Nix At Work?

How? Why? How about you?

Pol Dellaiera

CfgMgmtCamp Ghent

Monday 2 February 2026

Introduction

Who am I?

- Hi, I am Pol Dellaiera.
- **Software Engineer** at the **European Commission**.
- **Scientific Collaborator** at UMONS University.
- In 2021, I discovered the ❄ ecosystem.
- I started using it at work the same year.
- I started advocating for it the same year.
- In 2025, after 10 years, I left **Digit** and joined **DG EAC**.
- Some of our pipelines are now powered by Nix.
- We use Nix to build some of our container images.
- We also extensively use Nix Shells to spawn development environments, and much more.

What is Nix?

Well, Nix is

Way too many things.

An opinionated definition

Nix is a **build tool**, it can build **anything**:

- Shell scripts
- Packages
- Container images
- Operating systems

The cool thing is that Nix will do everything to **maximise the chances of having reproducible build artefacts**. Whether it is a shell script or a full OS.

On top of that, Nix is also a **programming language** and a **package manager**.

> **Important**
>
> Nix is **not** an operating system, there's **NixOS** for that.

An opinionated definition

Nix is **loyal** and **predictable**, ***for sure!***

Hands-on

- By the end of this talk, you’ll have access to a **public Git repository**[1] where each commit corresponds to a step in the process of adopting Nix. Each commit is also attached to a branch named after the step it represents.
- You will be able to clone it and follow along at your own pace, at home or while commuting...
- Or simply follow it online by using the GitHub interface to check the changes commit after commit, step after step.
- We will start with a very trivial scenario using a shell script, and we will slowly increase the complexity of the use cases.
- The last step will show you how to build a complete operating system with a single Nix file and run it in a virtual machine.

[1] https://github.com/drupol/nix-hands-on

In that case, without further ado,
let’s get our hands dirty, **for sure**!

Well… it is not totally true because all the terminal interactions are pre-recorded to avoid unpleasant surprises during the talk.

Step 1: Shell script

Step 1: Shell script

Shell

```
#!/usr/bin/env bash

curl -L -s https://www.php.net/releases/active.php \
| jq -r '[.[][] | .version] | last' \
| cowsay
```

- **Problem:** It relies on some tools being installed on the host system.

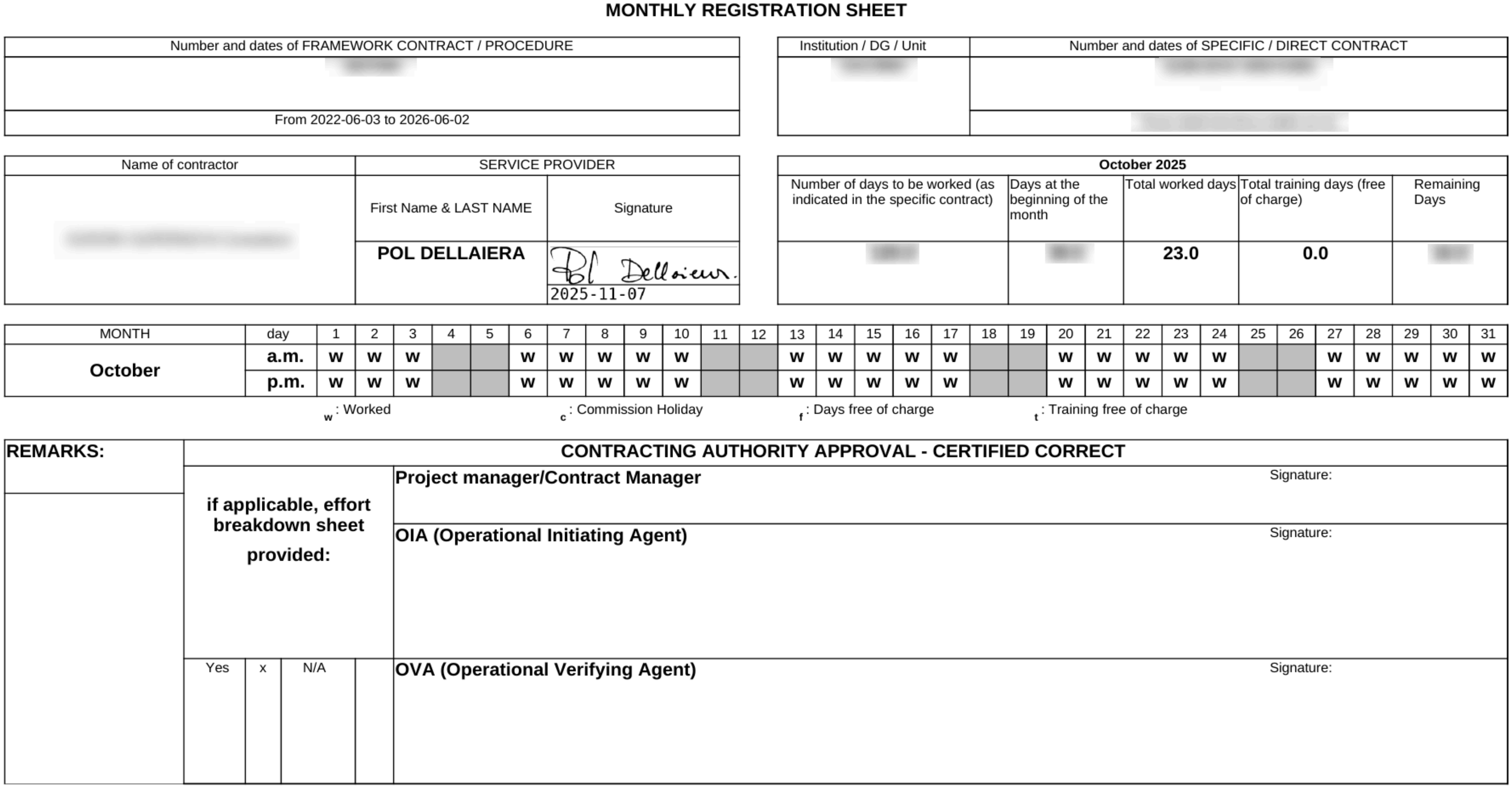
Step 2: Portable Shebang

Step 2: Portable Shebang

```shell
#!/usr/bin/env nix-shell
#! nix-shell -i bash --pure
#! nix-shell -p bash cacert curl jq cowsay
#! nix-shell -I nixpkgs=https://github.com/NixOS/nixpkgs/archive/ace24a96b5c7932a4955b151aa8b37e4cb154496.tar.gz
curl -L -s https://www.php.net/releases/active.php \
| jq -r '[.[][] | .version] | last' \
| cowsay
```

- The script now works on **any** machine with Nix installed.
- Dependencies are defined within the script itself.
- No need to install PHP globally anymore!
- Share your scripts without worrying about the target system.
- **Use Case**: A script for signing monthly timesheets

Step 2: Portable Shebang



MONTHLY REGISTRATION SHEET

Number and dates of FRAMEWORK CONTRACT / PROCEDURE
[redacted]
From 2022-06-03 to 2026-06-02

Institution / DG / Unit	Number and dates of SPECIFIC / DIRECT CONTRACT
[redacted]	[redacted]
	[redacted]

Name of contractor	SERVICE PROVIDER	
[redacted]	First Name & LAST NAME	Signature
	POL DELLAIERA	Pol Dellaiera 2025-11-07

October 2025				
Number of days to be worked (as indicated in the specific contract)	Days at the beginning of the month	Total worked days	Total training days (free of charge)	Remaining Days
[redacted]	[redacted]	**23.0**	**0.0**	[redacted]

MONTH	day	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
October	**a.m.**	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**
	p.m.	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**			**w**	**w**	**w**	**w**	**w**

w : Worked   c : Commission Holiday   f : Days free of charge   t : Training free of charge

REMARKS:	**CONTRACTING AUTHORITY APPROVAL - CERTIFIED CORRECT**				
	if applicable, effort breakdown sheet provided:			**Project manager/Contract Manager**	Signature:
				OIA (Operational Initiating Agent)	Signature:
	Yes	x	N/A	**OVA (Operational Verifying Agent)**	Signature:

Step 3: Development shells

- A **development shell** is an **ephemeral environment** with **all the tools you need** for development. Once the shell is closed, the environment is gone.
- Introduction of `shell.nix`.
- Defines a reproducible development environment with PHP and Composer.

Step 4: Multiple shells

Step 4: Multiple shells

- We add a development shell for GO.
- `shell.nix` now returns an attribute set of shells.
- Usage:
  - `nix-shell --attr php`
  - `nix-shell --attr go`

Step 5: Migrating to Flakes

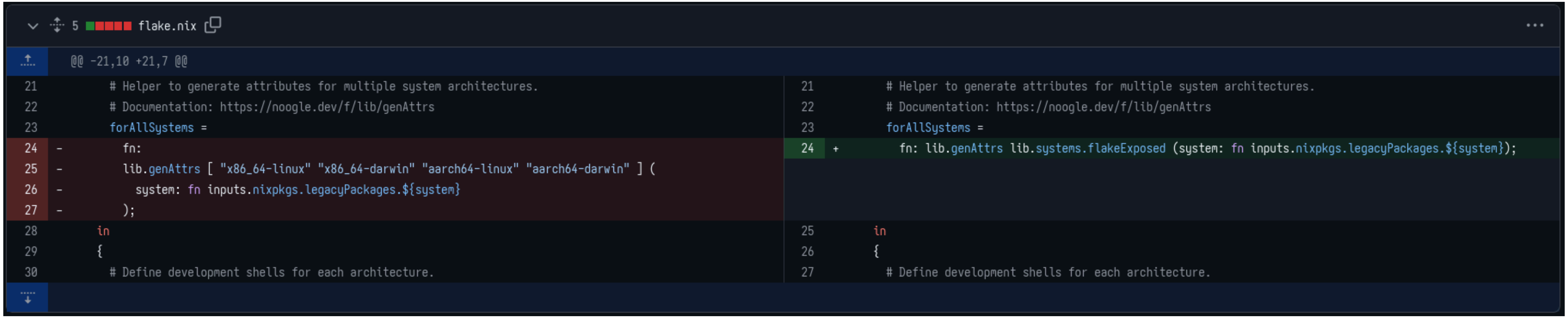
Step 5: Migrating to Flakes

- **Flake**: A framework dictating how to structure a Nix project.
- Explicitly enable the experimental feature flags
- Conversion to a `flake.nix` project.
- Unlike previous steps, locking dependencies happens in a separate file, `flake.lock`.
- Unified command-line interface:
  - `nix develop .#php`
  - `nix develop .#go`

Step 6 & 7: Multiple System Support

Step 6 & 7: Multiple System Support

- **Step 6**: Validating compatibility with multiple architectures (`x86_64-linux`, `aarch64-darwin`, etc.).
- **Step 7**: Using `nixpkgs` default systems list to avoid hard-coding.



5 🟩🟥🟥🟥🟥 flake.nix

```
@@ -21,10 +21,7 @@
```

```
      # Helper to generate attributes for multiple system architectures.
      # Documentation: https://noogle.dev/f/lib/genAttrs
      forAllSystems =
-       fn:
-       lib.genAttrs [ "x86_64-linux" "x86_64-darwin" "aarch64-linux" "aarch64-darwin" ] (
-         system: fn inputs.nixpkgs.legacyPackages.${system}
-       );
    in
    {
      # Define development shells for each architecture.
```

```
      # Helper to generate attributes for multiple system architectures.
      # Documentation: https://noogle.dev/f/lib/genAttrs
      forAllSystems =
+       fn: lib.genAttrs lib.systems.flakeExposed (system: fn inputs.nixpkgs.legacyPackages.${system});
    in
    {
      # Define development shells for each architecture.
```

Step 8: Using a framework

Step 8: Using a framework

- Introduction of the `flake.parts` framework.
- Brings modularity to Flakes.
- Abstracts boilerplate (like the `system` iteration logic).
- Makes configuration manageable and scalable.

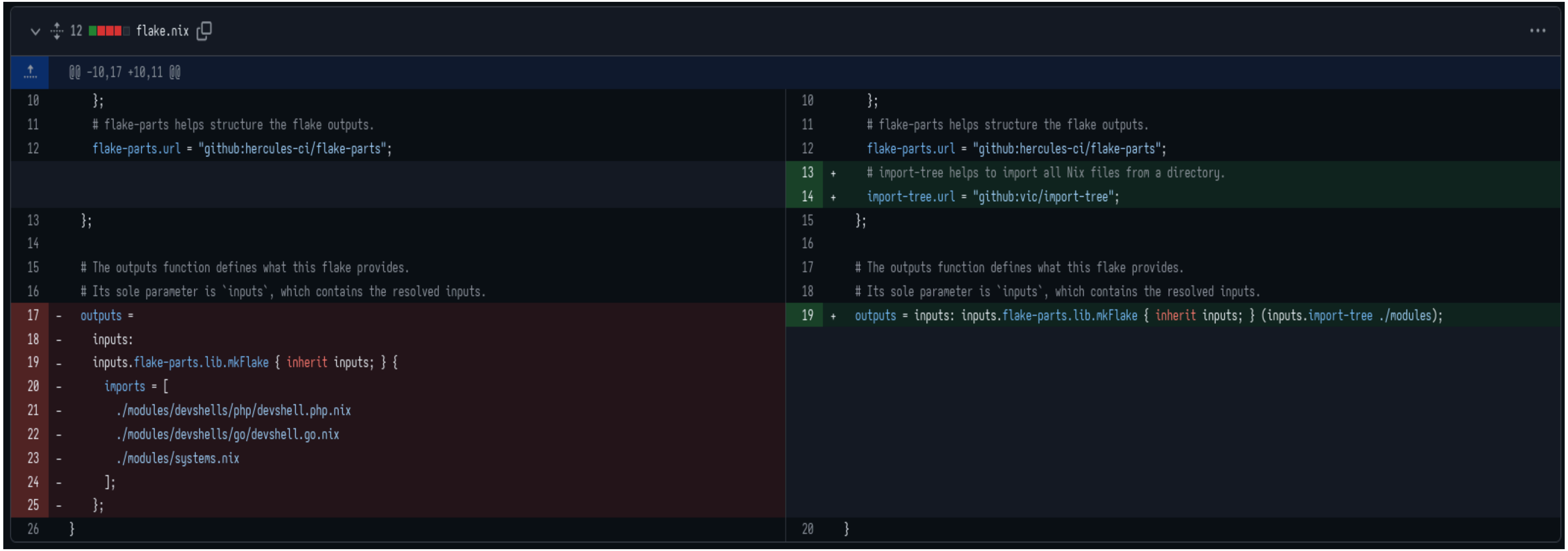
terminal

```
❯ git diff step7..step8 --stat -w ./flake.nix
flake.nix | 25 ++++++++++++++-----------
1 file changed, 14 insertions(+), 11 deletions(-)
```

Step 9 & 10: Modules & Autoloading

Step 9 & 10: Modules & Autoloading

- **Step 9**: Splitting `flake.nix` into smaller modules.
- **Step 10**: Using `vic/import-tree` to automatically load modules from the `modules/` directory.



```
@@ -10,17 +10,11 @@
```

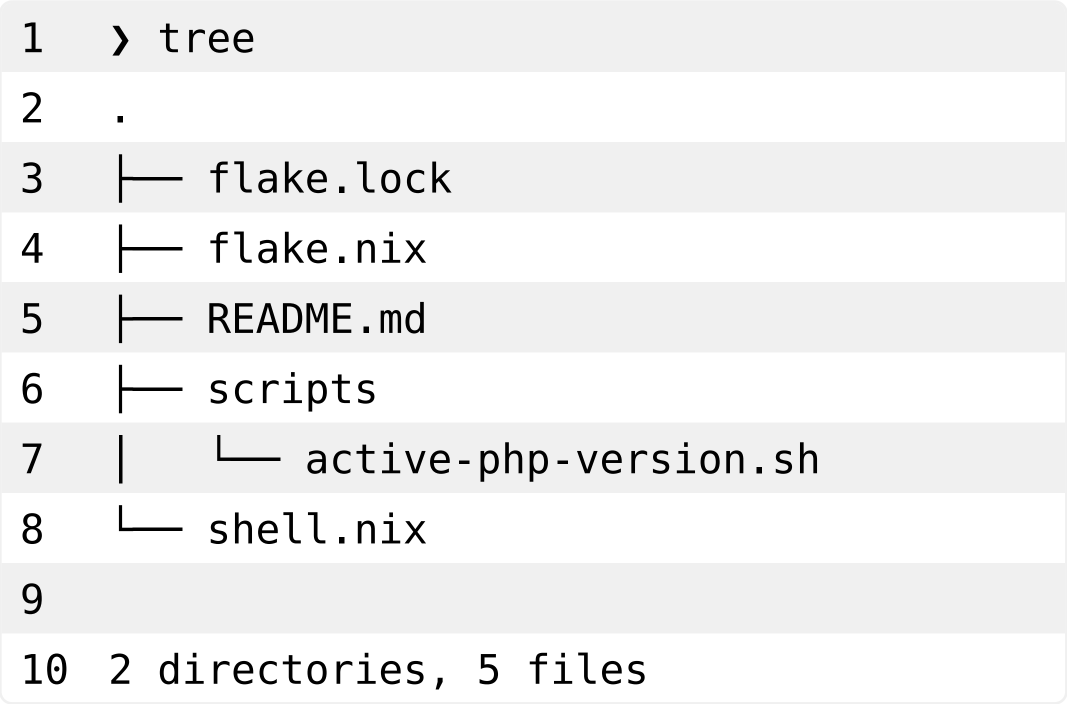
```
    };
    # flake-parts helps structure the flake outputs.
    flake-parts.url = "github:hercules-ci/flake-parts";

  };

  # The outputs function defines what this flake provides.
  # Its sole parameter is `inputs`, which contains the resolved inputs.
- outputs =
-   inputs:
-   inputs.flake-parts.lib.mkFlake { inherit inputs; } {
-     imports = [
-       ./modules/devshells/php/devshell.php.nix
-       ./modules/devshells/go/devshell.go.nix
-       ./modules/systems.nix
-     ];
-   };
}
```

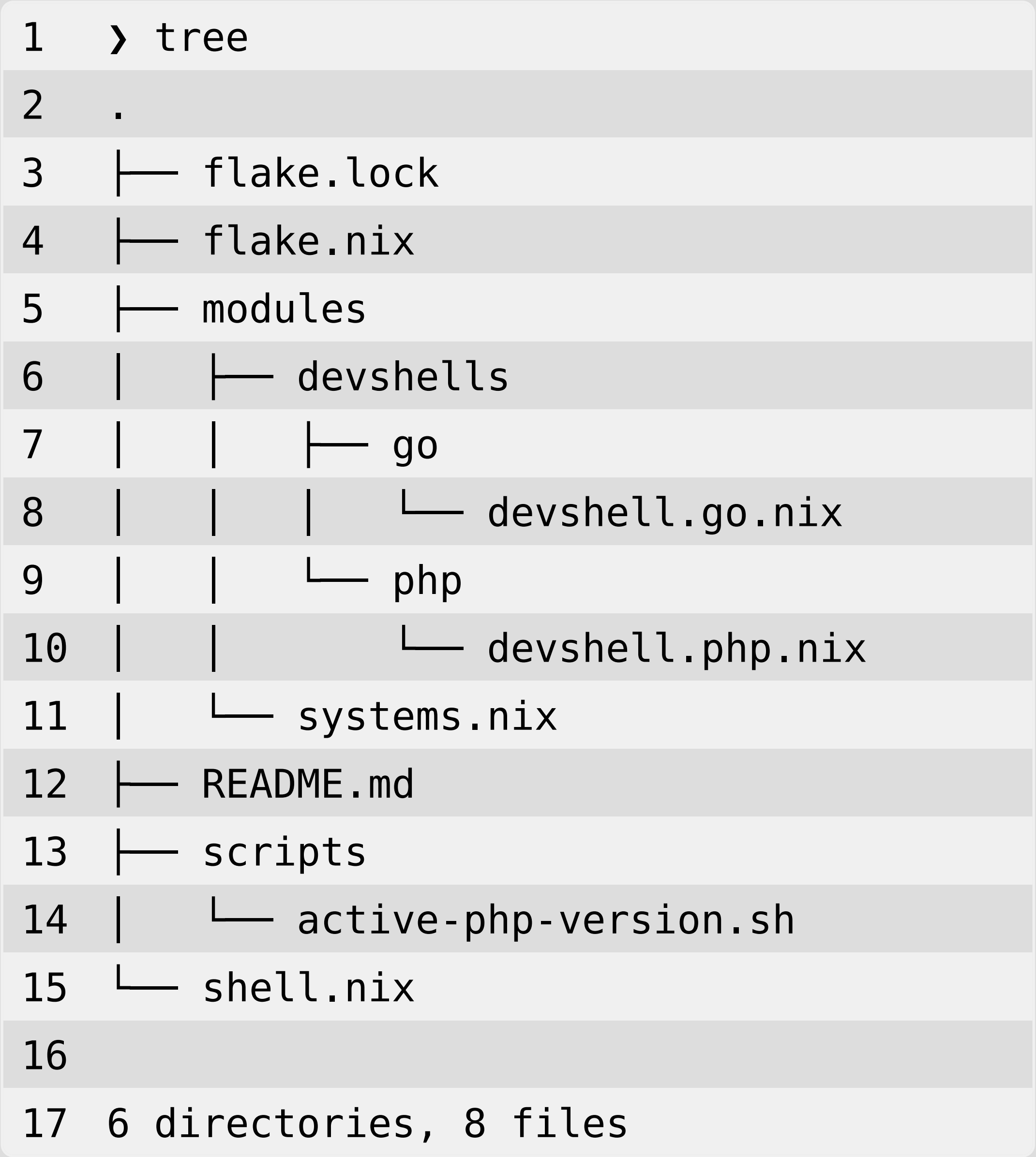
```
    };
    # flake-parts helps structure the flake outputs.
    flake-parts.url = "github:hercules-ci/flake-parts";
+   # import-tree helps to import all Nix files from a directory.
+   import-tree.url = "github:vic/import-tree";
  };

  # The outputs function defines what this flake provides.
  # Its sole parameter is `inputs`, which contains the resolved inputs.
+ outputs = inputs: inputs.flake-parts.lib.mkFlake { inherit inputs; } (inputs.import-tree ./modules);
}
```



```
❯ tree
.
├── flake.lock
├── flake.nix
├── README.md
├── scripts
│   └── active-php-version.sh
└── shell.nix

2 directories, 5 files
```

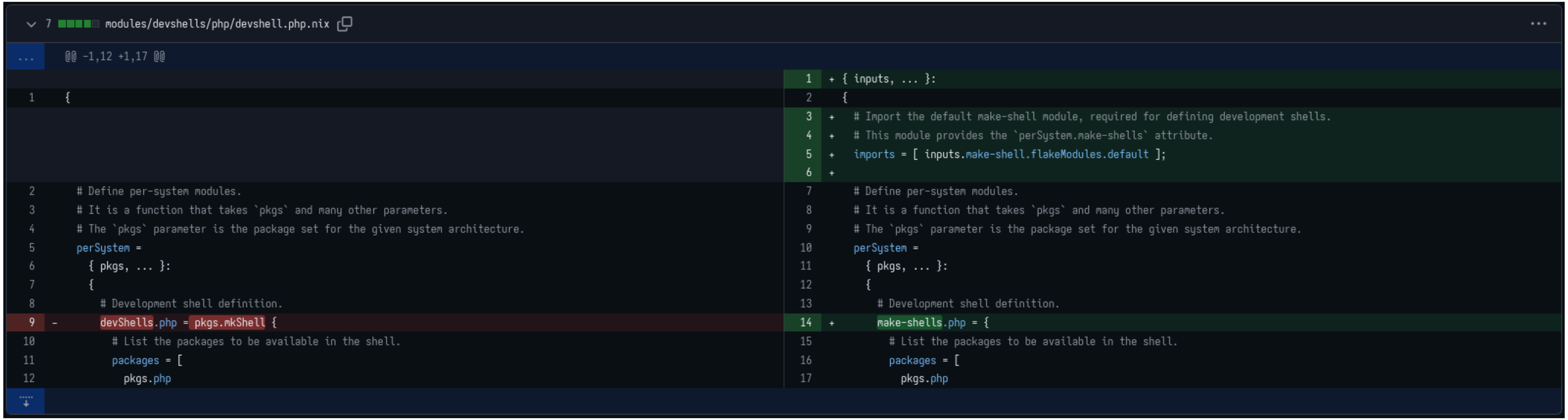


```
❯ tree
.
├── flake.lock
├── flake.nix
├── modules
│   ├── devshells
│   │   ├── go
│   │   │   └── devshell.go.nix
│   │   └── php
│   │       └── devshell.php.nix
│   └── systems.nix
├── README.md
├── scripts
│   └── active-php-version.sh
└── shell.nix

6 directories, 8 files
```

Step 11: Declarative Shells

- Adopting the `make-shell` component from `flake.parts`.
- Defining development shells becomes cleaner and more declarative, no more function calls.
- Merges configurations from multiple modules effectively.



7 modules/devshells/php/devshell.php.nix

```
@@ -1,12 +1,17 @@
```

```
  {
    # Define per-system modules.
    # It is a function that takes `pkgs` and many other parameters.
    # The `pkgs` parameter is the package set for the given system architecture.
    perSystem =
      { pkgs, ... }:
      {
        # Development shell definition.
-       devShells.php = pkgs.mkShell {
          # List the packages to be available in the shell.
          packages = [
            pkgs.php
```

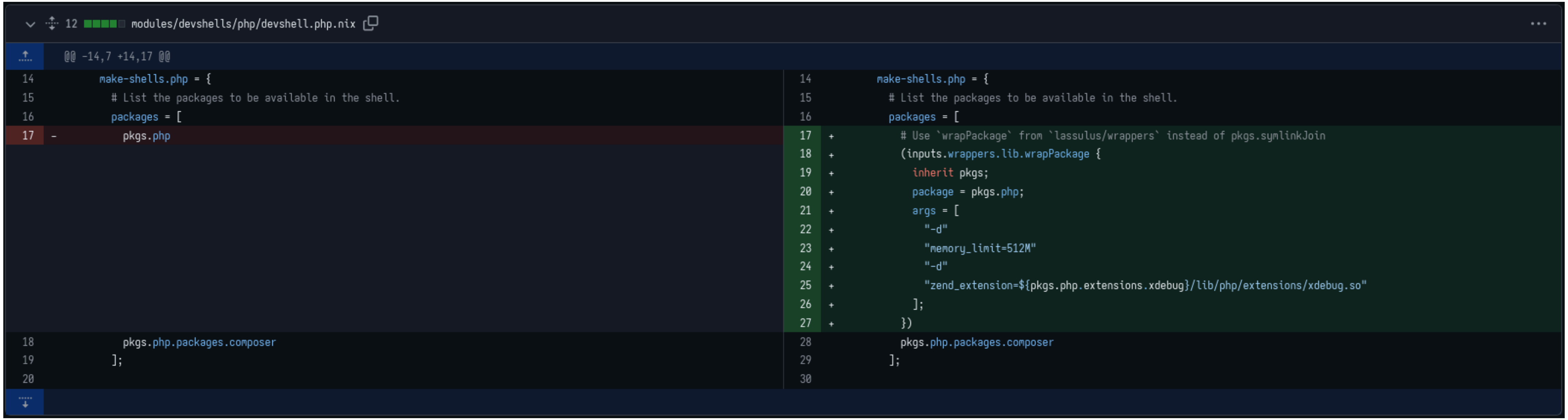
```
+ { inputs, ... }:
  {
+   # Import the default make-shell module, required for defining development shells.
+   # This module provides the `perSystem.make-shells` attribute.
+   imports = [ inputs.make-shell.flakeModules.default ];
+
    # Define per-system modules.
    # It is a function that takes `pkgs` and many other parameters.
    # The `pkgs` parameter is the package set for the given system architecture.
    perSystem =
      { pkgs, ... }:
      {
        # Development shell definition.
+       make-shells.php = {
          # List the packages to be available in the shell.
          packages = [
            pkgs.php
```

Step 12: More Shells!

- We add shells for:
  - Rust
  - Python
  - Node.js
- All co-exist in the same project, runable with:
  - `nix develop .#rust`
  - `nix develop .#python`
  - `nix develop .#node`
- or without *cloning* anything:
  - `nix develop github:drupol/nix-hands-on#rust`
- **Use Case**:
  - Polyglot development environments always available
  - Onboarding new team members with minimal setup

Step 13 & 14: Wrappers

- **Step 13**: Manual wrapping, interesting for learning purposes.
- **Step 14**: Using `lassulus/wrappers` library for a cleaner implementation.
- Useful for enforcing default arguments or config without modifying the upstream package.



12 modules/devshells/php/devshell.php.nix

```diff
@@ -14,7 +14,17 @@
      make-shells.php = {
        # List the packages to be available in the shell.
        packages = [
-         pkgs.php
+         # Use `wrapPackage` from `lassulus/wrappers` instead of pkgs.symlinkJoin
+         (inputs.wrappers.lib.wrapPackage {
+           inherit pkgs;
+           package = pkgs.php;
+           args = [
+             "-d"
+             "memory_limit=512M"
+             "-d"
+             "zend_extension=${pkgs.php.extensions.xdebug}/lib/php/extensions/xdebug.so"
+           ];
+         })
          pkgs.php.packages.composer
        ];

```

Step 15: Packaging

Step 15: Packaging

- Transforming our initial script into a Nix Package.
- The script is now a build artefact which can be built with `nix build` or run with `nix run`.
- To execute it: `nix run .#active-php-version`

Step 16: Packaging Binaries

Step 16: Packaging Binaries

- Adding `nodejs14-bin` package.
- Packaging pre-compiled binaries.
- Run without installing: `nix run .#nodejs14-bin -- --version`.

Step 17: Consuming our Packages

- Creating a `node14` development shell that **uses** our custom `nodejs14-bin` package.
- Demonstrates how to consume your own flake outputs within the same project.

Step 18: Formatting

Step 18: Formatting

Skipping this one, not really interesting for the demo.

Step 19: Containers

- Building an OCI-compliant container image with `dockerTools`.
- Contains our custom package.
- Bit-for-bit reproducible image.
- **Use Case**:
  - Providing development environments in containerized setups
  - Providing consistent runtime environments in CI/CD pipelines

Step 20: NixOS Configuration

Step 20: NixOS Configuration

- Defining a full operating system configuration.
- Can be deployed to a real machine or run in a VM.
- `nixos-rebuild switch --flake .#my-custom-config`

Conclusion

Conclusion

- We started with a trivial script.
- We saw how to make development shells
- We saw how to create custom scripts and packages
- We saw how to build container images including our packages
- We ended with a reproducible operating system.
- **Nix** scales from a single file to a whole infrastructure.
- There is so much more to explore!

Questions