purerl-cookbook Documentation

Rob Ashton

Oct 06, 2021

Contents

1 The build process **3**
- 1.1 Purerl (Server) Build . . . 3
- 1.2 Erlang (Server) Build . . . 3
- 1.3 Purescript (Client) Build . . . 4

2 Editors **9**
- 2.1 Other editors . . . 9

3 Skeleton **15**
- 3.1 Erlang . . . 15
- 3.2 Purerl . . . 17
- 3.3 Purescript . . . 24

4 Basic OTP **27**
- 4.1 OTP Entry Point . . . 27
- 4.2 OTP Supervisor . . . 28
- 4.3 OTP Gen servers . . . 30
- 4.4 Dynamic Supervision Trees . . . 31

5 Web Server **35**
- 5.1 Stetson . . . 35
- 5.2 Stetson Routing . . . 36
- 5.3 Stetson Handlers . . . 38
- 5.4 Stetson Websockets . . . 39
- 5.5 Stetson Streaming . . . 40

6 Logging **43**
- 6.1 Lager . . . 43
- 6.2 Logger . . . 44

7 Messaging **47**
- 7.1 Subscribing to Incoming messages (pseudo-example) . . . 47
- 7.2 Sending Outgoing Messages (pseudo example) . . . 48

8 Interop **59**
- 8.1 FFI . . . 59
- 8.2 Effects . . . 60

8.3 Error handling 62
8.4 Messaging 64
8.5 Message Routing 65
8.6 Untagged Unions 66

9 Navigation **69**

This is a collection of code examples/explanations of most things you might want to do in the Purerl world, most the code examples are from, or related to the demo_ps project so having that cloned locally will help as you read through these docs.

If you feel anything is missing, please head over to Github and open an issue or pull request and we'll try and get it sorted.

- Pursuit - module/function search for or Purerl
- Pinto - an opinionated wrapper for OTP in Purescript
- Stetson - an opinionated wrapper for Cowboy in Purescript

CHAPTER 1

The build process

In an ideal world there would be a single tool for building, releasing, testing and performing package management for Purerl applications.

This is not the case, and we have tools for building Purescript and we have tools for building and releasing Erlang. To add onto this we're writing our clientside application in Purescript as well so that makes for three isolated build processes.

We *do* use a single tool for running all of the individual builds however, and that is rebar3 (the Erlang build tool), this will invoke both the client and server builds and then the Erlang.

It is easier mentally, to separate the Erlang and Purescript aspects of a Purerl application out and treat them as wholly independent, but this is not the case either - as Purescript modules can have (documented, but not automated) runtime dependencies on Erlang packages.

1.1 Purerl (Server) Build

There is a Makefile in the *'server'* folder in demo_ps, which will compile the '*.purs*' and dependencies into '*.erl*' in an '*output*' folder and then copy them into '*src/compiled_ps*'. This uses the tools of *spago*, *dhall*, *purs* and *purerl* for this task and the contents of '*packages.dhall*' and '*spago.dhall*' to determine how to do that.

Assuming this is succesful, then we are strictly back in Erlang world and using Erlang tools to handle these compiled files.

1.2 Erlang (Server) Build

The tool for building Erlang is *Rebar3*, which reads the contents of '*rebar.config*' to determine what dependencies to bring down and compile as well as anything found in the '*src*' directory (including our *compiled_ps*' from the Purerl world).

There are hooks in the '*rebar.config*' to invoke the Makefile found in '*server*' in order to perform the purescript build automatically before running the Erlang build.

1.3 Purescript (Client) Build

There is a Makefile in the *'client'* folder in demo_ps, which will compile the *.purs* and dependencies into *.js* in an 'output' folder and bundle them so the browser can understand them. This uses the tools of *spago*, *dhall* and *purs* for this task and the contents of *packages.dhall* and *spago.dhall* to determine how to do that.

The output of this will be copied into the 'priv' folder of the Erlang application so they can be served by the web server (Stetson).

Again, there are hooks in the *rebar.config* to invoke this Makefile, so the only command that actually needs executing for performing all of the above tasks is *rebar3 compile*.

1.3.1 Required Tools

There are quite a lot of dependencies and tools required to build a fully fledged Purerl application, it is for this purpose that the nix/docker scripts are used in the demo_ps project.

They can obviously be installed independently or by other automated means so for that purpose they are linked and detailed below for those wishing to embark on that journey.

Purescript

At the heart of it, Purescript is just a binary *'purs'* that knows how to compile Purescript into JS (its original purpose). Binary releases are provided for most platforms and this just needs to be in path.

Purerl

Purs (above) supports different backends, so with the right switches (in spago.dhall) we can use a different backend and compile into something else. In this case we're compiling the Purescript into Erlang.

Spago

Spago is a build tool used for both Purescript on the frontend and Purerl on the backend, it is used to download dependencies for the relevant application and also to configure the inputs to the purs compiler. That is which files it needs to compile, which backend to use for that compilation process amongst any other flags configureed.

Dhall

Dhall is a typed configuration language used for more than one thing, but specifically in this case it's used to describe the available dependencies for a Purescript project in the form of "package sets".

Erlang

Erlang is the original language that compiles into BEAM, which is what is executed by the Erlang runtime. It comes with a compiler (erlc) for this purpose, and various other tools that we don't need to know about here.

Rebar3

Rebar3 is a build tool for Erlang which reads the configuration for the project and pulls down dependencies and knows how to invoke the Erlang compiler on both those dependencies and the code written within the project. It also knows how to read various other assets in the project in order to package them up for release.

Purescript Language Server

The way to do add intelligence to your editor when working against Purescript

1.3.2 Docker

A docker file is provided in the demo-ps Purerl project that will allow for the building and execution of the project.

This is not by any means a "best practises" Docker development environment, I don't actually have a clue what a "best practises" Docker development environment would look like but I know this is not it (It's massive, for a start). This has been thrown together to make it easy to run the demo_ps project without having to manually install the tools required which has got to be a good thing.

This is definitely a good starting point for learning what tools are needed for a development environment and where to get them from (replace linux with macos in those download URLs and you're pretty good to go, binaries are available for most platforms across these projects)

Pull requests happily accepted if anybody wants to replace the docker workflow/files with something a little more appropriate.

```
FROM ubuntu:20.10

# Sigh
RUN apt update

# Erlang 22-3
RUN DEBIAN_FRONTEND="noninteractive" apt-get install -y git bash erlang curl build-
↪essential

RUN groupadd --gid 1000 dev \
  && useradd --uid 1000 --gid dev --shell /bin/bash --create-home dev

# Rebar3
RUN cd  /opt/ \
  && curl https://rebar3.s3.amazonaws.com/rebar3 > /usr/local/bin/rebar3 \
  && chmod +x /usr/local/bin/rebar3

# Purescript
RUN cd /opt/ \
    && curl -L https://github.com/purescript/purescript/releases/download/v0.14.4/
↪linux64.tar.gz > purescript.tar.gz \
    && tar -xvf purescript.tar.gz \
    && cp purescript/purs /usr/local/bin/purs  \
    && rm purescript.tar.gz

# Purerl
RUN cd /opt/ \
    && curl -L https://github.com/purerl/purerl/releases/download/v0.0.12/linux64.tar.
↪gz > purerl.tar.gz \
    && tar -xvf purerl.tar.gz \
```

(continues on next page)

(continued from previous page)

```
    && cp purerl/purerl /usr/local/bin/purerl  \
    && rm purerl.tar.gz

# Spago
RUN cd /opt/ \
    && curl -L https://github.com/purescript/spago/releases/download/0.20.3/linux.tar.
↪gz > spago.tar.gz \
    && tar -xvf spago.tar.gz \
    && cp spago /usr/local/bin/spago  \
    && rm spago.tar.gz

# Dhall
RUN cd /opt/ \
    && curl -L https://github.com/dhall-lang/dhall-haskell/releases/download/1.33.1/
↪dhall-1.33.1-x86_64-linux.tar.bz2 > dhall-json.tar.bz2 \
    && tar -xjvf dhall-json.tar.bz2 \
    && cp bin/dhall /usr/local/bin/dhall  \
    && rm dhall-json.tar.bz2
```

For convenience, the scripts *./build_docker_image.sh* and *./run_docker_image.sh* are provided, the project can be built therefore with

```
# Build the actual docker image
./build_docker_image.sh

# Compile the project
./run_docker_image.sh rebar3 compile

# Build a release
./run_docker_image.sh rebar3 release

# Run the whole shebang
./run
```

1.3.3 Nix

Nix is probably the easiest way to get started with Purerl (and stay up to date with releases).

Packages are provided in the following Github repos

- nixerl/nixpkgs-nixerl Erlang + Rebar3 Releases
- purerl/nixpkgs-purerl Purerl backend
- id3as/nixpkgs-purerl-support Purerl support packages

The core nix packages do contain tools like Purescript and Dhall, but these can laig a bit behind at times - the above repos when combined contain packages for everything sat at at up-to-date versions of those things (written as overlays on top of existing packages where possible).

Using them

An up to date and working shell.nix can be found in the demo_ps project and can usually be copied as-is. Combined with direnv, a sensible nix-shell can automatically provide a functional Purerl development environment as soon as you enter the directory for a project.

Essentially if the .envrc does

```
use_nix
```

And a shell.nix is provided next to this file, for example

```
let
  erlangReleases = builtins.fetchTarball https://github.com/nixerl/nixpkgs-nixerl/
↪archive/v1.0.18-devel.tar.gz;

  pinnedNixHash = "e5f945b13b3f6a39ec9fbb66c9794b277dc32aa1";
  pinnedNix =
    builtins.fetchGit {
      name = "nixpkgs-pinned";
      url = "https://github.com/NixOS/nixpkgs.git";
      rev = "${pinnedNixHash}";
    };

  purerlReleases =
    builtins.fetchGit {
      url = "https://github.com/purerl/nixpkgs-purerl.git";
      ref = "master";
      rev = "16582722c40f4c1a65c15f23e5f2438c6905981f";
    };

  purerlSupport =
    builtins.fetchGit {
      name = "purerl-support-packages";
      url = "git@github.com:id3as/nixpkgs-purerl-support.git";
      rev = "52926a56da6a8c526c403d26feaf52cc5f87a5d0";
    };

  nixpkgs =
    import pinnedNix {
      overlays = [
        (import erlangReleases)
        (import purerlReleases)
        (import purerlSupport)
      ];
    };

  erlangChannel = nixpkgs.nixerl.erlang-23-2-1.overrideScope' (self: super: {
    erlang = super.erlang.override {
      wxSupport = false;
    };
  });

  pls = nixpkgs.nodePackages.purescript-language-server.override {
    version = "0.15.4";
    src = builtins.fetchurl {
      url = "https://registry.npmjs.org/purescript-language-server/-/purescript-
↪language-server-0.15.4.tgz";
    };
  };

  pose = nixpkgs.nodePackages.purty.override {
      name = "prettier-plugin-purescript";
      packageName = "prettier-plugin-purescript";
```

(continues on next page)

(continued from previous page)

```
      version = "1.11.1";
      src = builtins.fetchurl {
        url = "https://registry.npmjs.org/@rowtype-yoga/prettier-plugin-purescript/-/
↪prettier-plugin-purescript-1.11.1.tgz";
      };
      meta = {
        description = "Hacked in Purescript Prettier Plugin";
        license = "MIT";
      };
    };
in

with nixpkgs;

mkShell {
  buildInputs = with pkgs; [

    erlangChannel.erlang
    erlangChannel.rebar3
    erlangChannel.erlang-ls

    # Purescript itself
    purerl-support.purescript-0-14-4
    purerl-support.spago-0-20-3
    purerl-support.dhall-json-1-5-0
    purerl-support.psa-0-8-2

    # Purerl backend for purescript
    purerl.purerl-0-0-12

    # The Language server for purescript
    pls

    # The current hotness for code formatting
    pose

   ];
}
```

Then allowing direnv to execute within the directory will provide all the tooling required for building the project once nix has downloaded and built the required assets, simples. Building and running the project from within this nix shell therefore looks like this:

```
# Compile the project
rebar3 compile

# Build a release
rebar3 release

# Run the whole shebang
./run
```

CHAPTER 2

Editors

One of the handy aspects of Purerl (as opposed to a 'new' language on top of BEAM) is that tooling already exists for working with Purescript and that tooling is also compatible with Purerl with no changes.

There are some nuances that we'll document here and add to if anything else comes up.

2.1 Other editors

If your editor is not listed, chances are you can get it working easily enough using a combination of psc-ide whatever LSP support that editor has, and the use of the purescript-language-server. It's certainly the best/lowest-friction place to start.

2.1.1 Vim/Neovim

Language server support is available for Purescript, therefore it is available for Purerl too.

Neo-vim 0.5

Neovim 0.5 comes with a built in language server protocol implementation - all that is left is configuring it for use.

Some extra plug-ins can be installed for then implementing additional functionality on top of that, such as auto-completion.

The list of plug-ins currently in use in my config is

- purescript-vim (*syntax highlighting*)
- neovim/nvim-lspconfig (*common configs for the built-in lsp*)
- nvim-lua/lsp_extensions.nvim (*extensions on top of the lsp*)

And then

- hrsh7th/nvim-cmp (*auto-completion engine for nvim*)

- hrsh7th/cmp-nvim-lsp (*lsp source for the auto-completion engine*)
- hrsh7th/cmp-vsnip (*etc etc etc*)
- hrsh7th/cmp-path
- hrsh7th/cmp-buffer
- hrsh7th/vim-vsnip

With the following setup in init.lua

```
local nvim_lsp = require 'lspconfig'

local on_attach = function(client, bufnr)
  local function buf_set_keymap(...) vim.api.nvim_buf_set_keymap(bufnr, ...) end
  local function buf_set_option(...) vim.api.nvim_buf_set_option(bufnr, ...) end

   -- Mappings.
  local opts = { noremap=true, silent=true }

  buf_set_keymap('n', 'gD', '<cmd>lua vim.lsp.buf.declaration()<CR>', opts)
  buf_set_keymap('n', 'gd', '<cmd>lua vim.lsp.buf.definition()<CR>', opts)
  buf_set_keymap('n', 'gr', '<cmd>lua vim.lsp.buf.references()<CR>', opts)
  buf_set_keymap('n', 'g[', '<cmd>lua vim.lsp.diagnostic.goto_prev()<CR>', opts)
  buf_set_keymap('n', 'g]', '<cmd>lua vim.lsp.diagnostic.goto_next()<CR>', opts)
  buf_set_keymap('n', 'ga', '<cmd>lua vim.lsp.buf.code_action()<CR>', opts)
  buf_set_keymap('n', 'gh', '<cmd>lua vim.lsp.buf.hover()<CR>', opts)
  buf_set_keymap('n', '<space>q', '<cmd>lua vim.lsp.diagnostic.set_loclist()<CR>',␣
↪opts)
  buf_set_keymap('n', '<space>i', '<cmd>lua vim.lsp.diagnostic.set_loclist()<CR>',␣
↪opts)
  buf_set_keymap('n', '<space>f', '<cmd>lua vim.lsp.buf.formatting()<CR>', opts)
end

-- Configure Purescript
nvim_lsp['purescriptls'].setup {
  on_attach = on_attach,
  settings = {
    purescript = {
      formatter = "pose",
      codegenTargets = { "corefn" },
      addSpagoSources = true,
    },
  },
  flags = {
    debounce_text_changes = 150,
  }
}

-- Disable the annoying LSP virtual text
vim.lsp.handlers["textDocument/publishDiagnostics"] = vim.lsp.with(
  vim.lsp.diagnostic.on_publish_diagnostics, {
    virtual_text = false,
    underline = true,
    signs = true,
  }
)

-- Setup the cmp plugin for auto completion
```

(continues on next page)

(continued from previous page)

```
local cmp = require 'cmp'
cmp.setup({
  snippet = {
    expand = function(args)
        vim.fn["vsnip#anonymous"](args.body)
    end,
  },
  mapping = {
    ['<C-p>'] = cmp.mapping.select_prev_item(),
    ['<C-n>'] = cmp.mapping.select_next_item(),
    -- Add tab support
    ['<S-Tab>'] = cmp.mapping.select_prev_item(),
    ['<Tab>'] = cmp.mapping.select_next_item(),
    ['<C-d>'] = cmp.mapping.scroll_docs(-4),
    ['<C-f>'] = cmp.mapping.scroll_docs(4),
    ['<C-Space>'] = cmp.mapping.complete(),
    ['<C-e>'] = cmp.mapping.close(),
    ['<CR>'] = cmp.mapping.confirm({
      behavior = cmp.ConfirmBehavior.Insert,
      select = true,
    })
  },

  -- Installed sources for 'cmp'
  sources = {
    { name = 'nvim_lsp' },
    { name = 'vsnip' },
    { name = 'path' },
    { name = 'buffer' },
  },
})
```

additionally in init.vim

```
" Set completeopt to have a better completion experience
set completeopt=menuone,noinsert,noselect

" Avoid showing message extra message when using completion
set shortmess+=c

" Reserve space for the errors
set signcolumn=yes
```

With vim-coc

Add this to the config, using :CocConfig

```
"languageserver": {
  "purescript": {
    "command": "purescript-language-server",
    "args": ["--stdio"],
    "filetypes": ["purescript"],
    "rootPatterns": ["bower.json", "psc-package.json", "spago.dhall"],
    "settings": {
      "purescript": {
```

(continues on next page)

(continued from previous page)

```
        "addSpagoSources": true
      }
    }
  }
}
```

With vim-lsp

Note: This might be out of date, as the author hasn't used vim-lsp in over a year.

What we need is

- vim-lsp: An arbitrarily chosen LSP plugin for VIM
- purescript-language-server: The language server
- vim-purescript: Syntax highlighting (still)

The bare minimum config for getting this up and running is

```
if executable('purescript-language-server')
    au User lsp_setup call lsp#register_server({
      \ 'name': 'purescript-language-server',
      \ 'cmd': {server_info-> ['purescript-language-server', '--stdio']},
      \ 'allowlist': ['purescript']
      \ })
endif
```

But it's a bit better if you at least set the rootUri based on the manifest location, as that's rarely going to be the root of the Git repo in a Purerl project.

```
if executable('purescript-language-server')
    au User lsp_setup call lsp#register_server({
      \ 'name': 'purescript-language-server',
      \ 'cmd': {server_info-> ['purescript-language-server', '--stdio']},
      \ 'root_uri':{server_info->
      \ lsp#utils#path_to_uri(
      \       lsp#utils#find_nearest_parent_file_directory(
      \               lsp#utils#get_buffer_path(), ['spago.dhall']
      \       ))},
      \ 'allowlist': ['purescript']
      \ })
endif
```

Obviously it can then be configured further, and extra keybindings can be added when a buffer is opened in this mode

```
function! s:on_lsp_buffer_enabled() abort
    setlocal omnifunc=lsp#complete
    setlocal signcolumn=yes
    if exists('+tagfunc') | setlocal tagfunc=lsp#tagfunc | endif
endfunction

augroup lsp_install
    au!
    autocmd User lsp_buffer_enabled call s:on_lsp_buffer_enabled()
augroup END
```

This is quite a basic setup, config can be passed to the language server to make it more aware of spago/psc-package/etc, all of that is documented in the relevant projects.

The functionality is *rich* compared to the plain psc-ide experience, and is more fully documented on the vim-lsp github page.

In this default state, the editor will need restarting between editing client/server projects, with the use of local config this could probably be obliviated (separate ports for the language server, etc)

Code updates should generally be reflected much more responsively, so this makes for a much smoother experience than the direct psc-ide integration.

Without Language Server

Without the LSP, support for Purescript/Purerl can be gained by the installation of two plugins

- vim-psc-ide: Integration to 'purs ide'
- vim-purescript: Syntax highlighting

Functionality gained

- syntax highlighting
- purs ide started in background automatically
- compilation on file-save
- module import checking
- auto module imports
- function type checking

Caveats

- In the default state, :Pload will need to be ran a lot, or the purs ide will be out of sync with module changes
- Switching between client-side code and server-side code will mean an editor restart (multiple projects, two servers needed)

2.1.2 VSCode

VSCode is probably the simplest IDE to get up and running with, as there is simply a couple of extensions installable by the built-in extension manager.

- purescript-language-server: The language server
- vscode-language-purescript
- vscode-ide-purescript

The first vscode extension *should* be automatically installed by the secone one so is only there for completeness. The above being a complete package in a single place also means little documentation is required here cos it exists over there..

For the demo project, a spago setup is useful; That just means setting the following values in your *settings.json*

```
{
  "purescript.codegenTargets": [ "corefn" ],
  "purescript.addSpagoSources": true,
  "purescript.buildCommand": "spago build --purs-args --json-errors"
}
```

To get this to work effectively, you'll want to open the *server* and *client* code as 'folders' separately.

2.1.3 Emacs

Obviously the short answer to this page is use *vim* but if you're already set in your ways then you can carry on reading this page.

The majority of our team are using the language server protocol, although PSC-IDE does exist.

With Language Server

The purescript-language-server needs installing somewhere, and the following packages need to be present in emacs (again, all available in Melpa).

- lsp-mode
- dhall-mode

lsp-mode has support for Purescript built in, and just needs spinning up for purescript-mode

```
(add-hook 'purescript-mode-hook #'lsp)
```

Further docs for this are worth reading, I'm not an Emacs user so YMMV.

Feel free to send me a pull request for this page if you have a good Emacs set up based on either of the above, as I find both of these default setups to be distinctly lacking and don't know enough about Emacs to fix it.

Example configs

- Steve Strong

CHAPTER 3

Skeleton

As mentioned in the *build env* section, setting up a Purerl application is unfortunately a bit of a dance involving the merging of several worlds, the Erlang world, Rebar3/Relx and Purescript itself along with its build toools.

This can be quite daunting, Erlang already had quite a bit of overhead for a basic 'hello world' demo and adding the complexity of additional tooling and configuration to support Purescript can make it quite even tougher. If you're approaching Purerl from the point of view of either an Erlang programmer or a Purescript programmer than at least half of this world will be familiar to you, if you're coming at it fresh then you have my sympathies.

The easy solution for the most part is to copy an empty-ish application (like demo-ps and delete the bits you don't want. In time you'll learn what all the moving parts are when you need to make changes. The other solution is to just spend a bit of time reading, making notes and mapping what you read onto the code that exists in the demo project.

For the purposes of this section we'll separate these various aspects into

- *Erlang*: The various files needed to support a plain ol' Erlang Application
- *Purerl*: The various files needed to support a server-side Purescript application
- *Purescript*: The various files needed to support a client-side Purescript application

3.1 Erlang

The Erlang stack can seem quite involved, in the context of the demo_ps project, we've got

3.1.1 rebar.config / rebar.lock

This file contains the *Erlang* dependencies that need downloading and building as well as information about assets that need deploying along with any release (using *rebar3 release*)

In the case of demo_ps, those dependencies are fairly minimal

```
{deps,[
       {jsx, "2.8.0"},
       {recon, "2.3.6"},
       {eredis, {git, "https://github.com/wooga/eredis.git", {tag, "v1.2.0"}}},
       {erlydtl, {git, "https://github.com/erlydtl/erlydtl", {tag, "0.12.1"}}},
       {cowboy, { git, "https://github.com/ninenines/cowboy.git", { tag, "2.6.0" }}},
       {gproc, {git, "https://github.com/uwiger/gproc.git", {tag, "0.8.0"}}},
       {erlang_ls, { git, "https://github.com/erlang-ls/erlang_ls.git", {tag, "0.4.1"}
↪}}
```

- JSX is required because of the Purerl simple-json dependency
- Cowboy is required because of the Purerl Stetson dependency

These are not added automatically but are merely documented as dependencies in those project README.mds. This is something to be aware of when bringing down Purerl dependencies.

The other dependencies are there to support code that we write in Purerl ourselves (covered later in this guide).

We'll ignore the release process for now, but we'll note that at the end of thise file we're specifying the Makefiles that need executing to build the server and client apps in Purescript.

```
]}.
{pre_hooks,
  [
   {"(linux|darwin|solaris|win32)", compile, "bash -c 'make'"},
   {"(linux|darwin|solaris|win32)", clean, "make clean"}
```

3.1.2 src/demo_ps.app.src

This file is present to tell the runtime about the application, the most important aspect of this being the name of the module that needs executing.

```
  {mod, { bookApp@ps, []}},
```

This file *also* tells the runtime which applications need starting before this one, and that means most of the dependencies in rebar.config are duplicated in here. Welcome to Erlang.

```
  {applications,
   [kernel,
    stdlib,
    gproc,
    recon,
    sasl,
    cowboy,
    jsx,
    eredis
```

3.1.3 src/*

Any erlang found in this src folder will be compiled when running rebar3, this gives us the ability to write native Erlang that exists independently of the Purerl application.

3.1.4 src/compiled_ps

This is where the output of the Purerl code gets copied to, and because it happens to be beneath the src directory it then gets compiled to beam.

3.1.5 release-files/sys.config

This is a convention, but configuration is usually placed in this file.

3.1.6 priv/*

A special folder containing assets that'll be deployed with a release. Amongst other things, this is where we'd usually find the static portions of the website that a web application consists of (a index.html, some css, etc). We'll copy the bundled JS generated by the Purescript on the client side into here as well.

3.2 Purerl

The Purerl code is relatively easy to follow coming from any sort of JS environment, in essence it boils down to a single folder of code with a manifest describing the package and its dependencies. The compiler will take all of the Purescript and compile both it and the modules to the output directory and then it's up to us to copy that to somewhere the Erlang compiler can find so it can be further compiled into the beam format.

It is convenient sometimes to share some code between the client and server, and in the demo_ps project this is located in a '*./shared*' folder that is symlinked from '*./server/shared*'.

3.2.1 server/packages.dhall

This file contains a reference to a package set that is maintained by the Purerl Organisation. A package set is a collection of packages that (hopefully) work nicely together as well as a description of the dependencies between them.

```
let upstream =
      https://github.com/purerl/package-sets/releases/download/erl-0.14.3-20210709/
↪packages.dhall␣
↪sha256:9b07e1fe89050620e2ad7f7623d409f19b5e571f43c2bdb61242377f7b89d941
```

This is followed (presently) by some overrides of packages that exist within the package set, but of which we want later versions of (because we like to live life on the edge)

```

let overrides =
  { erl-simplebus = upstream.erl-simplebus //
    { version = "52d374a8a7a0bb13db6a8ac6552c55e4e2da7d9f"
    }
  , erl-test-eunit = upstream.erl-test-eunit //
    { version = "ed31f51d19f1faba764c32bbad00fa7e4ef752d2"
    }
  , exceptions = upstream.exceptions //
    { version = "edef0014db73aa3136dd4ab2290becb29f6fe6c6"
    , repo = "https://github.com/robashton/purescript-exceptions/"
    }
```

(continues on next page)

(continued from previous page)

```
  }
let additions =
      { erl-opentelemetry =
        { dependencies = [ "effect", "erl-lists", "erl-tuples" ]
        , repo = "https://github.com/id3as/purescript-erl-opentelemetry.git"
        , version = "f8842104a08e4d455084778f5e120347f4a28bd4"
        }
       , erl-binary =
        { dependencies = [ "prelude", "erl-lists" ]
        , repo = "https://github.com/id3as/purescript-erl-binary.git"
        , version = "423f1af8437670beab03463b3e9bc0a487f05ba4"
        }
      , erl-kernel =
        { dependencies =
          [ "convertable-options"
          , "datetime"
          , "effect"
          , "either"
          , "erl-atom"
          , "erl-binary"
          , "erl-lists"
          , "erl-process"
          , "erl-tuples"
          , "erl-untagged-union"
          , "foldable-traversable"
          , "foreign"
          , "functions"
          , "integers"
          , "maybe"
          , "newtype"
          , "partial"
          , "prelude"
          , "record"
          , "typelevel-prelude"
          , "unsafe-coerce"
          ]
        , repo = "https://github.com/id3as/purescript-erl-kernel.git"
        , version = "2c1f78a3aa6993e91e342a984c522b87b98bbb2b"
        }
      , erl-gun =
        { dependencies =
          [ "convertable-options"
          , "datetime"
          , "effect"
          , "either"
          , "erl-atom"
          , "erl-binary"
          , "erl-kernel"
          , "erl-lists"
          , "erl-maps"
          , "erl-process"
          , "erl-ssl"
          , "erl-tuples"
          , "erl-untagged-union"
          , "foreign"
          , "functions"
          , "maybe"
```

(continues on next page)

(continued from previous page)

```
          , "prelude"
          , "record"
          , "simple-json"
          , "typelevel-prelude"
          ]
        , repo = "https://github.com/id3as/purescript-erl-gun.git"
        , version = "c25358f9bae80b9a2512a46f91f51438a7f621fc"
        }
      , erl-otp-types =
        { dependencies =
          [ "erl-atom"
          , "erl-binary"
          , "erl-kernel"
          , "foreign"
          , "prelude"
          , "unsafe-reference"
          ]
        , repo = "https://github.com/id3as/purescript-erl-otp-types.git"
        , version = "6470bc379447c406456e8ef1e6a79c80e3c5e8d1"
        }
      , erl-ranch =
        { dependencies =
          [ "convertable-options"
          , "effect"
          , "either"
          , "erl-atom"
          , "erl-kernel"
          , "erl-lists"
          , "erl-maps"
          , "erl-otp-types"
          , "erl-process"
          , "erl-ssl"
          , "erl-tuples"
          , "exceptions"
          , "foreign"
          , "maybe"
          , "prelude"
          , "record"
          , "typelevel-prelude"
          , "unsafe-coerce"
          ]
        , repo = "https://github.com/id3as/purescript-erl-ranch.git"
        , version = "08a76bd850ba00c3a120c1d149bed07f9fcc165d"
        }
      , erl-ssl =
        { dependencies =
          [ "convertable-options"
          , "datetime"
          , "effect"
          , "either"
          , "maybe"
          , "erl-atom"
          , "erl-binary"
          , "erl-lists"
          , "erl-kernel"
          , "erl-tuples"
          , "erl-logger"
```

(continues on next page)

(continued from previous page)

```
          , "erl-otp-types"
          , "foreign"
          , "maybe"
          , "partial"
          , "prelude"
          , "record"
          , "unsafe-reference"
          ]
        , repo = "https://github.com/id3as/purescript-erl-ssl.git"
        , version = "2bd94ce343448406e579425e1b4140a6b6dd7de0"
        }
      , datetime-parsing =
        { dependencies =
                [  "arrays"
                 , "datetime"
                 , "either"
                 , "enums"
                 , "foldable-traversable"
                 , "integers"
                 , "lists"
                 , "maybe"
                 , "numbers"
                 , "parsing"
                 , "prelude"
                 , "psci-support"
                 , "strings"
                ]
        , repo = "https://github.com/flounders/purescript-datetime-parsing"
        , version = "10c0a9aecc60a2a5e8cff35bebe45be4dacaa7f8"
        }
      , sequences =
        { dependencies =
          [ "prelude"
          , "unsafe-coerce"
          , "partial"
          , "unfoldable"
          , "lazy"
          , "arrays"
          , "profunctor"
          , "maybe"
          , "tuples"
          , "newtype"
          ]
        , repo = "https://github.com/id3as/purescript-sequences.git"
        , version = "73fdb04afa32be8a3e3d1d37203592118d4307bc"
        }
      , convertable-options =
        { repo = "https://github.com/natefaubion/purescript-convertable-options"
        , dependencies = [ "effect", "maybe", "record" ]
        , version = "f20235d464e8767c469c3804cf6bec4501f970e6"
        }
      , erl-cowboy =
        { repo = "https://github.com/purerl/purescript-erl-cowboy.git"
        , dependencies =
          [ "console"
          , "effect"
          , "either"
```

(continues on next page)

(continued from previous page)

```
          , "erl-atom"
          , "erl-binary"
          , "erl-kernel"
          , "erl-lists"
          , "erl-maps"
          , "erl-modules"
          , "erl-ranch"
          , "erl-tuples"
          , "foreign"
          , "functions"
          , "maybe"
          , "prelude"
          , "transformers"
          , "tuples"
          , "unsafe-coerce"
          ]
        , version = "ca4dd4a6432817fbe6ef9ab1814046f6827046cd"
        }
      , unsafe-reference =
        { repo = "https://github.com/purerl/purescript-unsafe-reference.git"
        , dependencies = [ "prelude"  ]
        , version = "464ee74d0c3ef50e7b661c13399697431f4b6251"
        }
      , erl-stetson =
        { repo = "https://github.com/id3as/purescript-erl-stetson.git"
        , dependencies =
          [ "erl-atom"
          , "erl-binary"
          , "erl-lists"
          , "erl-maps"
          , "erl-tuples"
          , "erl-modules"
          , "erl-cowboy"
          , "foreign"
          , "maybe"
          , "prelude"
          , "transformers"
          , "routing-duplex"
          ]
        , version = "a0c0bb4b5ad9046dd69c77197dc5dd025883ada2"
        }
      , erl-untagged-union =
        { dependencies =
          [ "erl-atom"
          , "erl-binary"
          , "erl-lists"
          , "erl-tuples"
          , "debug"
          , "foreign"
          , "typelevel-prelude"
          , "maybe"
          , "partial"
          , "prelude"
          , "unsafe-coerce"
          ]
        , repo = "https://github.com/id3as/purescript-erl-untagged-union.git"
        , version = "eb7a10c7930c4b99f1a6bfce767daa814d45dd2b"
```

(continues on next page)

(continued from previous page)

```
    }
  , rationals =
    { repo = "https://github.com/anttih/purescript-rationals.git"
    , dependencies =
      [ "prelude"
      ]
    , version = "c883c972513380ae161d816ed42108acfe8cc8f6"
    }
  , erl-process =
    { repo = "https://github.com/id3as/purescript-erl-process.git"
    , dependencies =
      [ "console"
      , "prelude"
      , "effect"
      ]
    , version = "afbfa4e7a13c0d55609ff144d49982563fada7f5"
    }
  , erl-pinto =
    { repo = "https://github.com/id3as/purescript-erl-pinto.git"
    , dependencies =
      [ "erl-process"
      , "erl-lists"
      , "erl-atom"
      , "erl-kernel"
      , "datetime"
      , "erl-tuples"
      , "erl-modules"
      , "foreign"
      ]
    , version = "598587428b7b6711412312596f3825ea88d471d2"
    }
  , erl-nativerefs =
    { repo = "https://github.com/id3as/purescript-erl-nativerefs.git"
    , dependencies =
      [ "prelude"
      , "effect"
      , "erl-tuples"
      ]
    , version = "b90469380821615adf4cb58782ff246f79eec961"
    }
  , these =
    { repo = "https://github.com/purescript-contrib/purescript-these"
    , version = "a98d0a4e80c9a75fa359e1bcabb0230fb99c52fa"
    , dependencies =
      [ "prelude"
      , "quickcheck"
      , "quickcheck-laws"
      ]

    }
  , uri =
    { repo = "https://github.com/purescript-contrib/purescript-uri"
    , version = "041e717f0c5c663b787b532a00ac706e7897e932"
    , dependencies =
      [ "prelude"
      ]
    }
```

(continues on next page)

(continued from previous page)

```
      , quickcheck =
        { repo = "https://github.com/id3as/purescript-quickcheck"
        , version = "694a781d4f441cfb264b9efa368a0441532133af"
        , dependencies =
          [ "prelude"
          , "lcg"
          ]
        }
      , quickcheck-laws =
        { repo = "https://github.com/purescript-contrib/purescript-quickcheck-laws"
        , version = "464597522e5e001adc2619676584871f423b9ea0"
        , dependencies =
          [ "prelude"
          ]
        }
      }

in  upstream // overrides // additions
```

This gives us an amount of flexibility to work on both a stable set of packages as well as actively developed/updated packages that we might ourselves be maintaining.

3.2.2 server/spago.dhall

Having set up the package set we want to refer to, we *then* define both the packages we're interested in from that package set and where to look for the code files that we are going to write alongside all of this. We also specify that our backend is going to be 'purerl' because compiling all of this code into Javascript isn't going to do us very much good.

```
{-
-}
{ name = "demo"
, dependencies =
  [ "console"
  , "control"
  , "datetime"
  , "effect"
  , "either"
  , "erl-atom"
  , "erl-binary"
  , "erl-cowboy"
  , "erl-lists"
  , "erl-logger"
  , "erl-maps"
  , "erl-modules"
  , "erl-pinto"
  , "erl-process"
  , "erl-simplebus"
  , "erl-stetson"
  , "erl-tuples"
  , "filterable"
  , "foldable-traversable"
  , "foreign"
  , "maybe"
  , "newtype"
```

(continues on next page)

(continued from previous page)

```
  , "partial"
  , "prelude"
  , "psci-support"
  , "record"
  , "routing-duplex"
  , "simple-json"
  , "transformers"
  , "typelevel-prelude"
  , "unsafe-coerce"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
, backend = "purerl"
}
```

3.2.3 server/Makefile

Make is relatively well understood so while it's not strictly necessary to have in in this project it's nice to set up the build to be dependent on the files in the project so we don't build unnecessarily. We could of course just invoke *spago build* from the '*./rebar.config*' in top level and forgo this step.

3.2.4 server/src/*

All of the '*.purs*' found within here (and nested directories) will be built by *spago build* and '*.erl*' files will be produced the '*output*' directory corresponding to those files.

These then get copied into '*./src/compiled_ps*' for compilation by the standard Erlang workflow.

3.3 Purescript

For the most part, the Purescript code is entirely self contained, compiling into a single JS bundle that can be served by the web server running in Purerl.

That said, in a Purerl application is it convenient to share *some* code with the Purerl as well (view models and such), and this is why the '*./shared*' folder in the root of the project is symlinked to the '*./client/shared*' directory.

3.3.1 client/packages.dhall

This file contains a reference to a package set that is maintained by the Purescript Organisation. A package set is a collection of packages that (hopefully) work nicely together as well as a description of the dependencies between them.

```
let upstream =
      https://github.com/purescript/package-sets/releases/download/psc-0.14.0-
↪20210329/packages.dhall␣
↪sha256:32c90bbcd8c1018126be586097f05266b391f6aea9125cf10fba2292cb2b8c73

let overrides = {=}

let additions = {=}
```

(continues on next page)

(continued from previous page)

```
7
8 in  upstream // overrides // additions
```

In the case of demo_ps we're happy with this default package set so don't add anything to it or override anything.

3.3.2 server/spago.dhall

Having set up the package set we want to refer to, we *then* define both the packages we're interested in from that package set and where to look for the code files that we are going to write alongside all of this. We use the default backend to generate Javascript.

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "client"
, dependencies =
  [ "aff"
  , "debug"
  , "affjax"
  , "arrays"
  , "bifunctors"
  , "control"
  , "effect"
  , "either"
  , "foreign"
  , "functions"
  , "halogen"
  , "halogen-bootstrap4"
  , "http-methods"
  , "maybe"
  , "media-types"
  , "newtype"
  , "ordered-collections"
  , "prelude"
  , "psci-support"
  , "record"
  , "routing"
  , "routing-duplex"
  , "simple-json"
  , "transformers"
  , "tuples"
  , "typelevel-prelude"
  , "unsafe-coerce"
  , "web-events"
  , "web-uievents"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

3.3.3 server/Makefile

Make is relatively well understood so while it's not strictly necessary to have in in this project it's nice to set up the build to be dependent on the files in the project so we don't build unnecessarily. We could of course just invoke *spago bundle* from the '*./rebar.config*' in top level and forgo this step.

3.3.4 client/src/*

All of the '*.purs*' found within here (and nested directories) will be built by *spago build* and '*.js*' files will be produced the '*output*' directory corresponding to those files.

spago bundle produces a single bundle.js out of these which we can include from HTML to get our client side application up and running. That index.html happens to be found in *priv/www/index.html* and it is next to this file to which this bundle.js gets copied as part of build.

CHAPTER 4

Basic OTP

As we keep uncovering, the basic hello world of an Erlang application isn't as simple or Hackernews friendly as a single file with a http web server in it (Elixir has managed this to an extent, but typically it just pushes the learning curve to the right and you still have to go up it eventually).

In order to get something running, we need to

- Write an application entry point
- Tell Erlang where to find that entry point
- Write a supervision tree
- Write some children to go under that tree

Thankfully we can do *almost* all of this in Purescript (telling Erlang where to find the entry point is still writing Erlang, we don't make new applications very often at work so optimising for 'new projects' isn't something we've really focused on thus far).

- *Application Entry point*
- *Supervisor*
- *Basic gen server*
- *Dynamic Supervision Trees*

4.1 OTP Entry Point

As part of configuration for an Erlang application, an *.app.src* is provided which gets compiled into a *.app* and is then used to determine how to start an application and what need starting before that occurs (amongst various other settings).

There is no Purerl wrapper around this because it has (so far) been seen as a low value thing to abstract, the demo_ps project therefore has a file directly written at '*./src/demo_ps.app.src*' which is entirely Erlang specific and dealt with when we run *rebar3 compile*

```
{application, demo_ps,
 [{description, "An OTP application"},
  {vsn, "0.1.0"},
  {registered, []},
  {mod, { bookApp@ps, []}},
  {applications,
   [kernel,
    stdlib,
    gproc,
    recon,
    sasl,
    cowboy,
    jsx,
    eredis
   ]},
  {env,[]},
  {modules, []},
  {maintainers, []},
  {licenses, []},
  {links, []}
 ]}.
```

The most important part of this file as far as we're concerned here is the line that specifies which module is the entry point to our application. In this case that's *bookApp@ps*. This corresponds to the file '*./server/src/BookApp.Purs*' and this instantly tells us something.

When we compile the *.purs*, the module created is camelCased and the suffix '@ps' is added to it. (@ is a valid character in module names that nobody seems to use so we're unlikely to get collisions).

What does this module look like?

```
module BookApp where

import BookSup as BookSup
import Pinto.App as App

-- Our entry point is not tremendously exciting
-- but it is an entry point
start = App.simpleStart BookSup.startLink
```

Almost inconsequential. In our production Erlang apps we might sometimes do a bit of housekeeping as part of start-up, but the most basic application is essentially a proxy for starting the top level supervisor, so Pinto provides *App.simpleStart* for that case.

4.2 OTP Supervisor

Typically, the application in the entry point will kick off a single supervision tree that will contain other supervision trees as well as various worker processes (typically gen servers but absolutely not strictly so).

A supervisor has a name (because we might want to interact with it programatically once it has started), and typically exposes a function *startLink* that will call *Supervisor.StartLink* passing in a callback to an 'init' function that'll be invoked by OTP within the context of supervision process in order to get the children that need starting as part of this supervision tree.

```
startLink :: Effect (StartLinkResult SupervisorPid)
startLink = do
  Sup.startLink (Just $ Local $ atom "example_usp") init
```

At the heart of it, a supervision tree is just a list of children that will be spun up by the supervisor, and instructions (flags) on what to do when those children crash. The names of everything in this specification map onto the names in the underlying erlang API so for the most part no explicit documentation is required from it.

```
init :: Effect SupervisorSpec
init = do
  connectionString <- BookConfig.connectionString
  webPort <- BookConfig.webPort
  -- Unsurprisingly, this looks just like the specs in the Erlang docs
  -- this is intentional
  -- the difference being that the args for the startLinks are passed in to create␣
↪Effect Units
  -- which means they're typed
  pure
    { flags:
        { strategy: OneForOne
        , intensity: 1
        , period: Seconds 5.0
        }
    , childSpecs:
        (worker "book_web" $ BookWeb.startLink { webPort })
        : (worker "empty_server" $ EmptyGenServer.startLink {})
        : (worker "book_library" $ BookLibrary.startLink { connectionString })
        : (worker "handle_info_example" $ HandleInfoExample.startLink {})
        : (worker "monitor_example" $ MonitorExample.startLink {})
        : (worker "one_for_one_example" $ OneForOneSup.startLink)
        : nil
    }
```

That worker function just contains the defaults for our specific supervisor:

```
worker ::
  forall childProcess.
  HasPid childProcess =>
  String -> Effect (StartLinkResult childProcess) -> ErlChildSpec
worker id start =
  spec
    { id
    , childType: Worker
    , start
    , restartStrategy: RestartTransient
    , shutdownStrategy: ShutdownTimeout $ Milliseconds 5000.0
    }
```

So we can see in this code we simply return the flags for this tree and a list of children that need starting. Each of those children is just an (Effect childPid) and with this mechanism, it means that the arguments for each child are type checked (unlike in straight Erlang).

4.3 OTP Gen servers

The workhorse of the OTP world, it's no surprise that the API for Pinto.GenServer is one of the most involved of the APIs shown at this point.

The most basic Genserver is just a startLink taking in some arguments, calling the default *GenServer.startLink* with a 'spec' containing the init callback and optional callbacks for things like handle_info/terminate/etc.

That init callback will be invoked by OTP and is responsible for setting up the initial state of the gen server (or failing).

```
serverName :: RegistryName (ServerType Unit Unit Unit State)
serverName = Local $ atom "empty_gen_server"

startLink :: EmptyGenServerStartArgs -> Effect (StartLinkResult (ServerPid Unit Unit Unit State))
startLink args = GenServer.startLink $ (GenServer.defaultSpec $ init args) { name = Just serverName }

init :: EmptyGenServerStartArgs -> GenServer.InitFn Unit Unit Unit State
init _args = do
  pure $ InitOk {}
```

Now this is a bit of a useless example, our start args are a record with no fields, our state is a record with no fields and there are no operations defined over this empty state.

There is a lot of 'Unit' in this empty code, GenServer.ServerType allows us to define the type of our 'continue', 'stop' and 'info' message types, as well as our state. Unit is the default for 'we're not using those things' and you would interchange for your own data types as you required additional functionality.

Let's define a gen server that has some start args (an initial value), and a state (that value). How do we *do things* to that state once the process is started?

```
type CounterExampleStartArgs
  = { initialValue :: Int }

type State
  = { value :: Int }

serverName :: RegistryName (ServerType Unit Unit Unit State)
serverName = Local $ atom "counter_example"

startLink :: CounterExampleStartArgs -> Effect (StartLinkResult (ServerPid Unit Unit Unit State))
startLink args = GenServer.startLink $ (GenServer.defaultSpec $ init args) { name = Just serverName }

init :: CounterExampleStartArgs -> GenServer.InitFn Unit Unit Unit State
init args = do
  pure $ InitOk { value: args.initialValue }
```

In Erlang there are two ways you would typically talk to a gen server specifially, gen_server:cast (send a message and don't wait for a response) and gen_server:call (send a message and get something back).

This is quite a verbose process in Erlang as the message-send and the message-receive are written independently of each other despite often being identical. This can be useful when you're throwing versioned tuples around on a wing and a prayer but unless you're in the minority of circumstances where you're doing this to help with hotloading/upgrades it's quite a long winded way of 'calling a function within the context of my running gen server'.

In Pinto.GenServer this is represented instead as a simple callback that can be expressed inline at the callsite.

```
current :: Effect Int
current = GenServer.call (ByName serverName) (\_f s -> pure $ GenServer.reply s.value s)
```

```
add :: Int -> Effect Unit
add a = GenServer.cast (ByName serverName) (\s@{ value } -> pure $ GenServer.return s { value = value + a })
```

The return results and function signatures still map fairly cleanly onto the Erlang API so the documentation for Pinto and OTP don't need to diverge too much.

4.3.1 The monad

When operating inside a gen server context, we're actually inside a ReaderT with a whole pile of phantom types enforcing the various messages that a gen server can receive/return. This doesn't need to be thought about in too much detail unless you're sending a pull request to Pinto itself, but in essence this means that a few things need to be beared in mind when writing code.

In order to invoke an effect inside a gen server, it will need to be lifted into the Reader monad with *liftEffect*

```
import Effect.Class (liftEffect)

current :: Effect Int
current = GenServer.call (ByName serverName) (\_from s ->  do
  liftEffect $ SomeApi.doSomethingCool
  pure $ GenServer.reply s.value s)
```

On the bright side, being sat in this monad means getting hold of 'self' as a *Process Msg* is as simple as calling 'self' from the 'HasSelf' typeclass in Erl.Process.

```
import Erl.Process (self)

current :: Effect Int
current = GenServer.call (ByName serverName) (\_from s ->  do
  me <- self
  liftEffect $ SomeApi.playWithMe me
  pure $ GenServer.reply s.value s)
```

4.3.2 The callbacks

As mentioned, our call to startLink can optionally set up various callbacks. These are largely self explanatory if you already are familiar with Erlang and you can just follow the types. The goto example is probably handle_info for which there is a 'complete' example in the embedded repo for these docs.

4.4 Dynamic Supervision Trees

It is obviously possible to start and stop children on a normal supervisor by calling the appropriate functions and passing in the specification of the child involved, a supervisor has a *serverName* with which it was started and we just use that when calling the API against it. With a typical Supervisor the interaction is as follows.. (The behaviour of these functions mapping exactly onto the actual OTP documentation).

```
import Pinto.Sup as Sup

main :: Effect Unit
main = do
  -- Start the child with a spec
  Sup.startChild serverName $ Sup.spec
    { "my_child_name"
    , childType: Worker
    , start: MyGenServer.startLink {}
    , restartStrategy: RestartTransient
    , shutdownStrategy: ShutdownTimeout 5000
    }

  -- Stop and delete a child by id
  Sup.terminateChild serverName "my_child_name"
  Sup.deleteChild serverName "my_child_name"
```

This uses the same specification types as when *building a supervision tree* so shouldn't look too unfamiliar.

4.4.1 simple_one_for_one

A special case for supervisors in OTP is *simple_one_for_one*, where the whole supervision tree is set up for the benefit of a single type of child which gets defined by a template up front and gets 'completed' by the *start_child* call on the supervisor.

For this purpose, a separate module exists with a slightly different API under Pinto.Sup.Dynamic.

First we define a supervision tree that uses a a child template to set up everything except the arguments for a child:

```
import Pinto (RegistryName(..), RegistryReference(..), StartLinkResult)
import Pinto.Supervisor (ChildShutdownTimeoutStrategy(..), ChildType(..), RestartStrategy(..), crashIfChildNotStarted)
import Pinto.Supervisor.SimpleOneForOne (ChildSpec)
import Pinto.Supervisor.SimpleOneForOne as Sup

serverName :: RegistryName (Sup.SupervisorType OneForOneGenServerStartArgs OneForOneGenPid)
serverName = Local $ atom $ "one_for_one_example"

startLink :: Effect (StartLinkResult (Sup.SupervisorPid OneForOneGenServerStartArgs OneForOneGenPid))
startLink = Sup.startLink (Just $ Local $ atom "running_game_sup") init

init :: Effect (ChildSpec OneForOneGenServerStartArgs OneForOneGenPid)
init =
  pure { intensity: 100
    , period: Seconds 60.0
    , childType: Worker
    , start: OneForOneGen.startLink
    , restartStrategy: RestartTransient
    , shutdownStrategy: ShutdownTimeout $ Milliseconds 5000.0
```

The reader will note that the server name of a *dynamic* supervisor actually contains the types needed for the child to be started (the running pid of the child and the arguments the child expects). This allows us to then export an API for our supervisor to start children in that supervisor

```
startClient :: OneForOneGenServerStartArgs -> Effect OneForOneGenPid
startClient args = do
```

The API for terminating/stopping these children is slightly different, as rather than take an ID, the functions take the pid of the started child (just as in Erlang itself).

```
import Pinto.Sup.Dynamic as Sup

  -- Stop and delete a child by pid
  Sup.terminateChild serverName pid
  Sup.deleteChild serverName pid
```

By having this separate module for that special case of simple_one_for_one, we

- We enforce the use of the correct arguments for startLink on the child
- We enforce that children started have the correct/unified message types
- Get rid of the need for the specific errors that arise when calling the delete/terminate methods with the wrong arguments

CHAPTER 5

Web Server

The de-facto web server in Erlang is Cowboy for which a direct set of bindings is available in the Purerl package sets along with examples on how to use them.

This is good practise, to build direct bindings that are true (where possible) to underlying Erlang libraries/APIs and then build nicely typed Purerl on top of that. Building a whole application using these bindings directly however would be a bit burdensome and it is for this reason that I wrote Stetson which sits on top of Cowboy and exposes something a litle more Purerl specific.

Versioning is a case of 'check what we're linking to in the demo projects', which at the moment is Cowboy 2.8.

- *Stetson*: Intro to configuring Stetson as a web server
- *Routing*: Building typed routes to fire off handlers
- *Rest*: Writing restful handlers
- *Web sockets*: Writing Websocket handlers
- *Streaming*: Writing streaming (loop) handlers

5.1 Stetson

The primary Stetson entry point is *Stetson.configure* which returns a default configuration which can then be overriden using either the functions provided or by editing the record manually.

```
Stetson.configure
  # Stetson.port 8080
  # Stetson.bindTo 0 0 0 0
  # Stetson.startClear "http_listener"
```

Or

```
Stetson.startClear "http_listener"
  $ Stetson.configure { bindPort = 8080
                      , bindAddress = tuple4 0 0 0 0
                      }
```

Either method is comparable and down to preference and whether you wish to create erlang datatypes yourself.

The final step is to *startClear* (or other start methods such as startTls), which with a name kicks off the server with the configuration that was just built.

Very much like with Pinto, these options have as much as possible been taken 1-1 from the underlying library (Cowboy) so the documentation can be followed on the Cowboy docs rather than duplicated here with different terminology.

- StetsonConfig
- TCP options (ranch_tcp)
- HTTP options (cowboy_http)

It's all reasonably discoverable (the advantage of typed records in Purerl over grab-bag maps in Erlang (specced or otherwise), if anything is missing feel free to file an issue against Stetson (or even a pull request) - a lot of this functionality has been written and continues to be written on a very much on an on-demand basis.

5.2 Stetson Routing

The first step to take in Stetson once the basic configuration has been sorted out is to define some routes and callbacks to invoke when those routes are matched.

In Cowboy this is expressed as a list of string-based routes along with the modules to fire up when those routes are matched and those modules are then responsible for pulling bindings out of the route along with validation of those bindings. This is inverted somewhat in Stetson as we up front define our routes in an ADT along with the types we expect in them, and *then* map those to the paths that'll handle them in the web server.

5.2.1 The routes

The routes for the demo_ps project can be found in the '*./shared/*' directory as it's handy to also be able to build the routes safely from Purescript client code.

```
1 data Route
2   = Books
3   | Book Isbn
4   | Assets (Array String)
5   | EventsWs
6   | EventsFirehoseRest
7   | DataStream
8   | OneForOne
9   | Index
```

Being a demo project there are a pile of nonsensical routes on this ADT but amongst them we have *Books* and *Book Isbn*. The former being a collection handler for listing the books and the latter (*Book Isbn*) being a route that takes a specific ISBN to look up a book from the database. Note the types being used are actual domain types so we're not simply passing strings around. *Array String* is the equivalent of *[...]* in Cowboy "anything under this path" so we've got that in a couple of places; once for the directory containing all the CSS/JS/etc and one for ensuring that any of the client-side routes will all hit *index.html*.

The next thing to do after defining this ADT is to declare how this type can be mapped to and from the actual paths that will be serving the requests, this uses 'Routing.Duplex which was originally written for Purescript but handily cross-compiles in Purerl without much fuss. That's quite handy as that means the same package can be used on the client-side to generate correct URLS without lazily concatenating strings or going into restful dances to avoid the need to know urls at all.

```
apiRoute :: RouteDuplex' Route
apiRoute =
  path ""
    $ sum
        { "Books": "api" / "books" / noArgs
        , "Book": "api" / "book" / isbn segment
        , "EventsWs": "api" / "events" / "ws" / noArgs
        , "EventsFirehoseRest": "api" / "events" / "stream" / noArgs
        , "DataStream": "api" / "stream" / noArgs
        , "OneForOne": "api" / "one_for_one" / noArgs
        , "Assets": "assets" / rest
        , "Index": noArgs
        , "Index2": segmentExcept "api" / rest
        }
```

If you're unfamiliar with Purescript then these strings might look alarming, rest assured this is compile-time checked against the ADT and typos will not be tolerated. Thanks SProxy.

Turning our ADT into a usable string (and vice versa) is just a case of using code from *routing-duplex*, like so

```
routeUrl :: Route -> String
routeUrl = RouteDuplex.print apiRoute
```

We don't need to worry about the inverse here, because Stetson has support for Routing Duplex built-into it.

5.2.2 Using our routes with Stetson

Having defined these routes, we can register them with Stetson using *Stetson.routes* when building our configuration, we use the *Routes.apiRoute* defined above and match up the routes to callbacks that will be invoked when those routes match.

```
    $ GenServer.liftEffect
    $ Stetson.startClear "http_listener"
    $ Stetson.configure
        { routes =
          Stetson.routes2
            -- These routes are defined in a typed object
            -- that dictate
            -- a) What paths to reach them on
            -- b) What arguments they expect (typed(!!))
            -- So the callbacks to these names are typed and can be referred to in␣
↪shared/Routes.purs
            Routes.apiRoute
            { "Book": book
            , "Books": books
            , "EventsWs": eventsWs
            , "EventsFirehoseRest": eventsFirehoseRest
            , "DataStream": dataStream
            , "OneForOne": oneForOne
            , "Assets": PrivDir "demo_ps" "www/assets"
```

(continues on next page)

(continued from previous page)

```
            , "Index": PrivFile "demo_ps" "www/index.html"
            , "Index2": (\(_ :: String) -> PrivFile "demo_ps" "www/index.html")
            }
        , bindPort = args.webPort
        , bindAddress = tuple4 0 0 0 0
```

In the case of Book, which was defined as *Book Isbn*, it expects a callback of type (*forall msg state. Isbn -> StetsonHandler msg state*), where msg and state are entirely down to the handler itself to define. (The bulk of this handler is elided from the example as it's very REST specific).

```
-- And try and load the book which may or may not exist
book :: Isbn -> StetsonHandler Unit (Maybe Book)
book id =
  routeHandler
    { init:
        \req -> do
          --  Conveniently typed, and the Maybe just goes into state
          book' <- BookLibrary.findByIsbn id
```

It's a few steps to get to the point where you have a working dispatcher over routes in Stetson, adding a new route is a case of

- Adding the route to the ADT
- Adding the mapping for the path to the Route with routing-duplex
- Adding a handler for the route in Stetson configuration

The good news is that because it's all then type checked, changing the inputs to handlers or moving routes around isn't a guessing game - with larger projects this is quite a big deal indeed.

5.3 Stetson Handlers

The most common use-case for a handler in Stetson is to interact with the workflow enforced by Cowboy which encourages the "*correct*" usage of HTTP status codes by a series of optional callbacks. Whereas in Cowboy these optional callbacks are functions sat on a module, in Stetson these are functions added inline as part of the building process.

The handler for this in Cowboy is cowboy_rest for which there is the equivalent module Stetson.Rest.

A very basic route handler could look like this:

```
import Stetson.Types (routeHandler)

book :: Isbn -> SimpleStetsonHandler (Maybe Book)
book id =
  routeHandler (\req -> do
                          book <- BookLibrary.findByIsbn id
                          Rest.initResult req book)
    # Rest.contentTypesProvided (\req state -> Rest.result (jsonWriter : nil) req␣
↪state)
  where
    jsonWriter = tuple2 "application/json" (\req state -> Rest.result (writeJSON␣
↪state) req state)
```

We need an init of some sort, for which we're using *routeHandler*, and then we're providing the contentTypesProvided callback (which maps to content_types_provided in Cowboy) which further provides the callbacks to serve those content types (in this case just application/json along with a callback that just calls writeJSON on the current state).

The eagle-eyed reader will notice the use of *SimpleStetsonHandler*, which is an alias for *StetsonHandler msg state* where msg is fixed to type 'Unit' as Rest handlers have no reason to be receiving messages of any kind.

As many of these callbacks can be provided as are needed, some examples provided below.

```
allowedMethods :: (Req -> state -> Effect (RestResult (List HttpMethod) state))
resourceExists :: (Req -> state -> Effect (RestResult Boolean state))
malformedRequest :: (Req -> state -> Effect (RestResult Boolean state))
allowMissingPost :: (Req -> state -> Effect (RestResult Boolean state))
contentTypesAccepted :: (Req -> state -> Effect (RestResult (List (Tuple2 String␣
↪(AcceptHandler state))) state))
contentTypesProvided :: (Req -> state -> Effect (RestResult (List (Tuple2 String␣
↪(ProvideHandler state))) state))
deleteResource :: (Req -> state -> Effect (RestResult Boolean state))
isAuthorized :: (Req -> state -> Effect (RestResult Authorized state))
movedTemporarily :: (Req -> state -> Effect (RestResult MovedResult state))
movedPermanently :: (Req -> state -> Effect (RestResult MovedResult state))
serviceAvailable :: (Req -> state -> Effect (RestResult Boolean state))
previouslyExisted :: (Req -> state -> Effect (RestResult Boolean state))
forbidden :: (Req -> state -> Effect (RestResult Boolean state))
isConflict :: (Req -> state -> Effect (RestResult Boolean state))
```

These map almost directly onto their similarly named counterparts in Cowboy which means the documentation for the latter can be read to determine their usage. A complete workflow is provided in the Cowboy docs for the various responses that will be sent as a result of nagivating these callbacks.

While building a single rest handler in Cowboy and/or Stetson can be quite a verbose process, the nature of everything being a function in Stetson means that once commonality has been identified in a user application it is very easy to start composing handlers out of common functions (for example, a resourceExists could operate over a state of Maybe a and return true or false, there is no need to write this multiple times).

5.4 Stetson Websockets

One of the "coolest" things we can do with Stetson in Purerl and a suitable client written in Purescript is use websockets to send *typed* messages back and forth without too much ceremony.

The handler for this in Cowboy is cowboy_websocket for which there is the equivalent module Stetson.Websocket.

The first thing we need to do is setup the handler by specifying some sort of message type and our state type (in this case, just 'unit').

```
1 -- This is a receiving  handler, which receives the message  typr defined above, and␣
  ↪holds a state of 'Unit'
2 eventsWs :: StetsonHandler EventsWsMsg Unit
3 eventsWs =
4   routeHandler
5     { init
6     , wsInit: wsInit
7     , wsHandle: wsHandle
8     , wsInfo: wsInfo
```

5.4.1 init

At this point in time, this handler is still just a plain old Cowboy handler and we need to signal to Cowboy that we'd like it to start invoking the callbacks for Websockets (Also, it kicks this off in another process so messags can safely be sent to it).

```
  -- init runs in a different process to the ws handler, so probably just run the␣
↪default handler here
```

5.4.2 wsInit

Once we've informed Cowboy that this is to be a Websocket handler, it'll invoke our wsInit (*websocket_init*) in the correct process, so this is the time to subscribe to any messages we might want to forward down to the client.

```
  -- emitter is of type (msg -> Effect Unit), anything passed into that will appear␣
↪in .info
  wsInit s = do
    -- Get our pid
    self <- self
    -- Subscribe to the bus, and redirect the events into our emitter after wrapping␣
↪them in our type
    void $ liftEffect $ SimpleBus.subscribe BookLibrary.bus $ BookMsg >>> send self
```

5.4.3 wsHandle

We then have wsHandle for messages set to us by the client (*websocket_handle*) given to us as a Frame (binary, text, ping, etc) and we can easily parse json into our model at this point if we receive a text message, or process the binary etc.

```
  -- Receives 'Frame' sent from client (text,ping,binary,etc)
```

5.4.4 wsInfo

Finally we have *info* into which messages sent to our process from elsewhere in Erlang will be received so we can proxy them down to our client in the form of the Frame type (binary, text, ping, etc).

```
  -- Receives messages that were sent into 'emitter', typically so they can then be
↪'Replied' into the websocket
```

With the use of messages that can easily be serialised/deserialised to/from JSON defined in a shared folder, the client and server can very easily communicate with a stream of back and forth typed messages.

5.5 Stetson Streaming

A reasonably common pattern for streaming data to the client is to subscribe to a bus/process of some sort and then send that data to the client as it comes in. The handler for that in Cowboy is cowboy_loop for which there is the equivalent module Stetson.Loop

5.5.1 Loop from the onset

Just like with WebSockets, the first step is to set up a handler with an appropriate message type, for the handler needs to receive messages to send down to the client in the form of some sort of data.

```
-- This is a handler analogous to cowboy_loop
dataStream :: StetsonHandler DataStreamMessage Unit
dataStream =
  routeHandler
  { init
    , loopInit: loopInit
    , loopInfo: loopInfo
```

Then, our init needs to send an initial response down to the client before signalling to Cowboy that we're to become a loop handler.

```
loopInit req state = do
  self <- self
  void $ liftEffect $ DataSource.registerClient $ send self <<< Data
  pure state
```

And then all that's left to do is define the handler for dealing with the messages that come in.

```
loopInfo msg req state = do
  case msg of
    Data iodata -> do
      -- Then stream that down to the client
      void $ liftEffect $ streamBody iodata req
      pure $ LoopOk req state
    ...
```

This is a simplified version of the code in demo_ps repo which also attaches a monitor to the remote process so the connection can be closed in case the data source goes missing.

5.5.2 Switch to Loop from Rest

A common pattern across our codebases for streaming handlers, is to use the Rest callbacks to negotiate a sensible response based on auth/availability/etc and then switch into a looping handler for actually sending the data.

```
-- but then switches into a LoopHandler for streaming the data once Conneg has taken␣
↪place
eventsFirehoseRest :: StetsonHandler EventsWsMsg Unit
eventsFirehoseRest =
  routeHandler
    { init: \req -> Rest.initResult req unit
    , allowedMethods: allowedMethods
    , contentTypesProvided: contentTypesProvided
    , loopInit
    , loopInfo
```

We see here that our init kicks off the Rest workflow for which callbacks are also configured, but also there is a loopInit and loopInfo provided.

In our content callback, once we've negotiated the various REST callbacks, we can signal to Cowboy that we want to stream the data now

```
  -- we'll call streamReply on Cowboy to let it know that's what we're doing
  streamEvents =
    tuple2 "application/json"
      ( \req state -> do
          req2 <- streamReply (StatusCode 200) Map.empty req
          -- And then we'll switch to the LoopHandler (head back up to Loop.init)
          Rest.switchHandler LoopHandler req2 state
```

The full code for this can be found in the demo_ps repo.

CHAPTER 6

Logging

In classic Erlang projects, the de-facto for logging for a very long time has been lager which has served us well for a very long time indeed.

If integrating with legacy code which uses lager extensively then you probably want to stick with it, and for that we have purescript-erl-lager.

For new projects, or projects where logging has been wrapped up in a pile of macros anyway (does anybody *not* do this?), switching/starting with the logger that ships with Erlang as of OTP 21.0 is a safer bet.

Certainly as we roll forwards with Purerl development, focus and support will typically be given to Logger in preference to lager.

- *Lager*
- *Logger*

6.1 Lager

Adding *erl-lager* to *spago.dhall* and *lager* to *rebar.config* is required for the use of Lager in a Purerl project.

It is a very basic module, with functions for the various levels being exposed in the form

```
-- lager:info("Something ~p", [ A ]).
info1 :: forall a. String ->  a -> Effect Unit

-- lager:info("Something ~p ~p", [ A, B ]).
info2 :: forall a b. String -> a -> b -> Effect Unit
```

There are no checks against the format string in play, and are no plans to do anything more fancy than the above; so your mileage may vary. Pull requests will no doubt be accepted or indeed simply forking the project to do something more advanced if you really want to use Lager are both valid options.

6.2 Logger

Adding *erl-logger* to *spago.dhall* is all that's required for logger, as the module is bundled with OTP 21.0 by default.

While the API is still quite simplistic, it does at least have the ability for per-module logging to be configured as well as support up-front for structured logging, which means supplying filterable context as well as the inclusion of actual code context (module/function/etc). The API is a little less user-friendly up-front but this serves for a better experience in a production project.

Methods are exposed for the various log levels (info/debug/warning/notice/etc) that take in a metadata with all of the information for the logging call, as well as a report which describes the logging call. It is important to note that metadata is the *primary* purpose of logging and the report is secondary to that.

The underlying logger module is capable of much more than is exposed in the Purerl module and presently only supports the needs of logging styles requested by clients of the code so far. Pull requests and requests are obviously accepted.

```
let domain = (atom "the_domain" : nil)
    metadata = Logger.traceMetadata domain "This is the message for the log"
    report = {}
 _ <- Logger.info metadata report
```

This is obviously quite verbose, but in essence we end up with per-project helpers for the various domains present within the project that help us do the logging that we need across them.

```
domain :: List Atom
domain = (atom "acme") : (atom "project") : (atom "component") : nil

logInfo :: report. String -> { | report }->
logInfo = Logger.info <<< Logger.traceMetadata domain

logWarning :: forall report. String -> { | report }->
logWarning = Logger.warning <<< Logger.traceMetadata domain
```

The usage of which is then simply

```
logInfo "Something happened to this stream" { streamId: stream.id, nodeId: node.id }
```

Now this is actually not quite right, as we're stuffing data into the *report* that might best be in the *metadata*, instead we might want to consider building the custom metadata ourselves.

We can set this globally *per process* so that all logs from a single process will automatically have this metadata applied

```
-- During process initialisation (for example Gen.Init )
_ <- Logger.addLoggerContext { streamId: stream.id, nodeId: node.id }

-- A typical log call elsewhere
logInfo "Something happened to this stream" {}
```

Now the metadata will be supplied for every logging call in this process, and it will *be* metadata as opposed to report-data which is the correct place for it to be.

If we need to build our own metadata on a per-call basis, then this is slightly more involved as it needs merging with the underlying pre-filled in metadata (containing domain, type, text). We can either do some row-level magic with Purescript, or just supply all of this information ourselves as part of the logging call.

```
Logger.warning { domain, type: Logger.Trace, text: "Something happened to this stream
↪", streamId: stream.id, nodeId: nodeId  } {}
```

And now we're back to the start again, calling Logger directly. Obviously it's possible to wrap this up in whatever way it most convenient for the size of the project, but the important thing is that the flexibility and power is there to do proper structured logging all the way down.

CHAPTER 7

Messaging

Messaging. Without it, Erlang wouldn't work very well. Many concepts are modelled as processes which do things in response to messages, and a lot of APIs revolve around starting up a process and waiting for messages to come back from it (as well as further coordinating that process by sending messages to it).

This is great, but in a typed world we can't just be flinging any old messages around the show - we need to up front specify what messages we will be receiving and what we'll be sending. Thankfully the problem of components sending typed messages about the place has been solved several times in platforms like Elm and frameworks like Halogen and being long time users of both of these it is no surprise that the patterns and libraries that have emerged in Purerl borrow several key concept from them.

In general, a module that's modelled as a process will define a data type for the message that it plans on being able to receive. In order to receive a message from external code, an appropriate data constructor can be passed into that external code for this to take place. (Note: For legacy APIs this is often not possible, please see Interop/Messaginginterop/messaging for detals on how to work around this.

7.1 Subscribing to Incoming messages (pseudo-example)

```
-- Our 'receive' message type
data Msg = Tick
         | SomethingHappened
         | MessageReceived OtherModule.Msg


init :: Effect Unit
init =
  -- Subscribing to messages
  self <- self
  OtherModule.sendMessageToMe $ send self <<< MessageReceived

receiveMessage :: Msg -> State -> Effect State
receiveMessage msg state =
  case msg of
```

(continues on next page)

(continued from previous page)

```
    Tick -> ...
    SomethingHappened -> ..
    MessageReceived msg -> case msg of ...
```

In gen servers, that receiveMessage would be a *handleInfo*, in Stetson it'd be be an *info* and in other 'process containers' it could be called anything else, but in general the pattern will be 'subscribe to messages by passing in a callback', 'receive messages and modify state accordingly' - all nice and typed.

7.2 Sending Outgoing Messages (pseudo example)

```
-- Our 'send message type
data Msg = Hi String String String String
         | AnotherMessage
         | Etc

sendMessageToMe :: (Msg -> Effect Unit) -> Effect Unit
sendMessageToMe emitter = emitter $ Hi "bob"
```

Typically of course we might store that callback only to invoke it when something occurs, or every time something happens; but in this example we send a message right away that ends up in the message box of the Receiver and gets passed into whatever function the process container exposes for that.

Because most APIs therefore boil down to passing in an emitter of type *(msg -> Effect Unit)*, the underlying framework or library in use at that particular time is irrelevant to how that works. Most of the time, the means of doing this is getting a typed *Process msg*, and composing a *send* over the top of it along with the contructor for the appropriate message type coming into the process.

For getting hold of 'self', in most contexts there is a typeclass in Erl.Process (HasSelf) which will do the right thing in most cases. (Pinto.GenServer, Pinto.GenStateM, Stetson.Loop, Stetson.WebSocket, ProcessM)

- *Subscribing to messages*
- *Message Bus*
- *Monitors*
- *Process (spawn_link)*
- *Timers*

7.2.1 Subscribing to messages

A typical pattern in Erlang would be to get hold of the current process, and initiate a subscription to messages that will then be sent to that pid.

```
Self = self()
some_api:subscribe(Self),
```

An equivalent to this exists in *Erl.Process.Raw*

```
self :: Effect Pid
```

But as we can see, this Pid has no type information associated with it so we could be sending and receiving anything. This is useful in test suites (when we can just use type inference, wishes, and prayers to send messages about the place for assertion), but less useful in a production setting.

```
send :: forall a. Pid -> a -> Effect Unit
```

What we really need is a typed Process

Getting hold of a Process Msg

In the latest package sets, a typeclass is defined so that within any monad that supports it we should be able to get hold of a Process Msg if one exists (as usually the type of Msg is encoded within that context).

```
class HasSelf (x :: Type -> Type) a | x -> a where
  self :: x (Process a)
```

This exists so that custom process types can be designed, but is implemented for all the common process types that already exist across Erl.Process, Erl.Pinto and Erl.Stetson.

Sending messages to a spawned process

The simplest example is a spawned process via Erl.Process.spawnLink

```
import Erl.Process (spawnLink, ProcessM, self, send)
import Pinto.Timer (sendEvery)

-- Our message type
data Msg
  = Tick
  | Stop

main :: Effect Unit
main = do
  -- note: pid is of type 'Process Msg'
  pid <- spawnLink startWorker

  -- After 5 seconds, send a Stop message to the spawned pid
  sleep 5000
  send pid Stop

startWorker :: ProcessM Msg Unit
startWorker = do

  -- note: me is of type 'Process Msg'
  me <- self

  -- Ask for a Tick message to be sent every second
  sendEvery 1000 Tick me

  -- And enter the receive loop
  workerLoop

workerLoop :: ProcessM Msg Unit
workerLoop = do
  -- msg is of type Msg
  msg <- receive
  case msg of
    Stop -> pure unit -- unit because it's ProcessM msg result
    Tick ->  do
```

(continues on next page)

(continued from previous page)

```
      log "Tick"
      workerLoop
```

As can be seen, we're in the ProcessM monad which has two types assocated with it - the messages we expect to receive and the return result of the process (which is typically unit because it'll get discarded anyway!).

So long as we restrict ourselves to using the typed API that exists here, we will never receive a message that we don't expect and life is good.

Sending messages to a spawned GenServer

The GenServer context is a bit heavier, most operations taking place inside a 'ResultT'

```
ResultT cont stop msg state result
```

The only relevant type parameter here is 'msg', and an implementation of HasSelf exists for this context that'll get you a *Process msg*

```
import Pinto.GenServer (InitFn, InfoFn, liftEffect)
import Erl.Process (self, send)

...

data Msg = Tick | SomethingElse

init :: Gen.InitFn State Msg
init =
  me <- self
  liftEffect $ sendEvery 1000 Tick me
  pure $ InitOk {}

handleInfo :: InfoFn Unit Unit Msg State
handleInfo msg state = ....
```

Receiving messages in a Stetson WebSocket handler

Stetson also implements HasSelf for websocket callbacks

```
# WebSocket.init (\s ->  do
                            self <- WebSocket.self
                            Gen.lift $ SomeModule.sendMeSomething $ SomeMessage >>>␣
↪send self

                            pure $ Stetson.NoReply s
                          )
# WebSocket.info (\(SomeMessage msg) state -> ...
```

Receiving messages in a Stetson Loop handler

And the same goes for the Loop handlers as well

```
# Loop.init (\s ->  do
                            self <- Loop.self
                            Gen.lift $ SomeModule.sendMeSomething $ SomeMessage >>>␣
↪send self

                            pure $ Stetson.NoReply s
                           )
# Loop.info (\(SomeMessage msg) state -> ...
```

In essence, anywhere we can get hold of Process msg, we can create an emitter that'll result in messages of the right type being sent to that process.

The typical convention (at present) for anything wishing to send messages back to a calling process in Purerl, is not to send the message directly but to instead accept a callback and let the consumer choose how to consume those messages.

```
-- We could just pass in a process to send messages to
 subscribe :: Config -> Process Msg -> Effect Unit

 -- Or, we could pass in a callback, which allows the consumer to decide what to do
 subscribe :: Config -> (Msg -> Effect Unit) -> Effect Unit
```

Of course, whilst passing the Process Msg in is less flexible, it is also less error prone - consider a callback with an error in it which crashes the process it was passed to - care should be taken to handle these when designing APIs.

7.2.2 Message Bus

A convenient method of sending messages to other processes is via a message bus; one can be contructed quite easily using gproc in Erlang and for ease a package erl-simplebus does just that.

Defining a bus

A bus is just a 'name' with a phantom type associated with it onto which messages of that type can be placed and received by multiple listeners

```
data BusMessage = StreamStarted String
                | Data Binary
                | Eof

bus :: SimpleBus.Bus String BusMessage
bus = SimpleBus.bus "file_reading_bus"
```

In the above example, we define a bus called *file_reading_bus*, which will be capable of distributing messages of *BusMessage*, The convention is that a module wishing to expose a bus will just export it via its module definition. By keeping the constructor for the ADT private, only the owner of the bus will be able to place messages on it.

```
module StreamReader ( bus
                    , BusMessage
                    ) where
```

To place a message onto the bus, the module that 'owns' the bus need only call *send*, passing in the bus involved and a constructed message.

```
_ <- SimpleBus.send bus $ StreamStarted "stargate.ts"
_ <- SimpleBus.send bus Eof
```

Subscribing to a bus

From another process or module, we only need call *subscribe*, passing in the bus and a callback to receive the messages. In a Genserver, this would look like this

```
data Msg = Tick
         | DoSomething String
         | StreamMessage StreamReader.BusMessage

init :: InitFn Unit Unit Msg State
init = do
  self <- self
  _ <- liftEffect $ SimpleBus.subscribe StreamReader.bus $ send self <<< ␣
↪StreamMessage
  pure $ InitOk {}

handleInfo :: InfoFn Unit Unit Msg State
handleInfo msg state = do
  case msg of
    Tick  ->
      handleTick state
    DoSomething what  ->
      handleSomething what state
    ReaderMessage msg  ->
      handleReaderMessage msg state
```

We can see clearly here the pattern of lifting an external module's message into our own type so we can handle it in our *handleInfo* dispatch loop.

Unsubscribing from a bus

It's actually rare that we'll ever unsubscribe from a bus; most of the time we'll subscribe on a process startup and then allow the subscription to be automatically cleaned up on process termination.

However, it's worth pointing out that *SimpleBus.subscribe* actually returns a reference of type *SubscriptionRef* which we can stash in our process state for use later on.

```
init :: InitFn Unit Unit Msg State
init = do
  self <- self
  busRef <- liftEffect $ SimpleBus.subscribe StreamReader.bus $ send self <<<␣
↪StreamMessage
  pure $ InitOk { busRef: Just busRef }


unsubscribe :: State -> Effect State
unsubscribe s@{ busRef: Nothing } = pure s
unsubscribe s@{ busRef: Just ref } = do
  void SimpleBus.unsubscribe ref
  pure s { busRef = Nothing }
```

When to use a bus

A bus is an extremely lazy way of sending messages about the place and care must be taken not to overuse them in complicated orchestration scenarios. In general they're *really* good for distributing *events* to multiple subscribers to let them know something has already happened and not *commands* that tell things to happen.

7.2.3 Monitors

Pids and processes

One of the most useful concepts in erlang is the monitor. Monitoring a pid means we get sent a message if that pid is non-existent, or when it otherwise terminates.

Obviously a direct FFI to *erlang:monitor* won't work in most processes, as it'll result in native Erlang tuples being sent directly to our Purerl where it'll immediately fail to match our expected message types.

Pinto has a wrapper for monitors that works around this by routing the messages through an emitter.

```
data Msg =
  ProcessDown MonitorMsg

self <- self
Monitor.monitor pid $ send self <<< ProcessDown
```

MonitorMsg has quite an involved type because it holds all the information that an Erlang monitor would give us.

```
data MonitorMsg
  = Down (MR.RouterRef MonitorRef) MonitorType MonitorObject MonitorInfo
```

It's quite common to disregard this message entirely as we can bundle the information that we actually need into our message at the time of subscription.

```
data Msg =
  ProcessDown Pid

self <- Gen.self
Monitor.monitor pid (\_ -> self ! ProcessDown pid)
```

Gen servers

A running GenServer has a pid that you can get hold of if you started it with startLink yourself which is precisely *not* how one would usually start a GenServer.

A convenience method exists therefore for getting hold of the typed process of an already-running gen server, so long as you have access to the name of that gen server (typically exported from the module implementing the gen server).

```
data Msg =
  ProcessDown Pid

self <- self
maybePid <- liftEffect $ GenServer.whereIs MyCoolGenServer.serverName
case maybePid of
  Just pid -> do
    void $ liftEffect $ Monitor.monitor pid (\_ -> send self DataSourceDied)
    pure unit
```

(continues on next page)

(continued from previous page)

```
  _ -> do
    liftEffect $ send self DataSourceAlreadyDown
    pure unit
```

7.2.4 Process (spawn_link)

Using gen servers for absolutely everything is very much overkill, quite often we just want to spin up a child process via *spawn* or *spawn_link*, and then communicate with it by sending it messages directly. This can be useful when performing longer tasks as part of a gen server for example but not wanting to block the execution of that gen server

We'll build up that as the example here because it's such a common pattern.

Spinning up a child process

The bare minimum to spin up a child process is

- Define the type of message it will expect to receive
- Define a function that will operate within the context of the newly spawned child process (the ProcessM Monad)

```
data ChildMsg = Tick

childProcess :: ProcessM ChildMsg Unit
childProcess = pure unit

init :: Effect Unit
init = do
  _ <- Process.spawnLink childProcess
```

Now in this example, the process will start up and then immediately terminate because we don't do anything in the function invoked as part of spawnLink - we did say the bare minimum required..

We can change this to wait for a message indefinitely and *then* exit by using the functions given to us in the ProcessM context.

```
data ChildMsg = Tick

childProcess :: ProcessM ChildMsg Unit
childProcess = do
  msg <- receive
  case msg of
    Tick -> pure unit

init :: Effect Unit
init = do
  _ <- Process.spawnLink childProcess
```

Or indeed, wait for a message and then loop and wait for a message again

```
data ChildMsg
  = Tick
  | Exit

childProcess :: ProcessM ChildMsg Unit
childProcess = do
```

(continues on next page)

(continued from previous page)

```
  msg <- receive
  case msg of
    Tick -> do
      log "tick"
      childProcess
    Exit -> pure unit

init :: Effect Unit
init = do
  _ <- Process.spawnLink childProcess
```

Note: an Exit value was added in this example, as *some* branch of the function *has* to return the expected type (unit), or the compiler will get upset.

So how we do we send this newly awakened process a message? We're presently discarding the result of spawnLink - which is of type *Process ChildMsg*, so we'll want that obviously.

```
data ChildMsg
  = Tick
  | Exit

childProcess :: ProcessM ChildMsg Unit
childProcess = do
  msg <- receive
  case msg of
    Tick -> do
      log "tick"
      childProcess
    Exit -> pure unit

init :: Effect Unit
init = do
  child <- Process.spawnLink childProcess
  child ! Tick
```

Next up we'll probably want to get a message back from our long running process, to do that we'll probably want to pass it a pid or a process - so let's move into the context of a GenServer and spin up a child process from there.

```
data ChildMsg
  = Tick
  | Exit

data Msg
  = Response

childProcess :: Process Msg -> ProcessM ChildMsg Unit
childProcess parent = do
  msg <- receive
  case msg of
    Tick -> do
      parent ! Response
      childProcess parent
    Exit -> pure unit

init :: InitFn Unit Unit Msg {}
init = do
  self <- self
```

(continues on next page)

(continued from previous page)

```
  child <- Process.spawnLink $ childProcess self
  child ! Tick

handleInfo :: InfoFn Unit Unit Msg State
handleInfo msg state =
  case msg of
    Response -> ...
```

And voila, now we have a gen server that starts a child process that when sent a 'Tick' message, responds to use with a 'Response' message and it's all type safe thanks to the wonders of Purescript.

7.2.5 Timers

Timers are a special case in Erlang itself, because they're leaned on so heavily there has always been effort to avoid spinning up processes all over the show for them. The module *timer* was implemented as a single process which processed all of the timer messages and maintains the references. That proved to be a bottleneck so we now have an even lower level timer implemention in the *erlang* module that avoids even that.

This is a long winded way of saying that the proxy processes that are used to perform message redirection/emitting from Pinto.MessageRouter are not desirable for this use case, instead the API takes a *Process msg* to send the messages to, and a pre-built *msg* to send when the timer is fired. This means we don't incur any extra overhead by virtue of being in Purerl.

Note: Timer.sendEvery uses the old *timer:send_every* implementation and Timer.sendAfter uses the new *erlang:send_after* code, this is an implementation detail but is worth pointing out in case it causes confusion.

```
data Msg = Tick

init :: InitFn Unit Unit Msg State
unit = do
  self <- self
  liftEffect $ Timer.sendAfter 500 Tick self
  pure $ InitOk {}

handleInfo :: InfoFn Unit Unit Msg State
handleInfo msg state = ...
```

Or

```
data Msg = Tick

init :: InitFn Unit Unit Msg State
unit = do
  self <- Gen.self
  Gen.lift $ Timer.sendEvery 500 Tick self
  pure $ InitOk {}

handleInfo :: InfoFn Unit Unit Msg State
handleInfo msg state = ...
```

Timers operate on anything that implement *HasProcess msg*, thus we can invoke them targetted at GenServer, ProcessM, Loop handlers, etc etc..

```
data ChildMsg
  = Tick
```

(continues on next page)

(continued from previous page)

```
  | Exit

childProcess :: ProcessM ChildMsg Unit
childProcess = do
  msg <- receive
  case msg of
    Tick ->
      childProcess
    Exit -> pure unit

init :: InitFn Unit Unit Msg State
init = do
  child <- liftEffect $ Process.spawnLink childProcess
  Timer.sendEvery 500 Tick child
  pure $ InitOk { child }
```

CHAPTER 8

Interop

Whether writing Purescript green-field, or writing against legacy code - the nature of the platform is that eventually you will need to write some Erlang. There are a lot of popular modules in Erlang that don't have maintained bindings yet and swathes of core Erlang code that aren't exposed in Purescript.

In general this gets done ad-hoc as required in modules internal to our projects and once there is sufficient coverage these get promoted to actual modules in the package set. It is wise to avoid writing Erlang *as much as possible* when committing to Purerl no matter how tempting it may be to "just drop into Erlang for this module", that'll be where you'll get crashes for the next couple of days.

The best practise is to write *thin* bindings that exactly represent the underlying types and functions and then use that from more Purescript that exposes that in a nicer way. It is often tempting to skip that step and go straight to 'exposed in a nice way' but this is usually a mistake.

- *FFI*
- *Effects*
- *Errors*
- *Messaging*

8.1 FFI

8.1.1 Module names

A module **MyModule.purs** with the below definition will compile to the Erlang module **myModule@ps**. Note the camelCasing and the suffix of @ps added to the module name.

```
module MyModule where
```

A module **MyModule.purs** with the below definition will compile to **acmeCorp_myModule@ps**. Note the underscore between namespaces, as well as the camelCasing per namespace and the eventual suffix of @ps.

```
module AcmeCorp.MyModule where
```

These details are important when writing foreign function imports in Erlang, the means of doing so being to create an Erlang module next to the Purescript module, with the same name but with a .erl suffix. **MyModule.purs** therefore would have a corresponding **MyModule.erl** if we wanted to do FFI.

The name of the compiled module comes into play, as the Erlang module requires an appropriate name to go with it.

- **MyModule.purs** with **module MyModule** in Purescript would have a foreign import module of **MyModule.erl** containing **-module(myModule@foreign)** in Erlang
- **MyModule.purs** with **module AcmeCorp.MyModule** in Purescript would have a foreign import module of **MyModule.erl** containing **-module(acmeCorp_myModule@foreign)** in Erlang

8.1.2 Foreign Function Imports

Having defined a Purescript module with an appropriately named Erlang module side by side, the next thing would be to define a function in Erlang that we can call from Purescript.

```
-module(myModule@foreign).

-export([ add/2 ]).

add(X,Y) -> X + Y.
```

To create a function that's callable from Purescript, we need to import this as a foreign function in our Purescript module. This can be exported from the module just like any other function at that point.

```
module MyModule ( add )
  where

foreign import add :: Int -> Int -> Int
```

In general where we have legacy Erlang of the form *my_module:do_stuff*, we'd be creating *MyModule.purs* and *MyModule.erl* and defining functions that map onto that legacy API, and thus we can interact with our existing code in a reasonably safe manner.

8.2 Effects

A lot of interop is effectful, and care really must be taken to describe it as such, consider the legacy API below

```
{ ok, Handle } = legacy_api:open_database(ConnectionString),
{ ok, Value } = legacy_api:read_from_database(Handle, Key).
```

A *naive* implementation of an FFI module for this might look like this

```
module LegacyApi where

foreign import data Handle :: Type

foreign import openDatabase :: String -> Handle
foreign import readFromDatabase :: Handle -> String -> String
```

```
-module(legacyApi@foreign).

-export([ openDatabase/1, readFromDatabase/2 ]).

openDatabase(ConnectionString) ->
  { ok, Handle } = legacy_api:open_database(ConnectionString),
  Handle.

readFromDatabase(Handle, Key) ->
  { ok, Value } = legacy_api:read_from_database(Handle, Key),
  Value.
```

But this would be a lie, both opening a database and reading from a database are clearly effectful actions; whilst this code will work when invoked from Purescript, the effectful actions will be taking place outside of the Effect system and this will bite us in the ass in the form of runtime errors later down the line when we accidentally end up invoking side effects from the wrong processes.

A more *correct* implementation of this would be to define these functions as effectful.

```
module LegacyApi where

foreign import data Handle :: Type

foreign import openDatabase :: String -> Effect Handle
foreign import readFromDatabase :: Handle -> String -> Effect String
```

We can view an Effect as 'a function' to be invoked at the top level of execution - we might create a whole stack of effects as a result of calling an effectful function and these all get bubbled up to the point of entry which is then responsible for actually unpacking the result. Any effectful action is just a function that returns a function - functions all the way down.

```
-module(legacyApi@foreign).

-export([ openDatabase/1, readFromDatabase/2 ]).

openDatabase(ConnectionString) ->
  fun() ->
    { ok, Handle } = legacy_api:open_database(ConnectionString),
    Handle
  end.

readFromDatabase(Handle, Key) ->
  fun() ->
    { ok, Value } = legacy_api:read_from_database(Handle, Key),
    Value
  end.
```

8.2.1 Passing effectful Purescript functions back to Erlang

Quite often, Erlang APIs will take in a module name on which it will invoke several functions (perhaps defined as a "behaviour"), easy examples come to mind would be the gen_server callbacks and cowboy_rest/cowboy_loop callbacks. For the purposes of this example we'll define an interface for handling events from some sort of legacy Erlang system.

An implementation of our imaginary event callback module in Erlang might look like this

```
-module(callback_module).

-export([ handle_event/1 ]).

handle_event(Event) ->
  db:write_event(Event).
```

And we'd register that with the system with a call that looked something like

```
legacy_system:register_callbacks(callback_module).
```

If we wanted to write our callback module directly in Purescript, a naive implementation would look like this

```
module CallbackModule where

handle_event :: Effect Atom
handle_event ev = do
  void $ Db.writeEvent ev
  pure $ (atom "ok")
```

Registered with something like this

```
legacySystem.registerCallbacks (atom "callbackModule@ps)
```

However, if we are to invoke handle_event from Erlang, we would quickly discover that it does not return the *(atom "ok")* as expected, but instead something like *#Fun<callbackModule@ps.97.23242010>* (because an Effect is just a function).

We could remove the Effect from our function definition but this would leave us unable to perform side effects. Handily we have functions to help with this kind of dance in Effect.Uncurried

```
module CallbackModule where

import Effect.Uncurried (EffectFn1, mkEffectFn1)

handle_event :: EffectFn1 Event Atom
handle_event = mkEffectFn1 \ev -> do
    void $ Db.writeEvent ev
    pure $ (atom "ok")
```

This will give us an effectful function in a callback, but at the top level it'll execute the effect and return the result to the native Erlang code. These uncurried helpers are available for functions up to 10 arguments deep and if you really need more than that the only real problem is that you have a function that big in the first place - creating additional versions of mkEffectFn is just a case of taking the code from the Effect.Uncurried module and adding some parameters.

8.3 Error handling

In the previous code, we had the following FFI

```
-module(legacyApi@foreign).

-export([ openDatabase/1, readFromDatabase/2 ]).

openDatabase(ConnectionString) ->
  fun() ->
```

(continues on next page)

(continued from previous page)

```
    { ok, Handle } = legacy_api:open_database(ConnectionString),
    Handle
  end.

readFromDatabase(Handle, Key) ->
  fun() ->
    { ok, Value } = legacy_api:read_from_database(Handle, Key),
    Value
  end.
```

There are runtime crashes in this code that may or may not be desirable *("it depends")*. Let's say for the sake of argument that we in a situation where failing to open a database shouldn't crash the containing process.

A good way to model this in Purescript would be to expose the API as a **Maybe Handle**, or **Either ErrorMessage Handle**

```
foreign import openDatabase :: String -> Effect (Maybe Handle)
```

By snooping around some other compiled Erlang, we can see that **Maybe Handle** is represented as a tuple of either **{just, Handle}** or **{nothing}**, so in our FFI we could use this to fulfil the foreign import definition above.

```
openDatabase(ConnectionString) ->
  fun() ->
    case legacy_api:open_database(ConnectionString) of
      { ok, Handle } -> { just, Handle };
      _ -> {nothing}
    end
  end.
```

Once again however, we're showing the wrong way to do things before we demonstrate the right way. Relying on the types that the compiler generates is typically a bad way of doing business, they are subject to change and aren't remotely type-checked. The pattern is therefore to write an FFI that passes in the appropriate contructors for the Maybe type, and then export a function that uses this FFI and hides that detail.

```
foreign import openDatabaseImpl :: (Handle -> Maybe Handle) -> Maybe Handle -> String␣
↪-> Effect (Maybe Handle)

openDatabase :: String -> Effect (Maybe Handle)
openDatabase = openDatabase Just Nothing
```

and

```
openDatabase(Just, Nothing, ConnectionString) ->
  fun() ->
    case legacy_api:open_database(ConnectionString) of
      { ok, Handle } -> Just(Handle);
      _ -> Nothing
    end
  end.
```

This is typically the pattern for mapping to code that returns Purescript types and if you find yourself writing more code than this in Erlang then it's a sign that the FFI is too heavy and a thinner layer (and more Purescript) is required.

Note: While the above is "correct", it must be pointed out that in most of our code these days, we simply return {just} and {nothing} from FFI as a matter of course as it is very common - for most other data types however, constructors are still passed in.

8.4 Messaging

Yes - another chapter called "messaging", because this *is* Erlang after all.

It is very common for legacy APIs to send arbitary messages back to the invoking process as a means of communication, convenient, useful, handy... not immediately practical in Purescript however.

Consider the following code, where in Erlang we subscribe to some API that immediately starts sending us some sort of erlang tuple/record.

```
%% Subscribe to the legacy API
{ ok, Ref } = legacy_api:start()

%% And start receiving messages from it
receive
  #legacy_message{} ->  ... ... ... ...
```

If we were to write a straight wrapper for this API in Purescript, it'd look very simple indeed

First, the purescript foreign import, which merely invokes the function and returns the ref

```
module LegacyApi where

foreign import start :: Effect Handle
```

Which, in the Erlang is unpacked as thus

```
-module(legacyApi@foreign).

start() ->
  fun() ->
     { ok, Ref } = legacy_api:start(),
     Ref
  end
end.
```

Now we have a problem - if we try and use this in Purescript, our message receiving code has to operate on the Foreign data type because it has no idea what an Erlang record is.

A further call into the LegacyApi wrapper could unpack this of course so this doesn't present an immediate problem.

```
do
  _subscription <- LegacyApi.start

  msg :: Foreign <- receive

  case LegacyApi.interpretForeign msg of
    LegacyApi.ThisHappened -> ....
```

This *might* be okay, but it means if we want to receive any other kind of message we are out of luck unless we pack *them* into Foreign as well, and ask various mappers to attempt to unpack these foreigns in sequence until one works and oh boy this is not enjoyable in the slightest.

```
do
  _subscription <- LegacyApi.start
  _subscription2 <- LegacyApi2.start

  msg :: Foreign <- receive
```

(continues on next page)

(continued from previous page)

```
  let ourMsg :: Msg
      ourMsg = case LegacyApi.interpretForeign msg of
                  Just r -> Just $ LegacyMsg r
                  Nothing -> LegacyMsg2 <$> LegacyApi2.interpretForeign msg
```

This can somewhat get out of hand as we interact with more APIs and isn't a terribly forthright way of doing business, what we really want to write is

```
do
  msg <- receive
```

and that be the end of it

8.4.1 The choices

It'd be nice to be able to unpack these Foreigns into sensible types before we see them in our process, and to do this we have the following options

- *Routing* - intercept the messages with a proxy process and lift them into more appropriate types before sending them to the owning process
- *Untagged Unions* - describe the messages with an ADT and have them matched inline into more appropriate types

Most of the time you'll want Routing as processes are cheap and this is easy, but if writing a wrapper around a native Erlang library, untagged unions might be more useful.

8.5 Message Routing

The package erl-pinto contains a module (Pinto.MessageRouter) whose job it is is to perform the subscription on your behalf and then translate any messages using a provided function.

Practically any legacy API with a '*start*' and '*stop*' of some sort (subscribe/unsubscribe) can be shuffled behind a message router and that will convert the messages and pass them into the appropriate emitter.

Note: The message router will automatically terminate when its parent terminates and also call the stop method it was provided with when that happens.

Using the example on the original messaging page, if we wished to use that legacy API as written, we could invoke it behind a message router inside a gen server or similar like so

```
import Pinto.MessageRouter as MR
import LegacyApi as LegacyApi
import Pinto.GenServer (InitFn)
import Erl.Process (self, send)

data Msg
 = SomeMessage
 | LegacyMessage LegacyApi.Msg

init :: InitFn Unit Unit Msg State
init = do
  me <- self
  MR.startRouter LegacyApi.start LegacyApi.stop (send me <<< LegacyMessage <<<␣
↪LegacyApi.interpretForeign)
```

(continues on next page)

(continued from previous page)

In this, we start up a router - letting it know about the start/stop methods of the LegacyApi and in the callback

- calling interpretForeign on the Foreign tht was received
- lifting it into the Msg type with the LegacyMessage constructor
- send it to the parent process

Thus enabling us to receive messages from more than one source, but lifted into the correct types

```
handleInfo msg state =
 case msg of
   SomeMessage -> handleSomeMessage state
   LegacyMessage msg -> handleLegacyMessage msg state
```

An equivalent method **MR.maybeStartRouter** exists for cases where an instance of our legacy code may fail.

A note worth making is that this incurs the "cost" of an additional process so shouldn't be used in excessively performance oriented code - in that case we'd be better off

- accepting the foreign messages directly
- Simply writing code in Erlang to do the lifting
- Using *Untagged Unions <messaging-untagged>* to make our purescript aware of the incoming types in the first place

8.6 Untagged Unions

If when interacting with our legacy API we decided that the cost of spinning up an extra process was too much, it is possible with the use of the package erl-untagged-union to define an ADT in Purescript that maps onto the original data type and an instance of the appropriate typeclass to tell the code how to match on it. This isn't *too* different philosophically from the act of calling multiple interpreters against a Foreign to see which one succeeds except it does provide a more formal way of describing the alternatives and can provide value when writing code against a message based API that is going to be in heavy use and has multiple message types that need unpacking.

Consider the following legacy API in Erlang:

```
%% Subscribe to the legacy API
{ ok, Ref } = legacy_api:start()

%% And start receiving messages from it
receive
  { data, From, Binary } -> ...
  { info, From, {trace, Binary} } -> ...
  { error, Error } -> ...
```

We could describe these messages in the form of a Purescript ADT pretty easily

```
data LegacyMsg
 = Data Pid Binary
 | Info Pid (Tuple Atom Binary)
 | Error Foreign
```

And we *could* indeed provide a method as part of the legacy API to convert the foreign messages into this, if we were so inclined

```
interpretForeign :: Foreign -> LegacyMsg
interpretForeign = ...
```

But this is potentially burdensome to write - not to mention error prone, so instead we can use the untagged unions module to help describe this message.

8.6.1 Describing the types

```
import Data.Generic.Rep (class Generic)
import Erl.Untagged.Union as U

derive instance genericMsg :: Generic LegacyMsg _

instance runtimeTypeLegacyMessage ::
  U.RuntimeType
    LegacyMsg
    ( U.RTOption (U.RTTuple3 (U.RTLiteralAtom "data") U.RTWildCard U.RTBinary)
        (U.RTOption (U.RTTuple3 (U.RTLiteralAtom "info") U.RTWildCard (RT.Tuple2 RT.
↪Atom U.RTBinary))
          (U.RTTuple2 (U.RTLiteralAtom "error") U.RTWildCard)))
```

The way this works, is that if a more concrete type than *RTWildCard* is provided, then in order for the message to be unpacked by that line then it *has* to match. Obviously in this case there is a atom with a literal value being matched against as a discriminator so we could get away with just using RTWildCard in the rest of the expression like so

```
import Data.Generic.Rep (class Generic)
import Erl.Untagged.Union as U

derive instance genericMsg :: Generic LegacyMsg _

instance runtimeTypeLegacyMessage ::
  U.RuntimeType
    LegacyMsg
    ( U.RTOption (U.RTTuple3 (U.RTLiteralAtom "data") U.RTWildCard U.RTWildcard)
        (U.RTOption (U.RTTuple3 (U.RTLiteralAtom "info") U.RTWildCard (RT.Tuple2 RT.
↪RTWildcard U.RTWildcard))
          (U.RTTuple2 (U.RTLiteralAtom "error") U.RTWildCard)))
```

We don't tend to do that however, in general the more accurately you describe your expected messages the more useful this library becomes.

8.6.2 Handling the messages

When writing a receive block that can be sent these untagged messages, we need to describe to the type system the types of messages we expect to receive.

```
import Erl.Untagged.Union (Union, type (|$|), type (|+|), Nil)
import Erl.Untagged.Union as U

type Msg
 = U.Union |$| LegacyMsg |+| Nil
```

"Msg is an untagged Union that may contain LegacyMsg, as described in the runtimeType typeclass"

```
msg :: Msg <- receive
(U.case_
 # U.on ( \(legacyMsg :: LegacyMsg) ->
   case legacyMsg of
     Data pid bin -> ...
     Info from info -> ...
     Error err -> ...
   )) msg
```

Further message types could be added and described and matched as thus

```
import Erl.Untagged.Union (Union, type (|$|), type (|+|), Nil)
import Erl.Untagged.Union as U

type Msg
 = U.Union |$| LegacyMsg |+| OtherMsg |+| Nil
```

with

```
msg :: Msg <- receive
(U.case_
 # U.on ( \(legacyMsg :: LegacyMsg) ->
   case legacyMsg of
     Data pid bin -> ...
     Info from info -> ...
     Error err -> ...
   )
 # U.on ( \(otherMsg :: OtherMsg) ->
   case otherMsg of
     ...
   )
) msg
```

In this way, the compiler will let us know if we're not being exhaustive.

8.6.3 Pros and cons..

- This mechanism is incredibly useful for describing message based APIs with complex message types in cases when we don't want the cost/burden of spinning up an additional proxy process but still want a convenient mechanism for unpacking these message types without having to write error-prone mapping code.
- It is however possible to make mistakes in the description instead, this mechanism just formalises the process somewhat.
- Once you have made the decision to write a process that receives an untagged union that means all messages need to be described as part of this untagged union - even if they are native Purescript types.

For the most part therefore, the message router is more convenient. For a good example of an API that uses untagged unions to good effect, have a look at the source code for erl-gun.

CHAPTER 9

Navigation

I've split this giant README into several sections, some of which are going to be pretty boring to read (or just simply more boring) than others. If you're already familiar with Erlang then you'll be most interested in the more Purerl specific code examples found within. If you get stuck at all, the easiest thing to do is ask the question "how does the demo_ps project do it?" and refer to that and the documentation here that describes it.

While it's possible to get started by just following these code samples and taking them further, it is expected that the reader unfamiliar with Erlang will spend time reading about those concepts further in more erlang-specfic documentation before building production apps in Purerl!

If you're still stuck, then e-mail me at robashton@codeofrob.com, or file an issue over at the GH repo for these docs.

- *Building demo_ps*
- *Application Structure*
- *Editor Setup*
- *Basic OTP*
- *Building a web server*
- *Logging*
- *Messaging*
- *Interop/FFI*