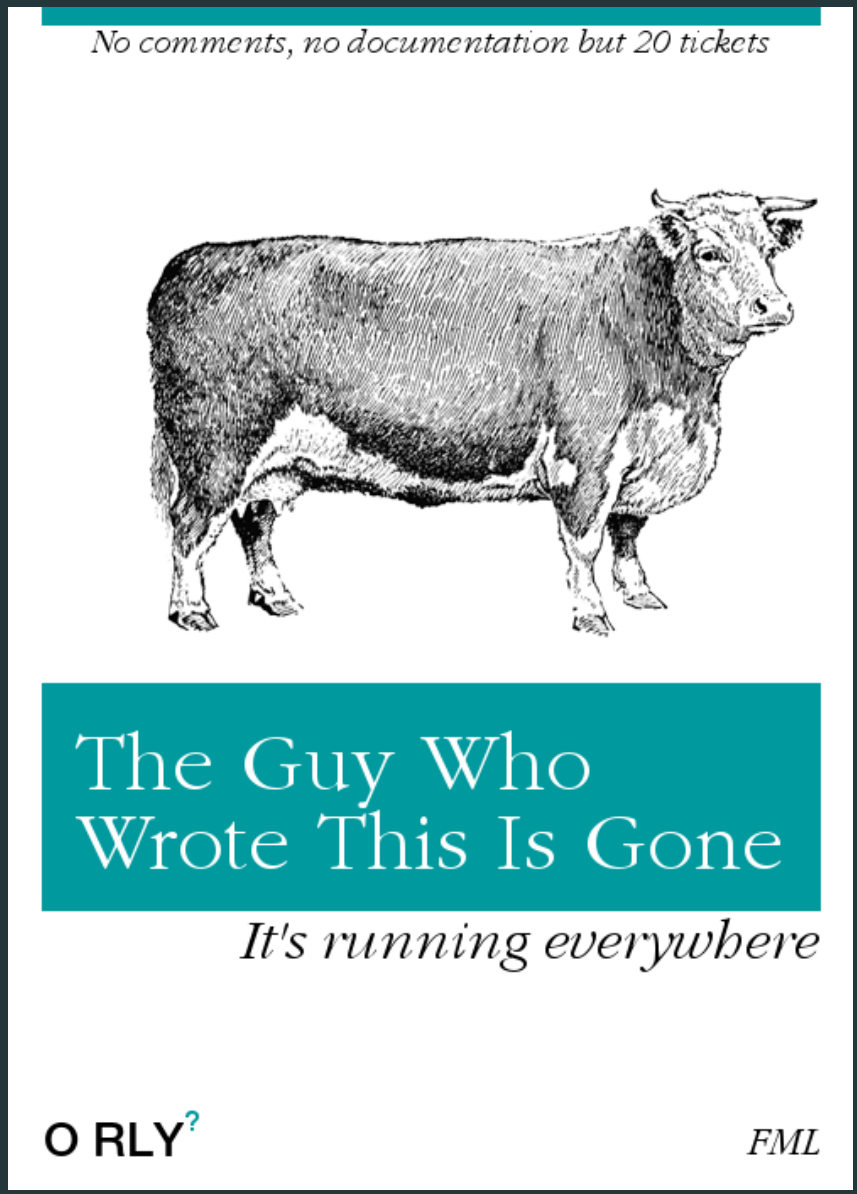
Nix in Status

What fresh madness is this?!

Jakub Grzegorz Sokołowski

05/06/2020

Contents Overview



Entrypoints

The two main files that are the starting point for Nix in `status-react` are:

https://github.com/status-im/status-react/blob/develop/default.nix

```
# for passing build options, see nix/README.md
{ config ? { } }:
let
  main = import ./nix { inherit config; };
in {
  # this is where the --attr argument selects the shell or target
  inherit (main) pkgs targets shells;
  inherit (main.pkgs) config;
}
```

https://github.com/status-im/status-react/blob/develop/shell.nix

```
# for passing build options, see nix/README.md
{ config ? { } }:
let
  main = import ./nix { inherit config; };
in
  # use the default shell when calling nix-shell without arguments
  main.shells.default
```

Entrypoints - Usage

Just running `nix-shell` starts the `default` shell:

```
 > nix-shell

[nix-shell:~/work/status-react]$ echo $name
status-react-shell
```

To pick a shell the `default.nix` file needs to be used:

```
 > nix-shell -A shells.android default.nix
No changes detected.
Done in: 0.392s

[nix-shell:~/work/status-react]$
```

This is possible because of this line in `default.nix`:

```
inherit (main) pkgs targets shells
```

Entrypoints - Building

You can build any target defined in `default.nix` using `nix-build`

```
 > nix-build --no-out-link -A targets.status-go.mobile.android
/nix/store/i6c9ps6n8fjahgw1dg2s2ybx0vvqs-status-go-0.54.0-0bffeab-android
```

This works because:

- `targets` is exposed in `default.nix` which loads:
- `status-go` that is exposed in `nix/targets.nix` which loads:
- `mobile` that is exposed in `nix/status-go/default.nix` which loads:
- `android` that is exposed in `nix/status-go/mobile/default.nix`

In effect we are calling the derivation defined in `nix/status-go/mobile/default.nix`.

Entrypoints - Targets

All main build targets are defined in `nix/targets.nix`:

```
{
  config ? {},
  pkgs ? import ./pkgs.nix { inherit config; }
}:

let
  inherit (pkgs) stdenv callPackage;

  status-go = callPackage ./status-go { };
  desktop = callPackage ./desktop { inherit status-go; };
  mobile = callPackage ./mobile { inherit status-go; };
in {
  inherit mobile desktop status-go;
}
```

Mobile and desktop targets use `status-go` so they inherit it.

We'll explain how that works in the later section.

Entrypoints - Shells

Created using `mkShell` and combined using `mergeSh`.

https://github.com/status-im/status-react/blob/develop/nix/shells.nix

```
default = mkShell {
  name = "status-react-shell"; # for identifying all shells

  buildInputs = with pkgs; lib.unique ([
    # core utilities that should always be present in a shell
    bash curl wget file unzip flock
    git gnumake jq ncurses gnugrep parallel
    lsof # used in start-react-native.sh
    # build specific utilities
    clojure maven watchman
    # other nice to have stuff
    yarn nodejs python27
  ]
  ...
```

Scripts

The `nix/scripts` directory contains a series of Bash scripts that simplify use of Nix.

```
 > ls -l nix/scripts
total 40
-rwxr-xr-x 1 sochan sochan 1769 Jun  2 17:56 build.sh
-rwxr-xr-x 1 sochan sochan 2833 Jun  2 17:56 clean.sh
-rwxr-xr-x 1 sochan sochan  446 Jun  2 17:56 gcroots.sh
-rwxr-xr-x 1 sochan sochan 4097 Jun  2 21:14 node_modules.sh
-rwxr-xr-x 1 sochan sochan  732 Jun  2 17:56 purge.sh
-rwxr-xr-x 1 sochan sochan 1404 Jun  2 17:56 setup.sh
-rwxr-xr-x 1 sochan sochan 2482 Jun  2 17:56 shell.sh
-rwxr-xr-x 1 sochan sochan  795 Jun  2 17:56 source.sh
```

They are mostly just wrappers around existing Nix commands to provide some default flags and support use of some environment variables.

Scripts - source.sh

It's used by almost all other scripts to source the Nix profile file, normally found in ${HOME}/.nix-profile/etc/profile.d/nix.sh.

nix/scripts/source.sh

```
function source_nix() {
  # Just stop if Nix is already available
  if [[ -x $(command -v nix) ]]; then
    return
  elif [[ -f "${NIX_PROFILE_SH}" ]]; then
    # Load Nix profile if it exists
    source "${NIX_PROFILE_SH}"
    return
  else
    # Setup Nix if not available
    ${GIT_ROOT}/nix/scripts/setup.sh
  fi

  # Load Nix profile
  source "${NIX_PROFILE_SH}"

  ...
```

Scripts - shell.sh

The nix/scripts/shell.sh script is essentially just a wrapper around nix-shell that handles the TARGET env variable:

```
if [[ -z "${TARGET}" ]]; then
    TARGET="default"
    echo -e "${YLW}Missing TARGET, assuming default target.${RST}" 1>&2
fi
entryPoint="default.nix"
nixArgs+=("--attr shells.${TARGET}")
```

And with appends additional default flags.

```
# ENTER_NIX_SHELL is the fake command used when `make shell` is run.
# It's a special string, not a variable, and a marker to not use `--run`.
if [[ "${@}" == "ENTER_NIX_SHELL" ]]; then
  exec nix-shell ${nixArgs[@]} ${entryPoint}
else
  exec nix-shell ${nixArgs[@]} --run "$@" ${entryPoint}
fi
```

It's written in such a way to allow our Makefile to use it for its SHELL.

Scripts - build.sh

Same before the nix/scripts/build.sh is a wrapper around nix-build that provides some default flags:

```
# Some defaults flags, --pure could be optional in the future.
# NOTE: The --keep-failed flag can be used for debugging issues.
nixOpts=(
  "--pure"
  "--fallback"
  "--no-out-link"
  "--show-trace"
  "--attr" "${targetAttr}"
)

# Run the actual build
echo "Running: nix-build "${nixOpts[@]}" "${@}" default.nix"
nixResultPath=$(nix-build "${nixOpts[@]}" "${@}" default.nix)

echo -e "\n${YLW}Extracting result${RST}: ${BLD}${nixResultPath}${RST}"

extractResults "${nixResultPath}"

echo -e "\n${GRN}SUCCESS${RST}"
```

Nix Packages

We use a fork of `nixpkgs` repo which applies a few small fixes defined in `nix/pkgs.nix`:

```
nixpkgsSrc = fetchFromGitHub {
  name = "nixpkgs-source";
  owner = "status-im";
  repo = "nixpkgs";
  rev = "6dacca5eb43a8bfb02fb09331df607d4465a28e9";
  sha256 = "0whwzll9lvrq4gg5j838skg7fqpvb55w4z7y44pzib32k613y2qn";
};
```

The custom `nixpkgs` are loaded from `pkgs.nix` via arguments for `default.nix`:

```
{
  config ? {},
  pkgs ? import ./pkgs.nix { inherit config; }
}:
...
```

Which is alternative to the standard way to load default `nixpkgs`:

```
{ pkgs ? import <nixpkgs> { } }:
```

Nix Packages - Config

Nix packages are instantiated by passing a `config` argument in the attribute set:

```
{
  status-im = {
    build-type = "pr";    # influences which .env file gets used
    build-number = 9999; # versionCode(android) and CFBundleVersion(iOS)

    android = {
      gradle-opts = null;     # Gradle options passed for Android builds
      keystore-path = null;   # Path to keystore for signing the APK
      abi-split = false;      # If APKs should be split by architectures
      abi-include = "armeabi-v7a;arm64-v8a;x86"; # Android architectures
    };
    nimbus = { src-override = null; };
    status-go = { src-override = null; };
  };
  # Android SDK requires an accepted license
  android_sdk.accept_license = true;
  # Android Env still needs old OpenSSL
  permittedInsecurePackages = [ "openssl-1.0.2u" ];
}
```

https://github.com/status-im/status-react/blob/develop/nix/config.nix

Nix Packages - Overlay

We apply and `overlay` defined in `nix/overlay.nix` via argument for `nixpkgs`

```
# import nixpkgs with a config override
(import nixpkgsSrc) {
  config = defaultConfig // statusDefaults // config;
  overlays = [ pkgsOverlay ];
}
```

The overlay contains additional or modified packages:

```
# Custom packages
aapt2 = callPackage ./pkgs/aapt2 { };
gomobile = callPackage ./pkgs/gomobile { };
qt5custom = callPackage ./pkgs/qt5custom { };
qtkeychain-src = callPackage ./pkgs/qtkeychain-src { };
appimagekit = callPackage ./pkgs/appimagekit { };
patchMavenSources = callPackage ./pkgs/patch-maven-srcs { };
goMavenResolver = callPackage ./pkgs/go-maven-resolver { };
```

Which makes them available in all expressions imported with `callPackage`.

Nix Packages - Packages

Custom packages are defined in nix/pkgs folder. For example:

nix/pkgs/go-maven-resolver/default.nix

```
{ lib, buildGoPackage, fetchFromGitHub }:

let
  inherit (lib) strings;
in buildGoPackage rec {
  pname = "go-maven-resolver";
  version = strings.substring 0 7 rev;
  owner = "status-im";
  repo = pname;
  rev = "72b6c12ab193f59d197e7f63273ec0c079b6e3a9";
  sha256 = "0r6k74716175pjy8vjz0fig2h010fabl00w114qd2wb2iwq4z3x4";
  goPackagePath = "github.com/${owner}/${repo}";
  goDeps = ./deps.nix;

  src = fetchFromGitHub {
    name = "${repo}-${version}-source";
    inherit owner repo rev sha256;
  };
}
```

Dependencies

Dependencies are managed in `nix/deps` and define in Nix consumable way how to provide dependencies normally managed by other package managers.

They are pulled in into `nixpkgs` via `nix/overlay.nix`:

```
  # Project dependencies
  deps = {
    clojure = callPackage ./deps/clojure { };
    gradle = callPackage ./deps/gradle { };
    nodejs = callPackage ./deps/nodejs { };
    nodejs-patched = callPackage ./deps/nodejs-patched { };
    react-native = callPackage ./deps/react-native { };
  };
```

https://github.com/status-im/status-react/blob/develop/nix/overlay.nix#L22-L29

Dependencies - Node.js

Node.js modules are handled using `yarn2nix` tool which takes `package.json` and `yarn.lock` as input and generates a Nix store package with `node_modules`.

```
{ lib, yarn2nix-moretea }:

let
  version = lib.fileContents ../../../VERSION;
  yarnLock = ../../../mobile/js_files/yarn.lock;
  packageJSON = ../../../mobile/js_files/package.json;
  packageJSONContent = lib.importJSON packageJSON;
in
  # Create a package for our project that contains all the dependencies
  yarn2nix-moretea.mkYarnModules rec {
    pname = "status-react";
    name = "${pname}-node-deps-${version}";
    inherit version packageJSON yarnLock;
  }
```

https://github.com/status-im/status-react/blob/develop/nix/deps/nodejs/default.nix

Dependencies - Node.js Patched

Modules provided by `yarn2nix` are normally fine, but we use `react-native-*` packages which have their own `gradle.build` files that reference external Maven repositories:

```
repositories {
    google()
    jcenter()
}
```

And these need to be patched and replaced with `mavenLocal()` to make sure Gradle doesn't try to fetch dependencies from remote repos.

This is done by `nix/deps/nodejs-patched/default.nix`, which applies other fixes.

```
for modBuildGradle in $(find -L ./node_modules -name build.gradle); do
    relativeToNode=${modBuildGradle#*node_modules/}
    moduleName=${relativeToNode%%/*}
    if [[ -L ./node_modules/$moduleName ]]; then
        unlink ./node_modules/$moduleName
        cp -r ${deps.nodejs}/node_modules/$moduleName ./node_modules/
        chmod u+w -R ./node_modules/$moduleName
    fi
    ${patchMavenSources} $modBuildGradle
done
```

Dependencies - Clojure

Clojure dependencies are collected by calling `nix/deps/clojure/generate.sh` which parses the output of the following command:

```
> yarn shadow-cljs classpath
```

And generates `nix/deps/clojure/deps.json` for all the found dependencies. It essentially contains details necessary to download all the JARs:

```
[
  {
    "path": "args4j/args4j/2.0.26/args4j-2.0.26",
    "host": "https://repo1.maven.org/maven2",
    "pom": {
      "sha1": "f0aada195340b361a1a45aa9c3d417c7cbb0a6b5",
      "sha256": "0151vnbmbjnr5089ik8np91r3a4g2ylcahih22zr49a3ilri6f80"
    },
    "jar": {
      "sha1": "01ebb18ebb3b379a74207d5af4ea7c8338ebd78b",
      "sha256": "193i0xgn295bdz5gk4zrk4ncfa29x8vl97bfd01kl1za44ixm6wq"
    }
  },
  ...
```

Dependencies - Gradle

The nix/deps/gradle/generate.sh script works in 4 steps:

1. Calls get_projects.sh to get list of Gradle sub-projects:

```
./gradlew projects --no-daemon --console plain 2>&1 \
    | grep "Project ':" \
    | sed -E "s;^.--- Project '\:([@_a-zA-Z0-9\-]+)';\1;"
```

2. Calls get_deps.sh for each project acquired previously:

```
./gradlew --no-daemon --console plain \
    "${BUILD_DEPS[@]}" \
    "${NORMAL_DEPS[@]}" \
    | awk -f ${AWK_SCRIPT}
```

3. Passes the found dependencies through go-maven-resolver to get URLs:

```
cat ${DEPS_LIST} | go-maven-resolver | sort -uV -o ./deps.urls
```

4. Generates deps.json for Nix consumption with url2json.sh:

```
parallel --will-cite --keep-order \
    "${CUR_DIR}/url2json.sh" \
    ::: ${URLS} \
    >> ./deps.json
```

Derivations

All of those overlays, packages, tools, and dependencies allow us to run:

```
 > nix-build --no-out-link -A targets.mobile.android.release
/nix/store/izvrkmjf1vkr07bhrlyyqyl4f8w-status-react-build-release-android

 > cd $(nix-build --no-out-link -A targets.mobile.android.release)
 > ls -l
total 71857
-r--r--r-- 1 root root 73482594 Jan  1  1970 app-release.apk
```

And get back a ready APK.

But the real question is how the derivations are defined.

Derivations - jsbundle

Combines the repo JS and Clojure sources as well as Node modules and Clojure dependencies to generate the `index.js` which can be used with Android build:

```
  buildPhase = ''
    # Assemble CLASSPATH from available clojure dependencies.
    # We append 'src' so it can find the local sources.
    export CLASS_PATH="$(find ${deps.clojure} \
      -iname '*.jar' | tr '\n' ':')src"

    # target must be one of the builds defined in shadow-cljs.edn
    java -cp "$CLASS_PATH" clojure.main \
      -m shadow.cljs.devtools.cli release mobile
  '';
```

https://github.com/status-im/status-react/blob/develop/nix/mobile/android/jsbundle/default.nix

Derivations - status-go

In nix/status-go/default.nix we can see that desktop and mobile versions of status-go are built from respective sub-directories:

```
  mobile = callPackage ./mobile {
    inherit meta source goBuildFlags goBuildLdFlags;
  };

  desktop = callPackage ./desktop {
    inherit meta source goBuildFlags goBuildLdFlags;
  };
```

The common settings like goBuildFlags are set in the higher level default.nix while the specific build types are defined in the sub folders:
nix/status-go/mobile/default.nix

```
  android = callPackage ./build.nix {
    platform = "android";
    # 386 is for android simulator
    architectures = [ "arm" "arm64" "386" ];
    outputFileName = "status-go-${source.shortRev}.aar";
    inherit meta source goBuildFlags goBuildLdFlags;
  };
```

Derivations - Android

Android APK builds are done in `nix/mobile/android/release.nix` and essentially just run Gradle after preparing all the necessary dependencies and environment variables.

The number of arguments you can pass to the Android derivation called for a separate sript for calling it: `scripts/release-android.sh`

It handles passing `config` arguments like `build-type` or `build-number` via `BUILD_TYPE` and `BUILD_NUMBER` environment variables. But the most crucial part is how it passes the Keystore for signing the resulting APK.

```
cfg+="status-im.android.keystore-path=\"$(must_get_env KEYSTORE_PATH)\";"

nixOpts+=(
 "--option" "extra-sandbox-paths" "${KEYSTORE_PATH} ${SECRETS_FILE_PATH}"
)
nixOpts+=(
  "--arg" "config" "{${cfg}}"
)

${NIX_SCRIPTS}/build.sh targets.mobile.android.release "${nixOpts[@]}"
```

It uses `extra-sandbox-paths` to avoid putting the keystore into the Nix store.

Derivations - iOS

In case of iOS we prepare a custom shell with all the necessary tools:

```
shell = mkShell {
  buildInputs = with pkgs; [
    xcodeWrapper watchman bundler procps
    flock # used in nix/scripts/node_modules.sh
  ];

  inputsFrom = [ fastlane.shell status-go-shell pod-shell ];

  shellHook = ''
    {
      cd "$STATUS_REACT_HOME"
      # Set up symlinks to mobile enviroment in project root
      ln -sf ./mobile/js_files/* ./
      # check if node modules changed and if so install them
      ./nix/scripts/node_modules.sh "${deps.nodejs-patched}"
    }
  '';
};
```

https://github.com/status-im/status-react/blob/develop/nix/mobile/ios/default.nix

Resources

In Repo

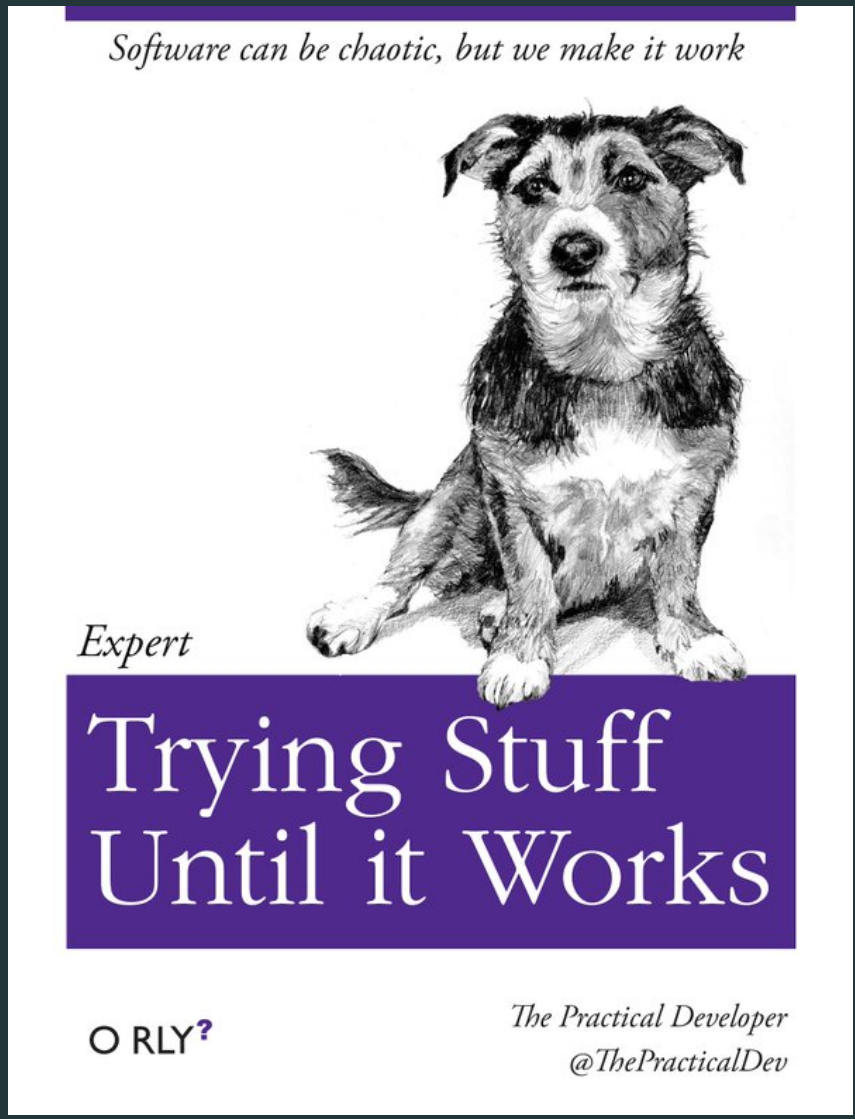
- `nix/README.md`
- `nix/KNOWN_ISSUES.md`
- `nix/DETAILS.md`

Previous Presentation

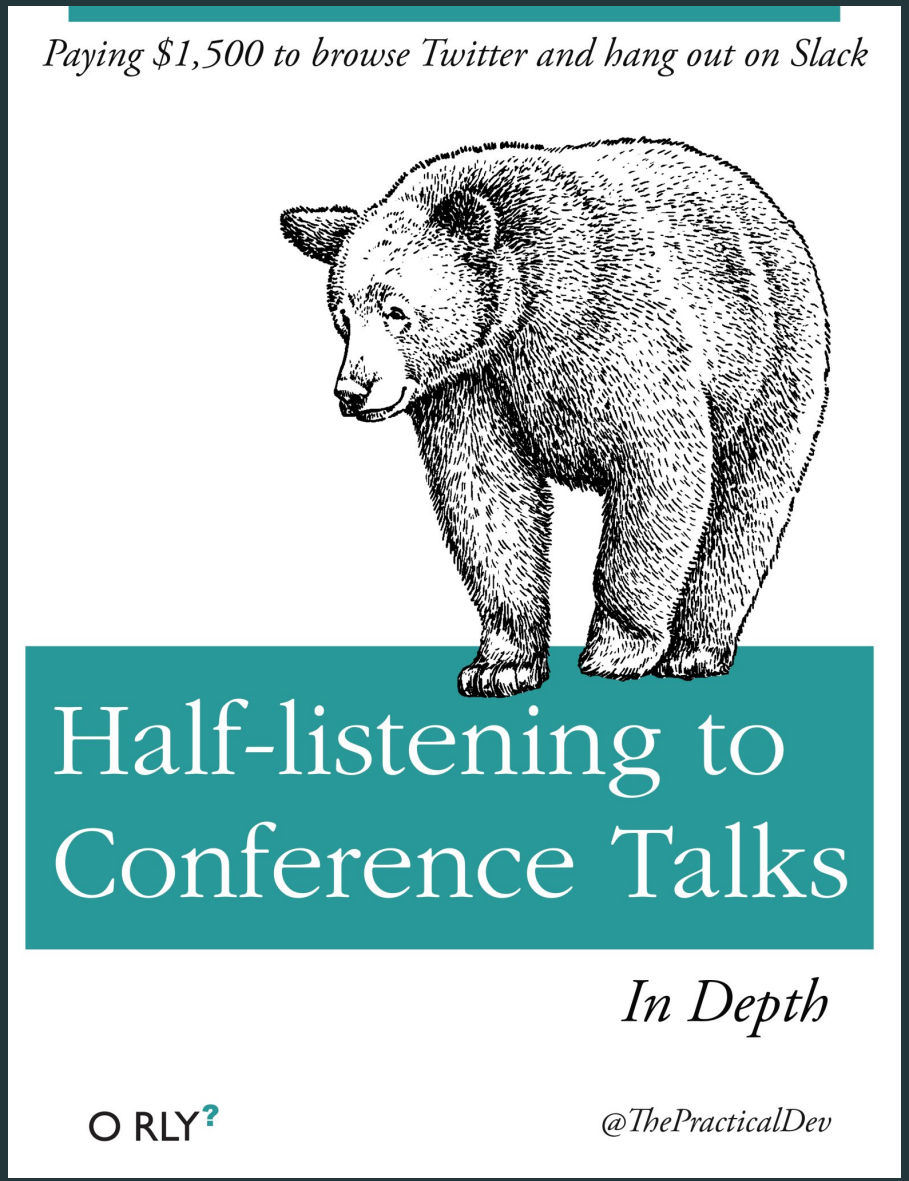
- Presentation
- PDF

New Cool Shit

Nix RFC #49 adds something called `flakes`, which is supposed to be a new way to centrally manage Nix builds for a repository.



Questions?


Paying $1,500 to browse Twitter and hang out on Slack
Half-listening to Conference Talks
In Depth
O RLY?
@ThePracticalDev