

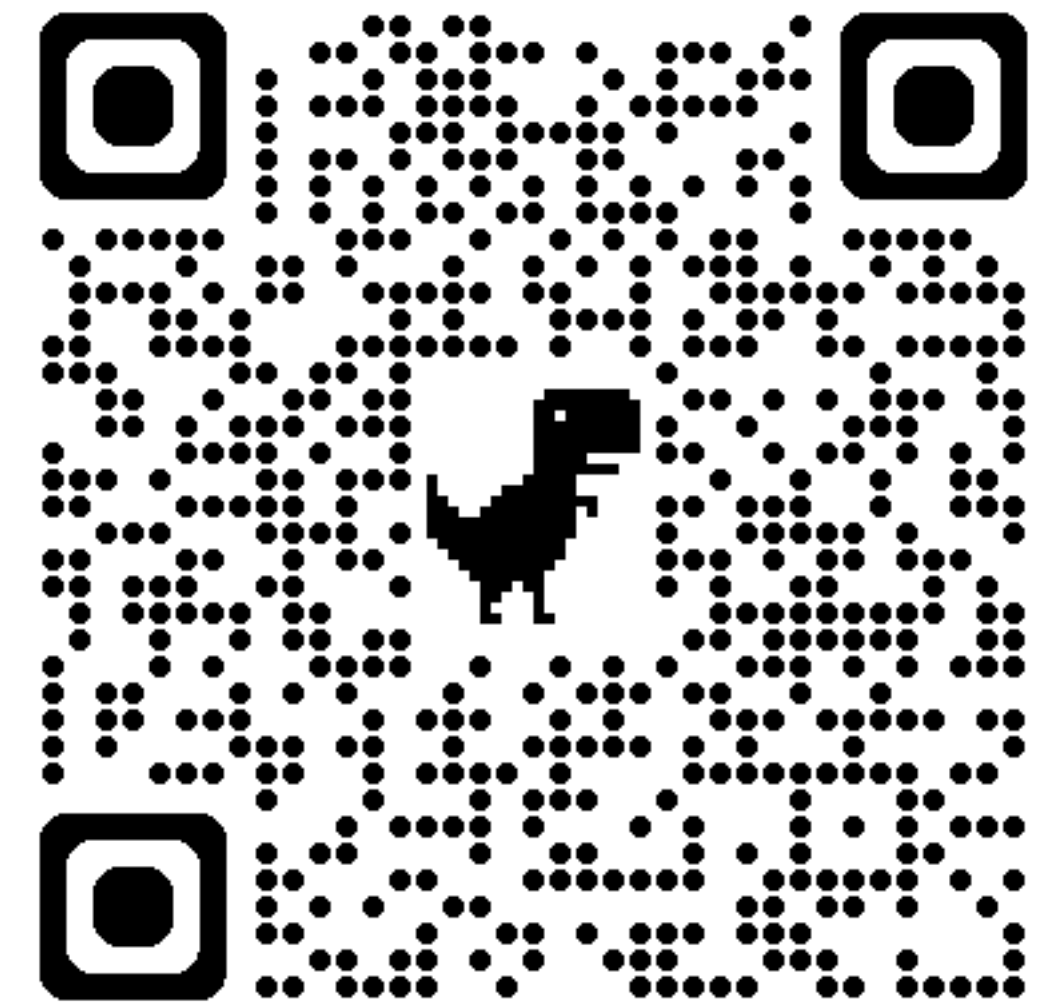
Nix: Sandbox and Reproducible Builds

Daniel Hinojosa



In this Presentation

- What is Nix?
- Installing
- Creating Scripts
- Navigating `nixpkgs`
- Pinning
- Nix Language
- Creating Derivations
- Nix Store
- Docker
- Nix OS



Slides and Material: <https://github.com/dhinojosa/nix-sandbox-reproducible-builds>

What is Nix?



What is Nix?

- Nix is a powerful package manager and build system, known for its unique approach to package and configuration management.
- It's primarily used in Unix-like systems and has several distinctive features:
 - **Functional Package Management**
 - Reliable and Reproducible
 - Isolated Environments
 - Rollbacks and Atomic Upgrades
 - Declarative Configuration
 - Multi-user Profiles





Globally



Per Project

Diff between Nix and Docker?

Differences between Nix and Docker

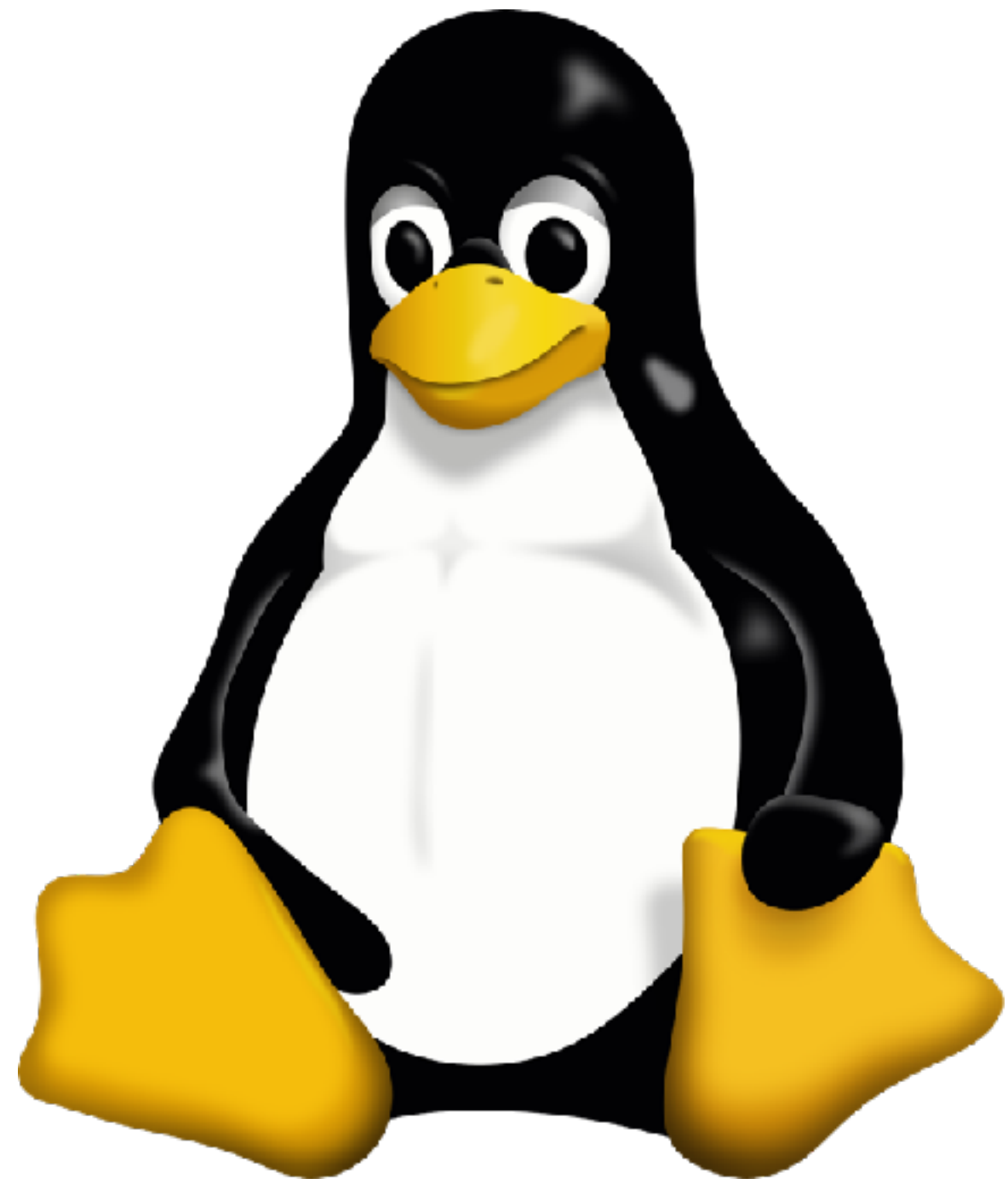
| Feature | Nix | Docker |
|----------------|--|---|
| Primary Goal | Reproducibility of builds and environments | Containerized application deployment |
| Technology | Functional package manager | Containerization platform |
| Isolation | None (processes share the host OS) | OS-level process and resource isolation |
| Dependencies | Declarative package management | Bundled with application in containers |
| Portability | Cross-platform without containers | Containers run anywhere Docker is supported |
| Resource Usage | Lightweight, host-level | More resource-intensive (container overhead) |
| Learning Curve | Steep | Moderate |
| Use Cases | Package management, CI/CD, reproducibility | App isolation, microservices, cloud deployments |

Diff between Nix and DevContainers?

Difference between Nix and DevContainers

| Feature | Nix | Dev Containers |
|---------------------|---|---|
| Primary Goal | Reproducibility, system-level control | Containerized development environments |
| Tooling | Nix package manager | Docker, VS Code |
| Configuration | Declarative Nix language | devcontainer.json, Dockerfiles |
| Isolation | Process-level isolation | Container-based isolation |
| Learning Curve | Steep | Moderate |
| Platform Dependency | None (cross-platform) | Docker + VS Code |
| Use Case | Package management, CI/CD, local environments | Development-only containerized environments |

Where can it be installed?





Installing



Standard Installation

- The following will run the installer interactively and with explanation
- Performs the default "type" of install for your platform:
 - Single-user on Linux
 - Multi-user on macOS

```
$ curl -L https://nixos.org/nix/install | sh
```

MacOS installs too complex to qualify as single-user, so this type is no longer supported on macOS.

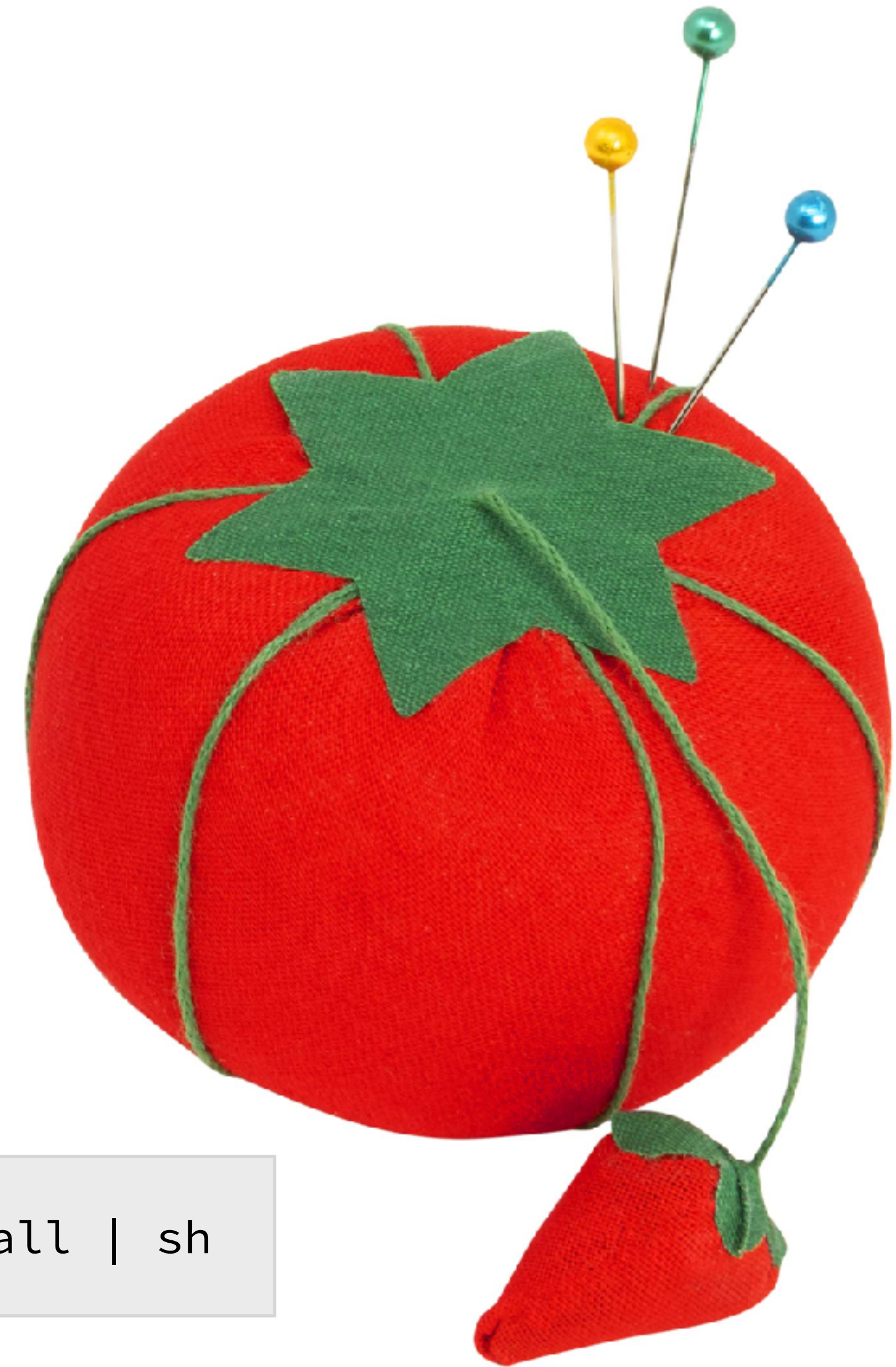
Linux Installation

```
$ curl -L https://nixos.org/nix/install | sh -s -- --daemon
```

Pinned Nix Installations

- Version-specific installation URLs for all Nix versions since 1.11.16 can be found at <http://releases.nixos.org>
- The corresponding SHA-256 hash can be found in the directory for the given version. Where `<version>` is the pinned version you would like.

```
$ curl -L https://releases.nixos.org/nix/nix-<version>/install | sh
```



Installing from a Tarball

- You can also download a binary tarball that contains Nix and all its dependencies.
- This is what the installation script at <https://nixos.org/nix/install> does automatically
- Unpack it somewhere (e.g. in /tmp), and then run the script named install inside the binary tarball

```
$ cd /tmp
$ tar xvj nix-1.8-x86_64-darwin.tar.bz2
$ cd nix-1.8-x86_64-darwin
$ ./install
```

Determining which Nix you have

```
$ nix --version
```

Which will return something similar to the following

```
nix (Nix) 2.18.1
```


Nixpkgs

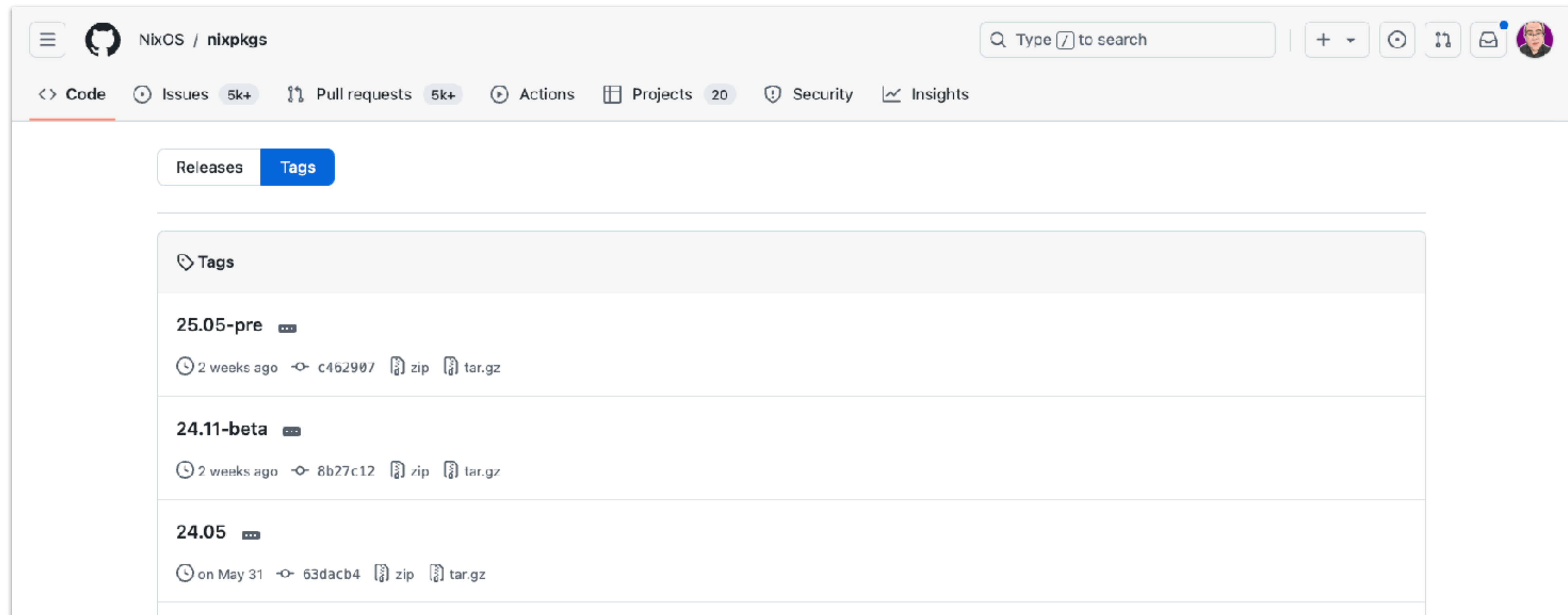


What are `nixpkgs`?

- Comprehensive list of packages for the Nix Package manager
- Each package has an associated specific hash
- Ideal for users and developers who need a consistent and reproducible environment across various platforms,

Where are these `nixpkgs`?

- Nix Packages are all a repository, located at <https://github.com/NixOS/nixpkgs>
- Each tagged version is located at <https://github.com/NixOS/nixpkgs/tags>



Demo: NixPkgs



- Let's look at nixpkgs online and by downloading the version

Using the Shell



`nix-env -u '*'`

What are `nixpkgs`?

- Comprehensive list of packages for the Nix Package manager
- Each package has an associated specific hash
- Ideal for users and developers who need a consistent and reproducible environment across various platforms,

Trying some commands

```
$ cowsay no can do  
The program 'cowsay' is currently not installed.
```

```
$ echo no chance | lolcat  
The program 'lolcat' is currently not installed.
```


Trying some commands

- Comprehensive list of packages for the Nix Package manager
- Each package has an associated specific hash
- Ideal for users and developers who need a consistent and reproducible environment across various platforms,

```
$ nix-shell -p cowsay lolcat
```

Demo: Nix-Store



- Let's show what is in the nix-store

The Nix Store



Demo: Nix-Store



- Let's show what is in the nix-store

Real World Applications



Let's Stop and Think

- Have you ever only required `gcLOUD`, `kubectL` for one application?
- Have you ever needed Helm 2 for one applications, but Helm 3 for another?
- Have you ever got stuck with other version conflicts and you just want a certain set up per project?

When you realize that waiting for the waiter makes you the waiter



Using the REPL



nix-repl

Use nix repl to evaluate Nix expressions interactively (by typing them on the command line)

```
$ nix repl
Welcome to Nix 2.13.3. Type :? for help.

nix-repl> 1 + 2
3
```


Using Instantiate



nix-instantiate

- Use `nix-instantiate --eval` to evaluate the expression in a Nix file.
- `--eval` is required to evaluate the file and do nothing else.
- If `--eval` is not present, it is expecting that the file will return a Derivation (which we will talk about soon)

```
$ echo 1 + 2 > file.nix
$ nix-instantiate --eval file.nix
3
```


nix-instantiate

If no name is specified after `--eval`, Nix will look for the *default.nix* file

```
$ echo 1 + 2 > default.nix
$ nix-instantiate --eval
3
```


The Nix *Functional* Language



Nix Language Contents

- Using the Nix language in practice entails multiple things
 - **Language:** syntax and semantics
 - **Libraries:** `builtins` and `pkgs.lib`
 - **Developer tools:** testing, debugging, linting, formatting, ...
 - **Generic build mechanisms:** `stdenv.mkDerivation`, trivial builders, ...
 - **Composition and configuration mechanisms:** `override`, `overrideAttrs`, `overlays`, `callPackage`, ...
 - **Ecosystem-specific packaging mechanisms:** `buildGoModule`, `buildPythonApplication`, ...
 - **NixOS module system:** `config`, `option`, ...

Basics

Language Basics



Whitespace

- Whitespace is used to delimit lexical tokens, where required.
- It is otherwise insignificant:

```
let  
  x = 1;  
  y = 2;  
in x + y
```

- It is the same as...

```
let x=1;y=2;in x+y
```


Comments

```
# Comments can be the full line with a hash
```

```
/* Multi line comments  
   can be done accordingly  
   in Java doc style  
*/
```


Strings

```
"Hello, World"
```


Special Strings

- Strings can span multiple lines.
- The special characters " and \ and the character sequence \${ must be escaped by prefixing them with a backslash (\).
- Newlines, carriage returns and tabs can be written as \n, \r and \t, respectively.

Indentation Strings

- This kind of string literal intelligently strips indentation from the start of each line.
- To be precise, it strips from each line a number of spaces equal to the minimal indentation of the string as a whole (disregarding the indentation of empty lines).
- For instance, the first and second line are indented two spaces, while the third line is indented four spaces.

```
''  
  This is the first line.  
  This is the second line.  
    This is the third line.  
''
```

- Thus, two spaces are stripped from each line, so the resulting string is:

```
"This is the first line.\nThis is the second line.\n  This is the third line.\n"
```


Booleans

```
true  
false
```


Nulls

```
null
```


String Interpolation

- String interpolation can be done with `${}`

```
let  
  name = "Nix";  
in  
"hello ${name}"
```

- This will return with the following:

```
"hello Nix"
```


Use Only Strings in Interpolation

- Only character strings or values that can be represented as a character string are allowed

```
let
  x = 1;
in
"${x} + ${x} = ${x + x}"
```

- This will return with the following error:

```
error: cannot coerce an integer to a string
      at «string»:4:2:
```

```
3| in
4| "${x} + ${x} = ${x + x}"
  | ^
5|
```

When there are no braces, it's a variable

- `$out` is a special variable that will store in `/nix/store`
- `${out}` is an interpolation of what is in `let`
- `out` in `let` is just a variable declaration

```
let  
  out = "Nix";  
in  
"echo ${out} > $out"
```


Lists

- Lists are formed enclosing *whitespace-separated list of values* between *square brackets*
- The following includes a number, a path, a string, a function with an attribute. Functions have to be surrounded by parenthesis ()

```
[ 123 ./foo.nix "abc" (f { x = y; }) ]
```

Creating a List and Mapping

```
let
  # Generate a list [1, 2, 3, 4, 5]
  numbers = [ 1 2 3 4 5 ];

  # Map each element (e.g., multiply by 2)
  doubledNumbers = builtins.map (x: x * 2) numbers;
in
  doubledNumbers
```


Attribute Sets

- Name, Value pairs in a set of curly brackets { }
- The attribute name must be a valid String or an identifier
- An identifier must start with a letter (a–z, A–Z) or underscore (_), and can otherwise contain letters (a–z, A–Z), numbers (0–9), apostrophes ('), or dashes (–)

Attribute Set Example

```
{  
  x = 123;  
  text = "Hello";  
  y = f { bla = 456; };  
}
```

- Use a . to access the value. The following resolves to "Foo"

```
{ a = "Foo"; b = "Bar"; }.a
```


Getting a default value from an attribute

- Provide a default value in an attribute selection using the or keyword.
- In the following both will resolve to "Xyzzy"

```
{ a = "Foo"; b = "Bar"; }.c or "Xyzzy"
```

```
{ a = "Foo"; b = "Bar"; }.c.d.e.f.g or "Xyzzy"
```

Attribute Names can be double quoted

- You can use arbitrary double-quoted strings as attribute names
- Both of the following will be evaluated to 123

```
{ "$!@#?" = 123; }. "$!@#?"
```

```
let bar = "bar"; in
```

```
{ "foo ${bar}" = 123; }. "foo ${bar}"
```


Attribute Names support interpolation

- Here interpolation can be used in both the attribute name and the value
- Both of the following will resolve to 123

```
let bar = "foo"; in  
{ foo = 123; }.${bar}
```

```
let bar = "foo"; in  
{ ${bar} = 123; }.foo
```

null prevents attribute name being added to the Set

- When an attribute name inside a set declaration evaluates to `null`, that attribute is simply not added to the set
- Reason is that `null` cannot be coerced to a string
- The following will evaluate to `{}`, an empty attribute set, if `foo` evaluates to `false`.

```
{ ${if foo then "bar" else null} = true; }
```


Recursive Attribute

```
rec {  
  one = 1;  
  two = one + 1;  
  three = two + 1;  
}
```

- The previous evaluates to the following:

```
{ one = 1; three = 3; two = 2; }
```

let..in

- Also called let binding.
- let expressions allow assigning names to values for repeated use

```
let  
  a = 1;  
in  
a + a
```


with

- The with expression allows access to attributes without repeatedly referencing their attribute set.

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
in
with a; [ x y z ]
```

- Where `with a; [x y z]` is the same as `[a.x a.y a.z]`

with out of scope

- In the following, `c=x` would return an error since `x` is not within scope
- The `x` in `b = with a; [x y z]` is valid, since `with` puts `a` in scope

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
in
{
  b = with a; [ x y z ];
  c = x;
}
```


inherit

- inherit is shorthand for assigning the value of a name from an existing scope to the same name in a nested scope.
- It is for convenience to avoid repeating the same name multiple times.

```
let
  x = 1;
  y = 2;
in
{
  inherit x y;
}
```

- Results in, and is equivalent to `x = x; y = y;`

```
{ x = 1; y = 2; }
```

inheriting specific attributes

- It is also possible to inherit names from a specific attribute set with parentheses (`inherit (...) ...`).

```
let
  a = { x = 1; y = 2; };
in
{
  inherit (a) x y;
}
```

- Results in, and is equivalent to `x = a.x; y = a.y`

```
{ x = 1; y = 2; }
```


inherit inside a let

- In a let, it will destructure to the values

```
let  
  inherit ({ x = 1; y = 2; }) x y;  
in [ x y ]
```

- Results in

```
[ 1 2 ]
```

Functions

Common Functions to Use



Functions

Functions are similar to lambdas in many programming languages

Single Argument Function:

```
x: x + 1
```

Multiple Argument Function:

```
x: y: x + y
```

Attribute Sets

Attribute Sets are Key-Value Structures

```
{  
  key1 = "value1";  
  key2 = 42;  
  key3 = [ "list" "of" "values" ];  
  nested = {  
    subkey1 = "nested value";  
  };  
}
```


Attribute Set Arguments in Functions

Arguments can also be attributes, here the argument must be an attribute

```
{ a, b } : a + b
```

Arguments can also be attributes, here the argument must be an attribute

```
{ a, b ? 0 } : a + b
```

Repeated Parameter Arguments

Additional Arguments can also be applied

```
{ a, b, ...}: a + b
```


Named Attribute Set

```
args@{ a, b, ... }: a + b + args.c
```

or

```
{ a, b, ... }@args: a + b + args.c
```

Calling a Single Argument Function

```
let  
  f = x: x + 1;  
in f 1
```


Calling a Single Argument Function as an Attribute

```
let  
  f = x: x.a;  
in  
f { a = 1; }
```

Calling a Single Argument Function by Attribute Name

```
let  
  f = x: x.a;  
  v = { a = 1; };  
in  
f v
```


Calling a Function Inline

```
(x: x + 1) 1
```

Non-Evaluated List of Functions

```
let  
  f = x: x + 1;  
  a = 1;  
in [ f a ]
```

Renders:

```
[ <LAMBDA> 1 ]
```

Nix is lazily evaluated

Evaluation of a List of Functions

```
let  
  f = x: x + 1;  
  a = 1;  
in [ (f a) ]
```

Renders:

```
[ 2 ]
```

Multiple Arguments

- Known as "curried" functions
- Functions are nested, therefore can be applied as down payments

```
x: y: x + y
```

- Applying x with 3 will return a function $y: 3 + y$
- When y is applied with, say, 9 we will get 12

Attribute Set Arguments

- Also known as “keyword arguments” or “destructuring” .
- Nix functions can be declared to require an attribute set with specific structure as argument.

```
{a, b}: a + b
```

- Here we can apply this function with an attribute:

```
let  
  f = {a, b}: a + b;  
in  
f { a = 1; b = 2; }
```


Unexpected Arguments in Attribute Set Argument

- The following will return an error, c is not in the argument attribute set

```
let  
  f = {a, b}: a + b;  
in  
f { a = 1; b = 2; c = 3; }
```

- If you really need c it should be used and possibly be declared in the incoming attribute set

Default Values

- Arguments can be brought in with defaults
- This is denoted by separating the attribute name and its default value with a question mark (?).
- Attributes in the argument are not required if they have a default value.

```
let  
  f = {a, b ? 0}: a + b;  
in  
f { a = 1; }
```

Default Values with an Empty Set

If all the attribute set has a default value, then you can just send an empty attribute set

```
let  
  f = {a ? 0, b ? 0}: a + b;  
in  
f { }
```


Additional Attributes

- Additional attributes are allowed with an ellipsis (...)
- This is referred to in other languages as varargs, repeated parameters, etc.

```
let  
  f = {a, b, ...}: a + b;  
in  
f { a = 1; b = 2; c = 3; }
```

Named Attribute Set Pattern

- Also known as “@ pattern”, “@ syntax”, or “at syntax”.
- An attribute set argument can be given a name to be accessible as a whole.
- This is denoted by prepending or appending the name to the attribute set argument, separated by the at sign (@).

```
{a, b, ...}@args: a + b + args.c
```

```
args@{a, b, ...}: a + b + args.c
```

- How it is applied...

```
let  
  f = {a, b, ...}@args: a + b + args.c;  
in  
f { a = 1; b = 2; c = 3; }
```

Additional Attributes

- Also known as “primitive operations” or “primops”.
- Nix comes with many functions that are built into the language.
- They are implemented in C++ as part of the Nix language interpreter.
- Let's visit the list of functions

Nix Libraries `pkgs.lib`

- The `nixpkgs` repository contains an attribute set called `lib`, which provides a large number of useful functions.
- They are implemented in the Nix language, as opposed to `builtins`, which are part of the language itself.

```
let  
  pkgs = import <nixpkgs> {};  
in  
pkgs.lib.strings.toUpper "lookup paths considered harmful"
```

This is not the idiomatic way to do so, since `<nixpkgs>` can be any version

Nix Libraries `pkgs.lib` Consistently

- The following uses pinning
- This is now forming a typical file in Nix and how to go about using it.

```
let
  nixpkgs = fetchTarball https://github.com/NixOS/nixpkgs/archive/06..b.tar.gz;
  pkgs = import nixpkgs {};
in
pkgs.lib.strings.toUpper "always pin your sources"
```

Using a variable `pkgs`

- In the following we see... a lambda, or function, that accepts `pkgs` as a variable
- It's not particularly typed, so we will have to assume that it is `<nixpkgs>`

```
{ pkgs, ... }:  
pkgs.lib.strings.removePrefix "no " "no true scotsman"
```


Where does pkgs come from?

If you have a file with arguments like pkgs they can come from the command line or another file, like *default.nix*

```
$ nix-instantiate --eval file.nix --arg pkgs 'import <nixpkgs>
{}'
"true scotsman"
```

What if we have `lib` as a variable?

Given the following nix file:

```
{ lib, ... }:  
let  
  to-be = true;  
in  
lib.trivial.or to-be (! to-be)
```

We can call the file with:

```
$ nix-instantiate --eval file.nix --arg lib '(import <nixpkgs>  
{}).lib'
```

Bringing in multiple arguments

You can explicitly bring in multiple arguments in a nix file to avoid any ambiguity

```
{ pkgs, lib, ... }:  
# ... multiple uses of `pkgs`  
# ... multiple uses of `lib`
```


How do we introspect?

`builtins.trace` offers a way to show evaluations as you run them

```
let
  a = 10;
  b = 20;
  c = builtins.toString a;
in
  builtins.trace "What we have ${c}" (a + b)
```

Showing types and docs

`:t` in the nix-repl will try to show the result type of the evaluation

```
:t <expr>
```

`:doc` will show the documentation of any builtin

```
:t <expr>
```

Builtins

Items that are built-in to the languages



Built-in Constants and Values

- Nix builtins offers many constants and value like
 - `systemtime`
 - `currentSystem`
 - `false`, and `true`
 - `nixPath`
 - `nixVersion`
 - `storeDir`

Importing

Bringing in from other sources



import

- `import` takes a path to a Nix file, reads it to evaluate the contained Nix expression, and returns the resulting value.
- If the path points to a directory, the file `default.nix` in that directory is used instead.

```
echo 1 + 2 > file.nix
```

```
import ./file.nix
```


importing a function

Whenever you see additional tokens after a call to import, the value it returns should be a function, and anything that follows are arguments to that function.

```
$ echo "x: x + 1" > file.nix
```

```
import ./file.nix 1
```

Fetchers

Getting binaries for your derivation



Fetchers

- The Nix language provides built-in impure functions to fetch files over the network during evaluation:
 - `builtins.fetchurl`
 - `builtins.fetchTarball`
 - `builtins.fetchGit`
 - `builtins.fetchClosure`
- These functions evaluate to a file system path in the Nix store

Fetchers

Fetches content from a URL

```
builtins.fetchurl "https://github.com/NixOS/nix/archive/7c..ff.tar.gz"
```

Fetches content from a URL and unpacks the tar

```
builtins.fetchTarball "https://github.com/NixOS/nix/archive/7c..ff.tar.gz"
```

View more fetchers in the [NixOS Manual](#)

Paths

How to use paths within Nix



Path

- Paths of course refer to the accessing files from the file system
- Paths need to contain at least one / to be considered a path

Absolute Path System Paths

- The Nix language offers convenience syntax for file system paths.
- Absolute paths always start with a slash (/).

```
/absolute/path
```

Relative File System Paths

- Paths are relative when they contain at least one slash (/) but do not start with one. They evaluate to the path relative to the file containing the expression.
- Assuming we are in `/current/directory`

Resolves to `/current/directory/relative`

```
./relative
```

Resolves to `/current/directory/relative/path`

```
relative/path
```

Paths with a .

- Since relative paths must contain a slash (/) but must not start with one, and the dot (.) denotes no change of directory, the combination ./ specifies the current directory as a relative path.
- Assuming we are in /current/directory, the following resolves to /current/directory

```
./
```


Lookup Paths

- The angle bracket path that represents an actual path

```
<nixpkgs>
```

- Resolves to the location of nixpkgs:

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs
```

Lookup Paths with Directories

- A path can be added to the nixpkgs

```
<nixpkgs/lib>
```

- Resolves to the location of nixpkgs:

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs/lib
```

- Doing this is not recommended since they are impure

Paths and Nix-Store

- When a file is used in string interpolation that file is added to the Nix Store as a side effect

```
$ echo 123 > data
```

```
"${./data}"
```

- Results in

```
"/nix/store/h1qj5h5n05b5d15q4n1drqq8mdg7dhqk-data"
```

- The results here are of the format `/nix/store/<hash>-<name>`

Such interpolated expressions must evaluate to something that can be represented as a character string.
A file system path is such a value, and its character string representation is the corresponding Nix store path

Paths and Interpolation

- Paths can include string interpolation and can themselves be interpolated in other expressions.
- At least one slash (/) must appear before any interpolated expression for the result to be recognized as a path.
- `a.${foo}/b.${bar}` is a division operation.
- `./a.${foo}/b.${bar}` is a path.

Creating a Default File



Creating a Default File

- `default.nix` Nix expression file used to define build or environment configurations.
- Acts as the default entry point for many Nix commands.
- Helps standardize and automate package builds or project setups.
- Works with `nix-shell` and `nix-build`.

Basic *default.nix* file

```
{ pkgs ? import <nixpkgs> {} }:  
  
pkgs.mkShell {  
  buildInputs = [  
    pkgs.neovim  
    pkgs.rustc  
    pkgs.kubectl  
  ];  
}
```

You can then run this with nix-shell

```
nix-shell
```

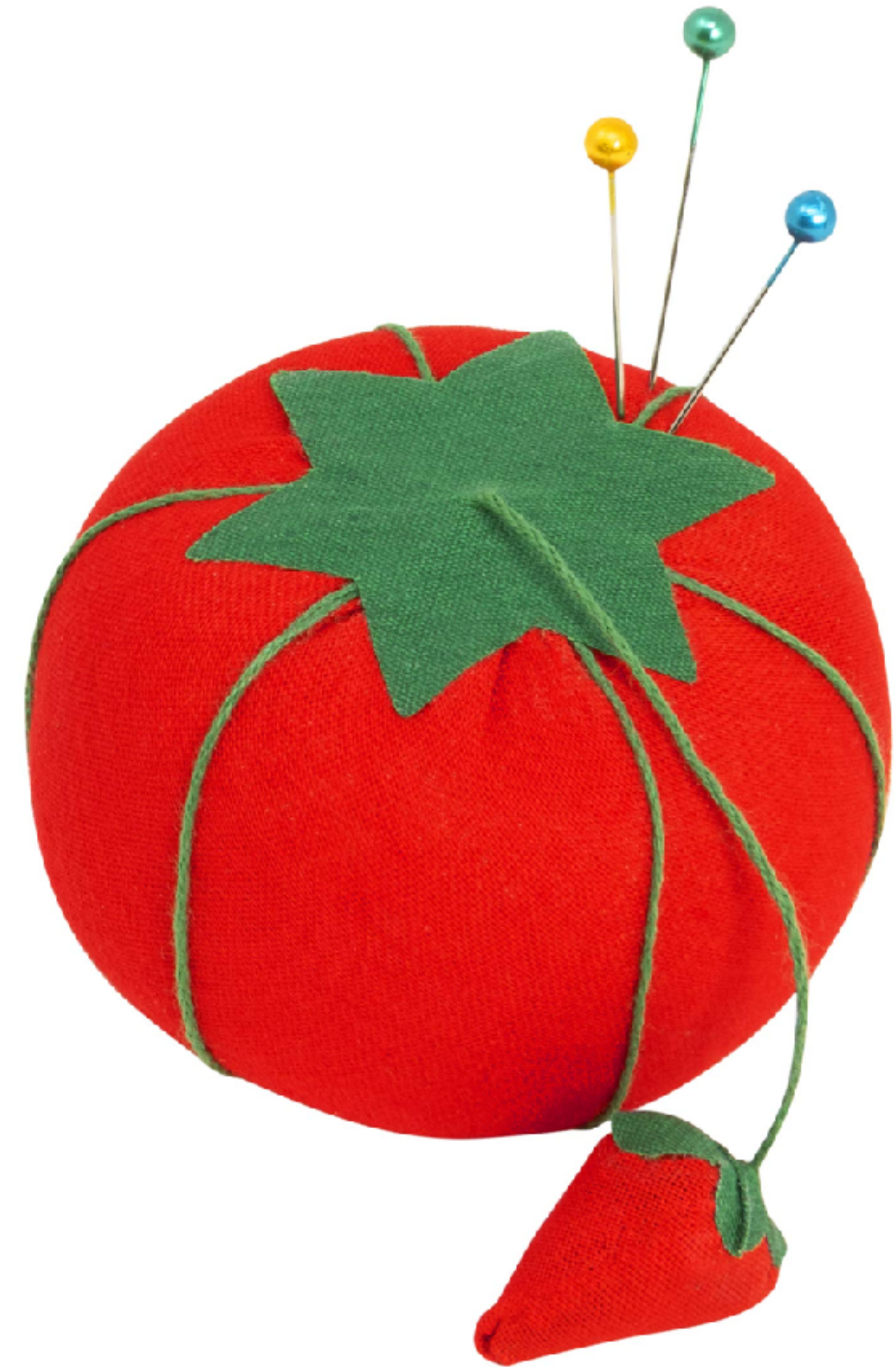

Pinning

Ensuring Repeatable Packages



Pinning Nix

- Pinning in Nix refers to fixing a specific version of the `nixpkgs` repository or other dependencies to ensure a reproducible and stable environment.
- Ensures that builds produce the same results, even if the `nixpkgs` repository evolves.
- Locks dependencies to a known good state, avoiding unexpected breakages due to upstream changes.



Pinning Nix

- Pinning package on a tarball

```
pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/refs/tags/24.05.tar.gz") {}
```

- Pinning package on a git

```
pkgs ? import (fetchGit {  
  url = "https://github.com/NixOS/nixpkgs.git";  
  rev = "a1b2c3d4e5"; # Specific commit hash  
) {}
```


Demo: Pinning



- Let's take a look at two forms of pinning.

Nix Derivations



What is a Nix Derivation?

- An Immutable Build Recipe.
- It takes other packages as inputs, you decide what to do with the inputs, and you build another package.
- You are deriving another package based on other packages.
- **This is analogous to how docker images are derived from a base image.**

Every package in `nixpkgs` is a derivation

- If you download the repository or view the package repository you will see all the applications that are available for you to use download and use in your *default.nix*
- You will find that in nearly all the nix files that they will either be a flake or standard nix file *that will have* `mkDerivation`
- ***The choice is yours; use derivations for your own project, or use derivations to create your own derivations.***

You have been deriving all along

```
FROM jenkins/jenkins:lts

# Switch to root to install Docker CLI
USER root

# Install Docker CLI
RUN apt-get update && \
    apt-get install -y docker.io && \
    apt-get clean

# Install Jenkins Docker plugin
RUN /usr/local/bin/install-plugins.sh docker

# Change back to Jenkins user
USER jenkins
```


An example of a derivation

```
{ pkgs ? import <nixpkgs> {} }:  
  
pkgs.stdenv.mkDerivation {  
  name = "hello-world";  
  src = null;  
  
  buildCommand = '  
    echo "Hello, Nix!" > $out  
  ';  
}
```

Create and use the derivation

- Run nix-build against the default.nix file with the derivation

```
nix-build
```

- View the result

```
cat ./result
```

- View the directory of the result

Demo: Derivation



- Let's take a look at derivation

Derivation Phases

Callbacks within your Derivation



Controlling

- The formal term is *Controlling Phases* - Package builds are split into phases to make it easier to override specific parts of the build (e.g., unpacking the sources or installing the binaries).
- Each phase can be overridden in its entirety either by setting the environment variable `namePhase` to a string containing some shell commands to be executed, or by redefining the shell function `namePhase`.
- When overriding a phase, for example `installPhase`, it is important to start with `runHook`, `preInstall` and end it with `runHook postInstall`, otherwise `preInstall` and `postInstall` will not be run.
- Even if you don't use them directly, it is good practice to do so anyway for downstream users who would want to add a `postInstall` by overriding your derivation.

Demo: Derivation Phases



- Let's take a look at derivation with phases

Nix Channel



What is a Nix Channel?

- An Immutable Build Recipe.
- It takes other packages as inputs, you decide what to do with the inputs, and you build another package.
- You are deriving another package based on other packages.
- **This is analogous to how docker images are derived from a base image.**



Uploading your Derivation



How do upload a derivation?

- Prepare and test your derivation
- Clone nixpkgs directory

```
git clone https://github.com/NixOS/nixpkgs.git  
cd nixpkgs
```

- Add your derivation to the appropriate directory
- Create a pull request
- Respond to your pull request

Managing Packages



nix-env

- You can use it to install, upgrade, and erase packages
- Used to query what packages are installed or are available for installation

Creating Docker Images with Nix



Nix to create Docker images

- Nix can also be used to create consistent Docker images.
- Builds are declarative, ensuring dependencies are explicitly defined.

```
{ pkgs ? import <nixpkgs> {} }:  
pkgs.dockerTools.buildImage {  
  name = "my-docker-image";  
  contents = [ pkgs.hello ];  
  config = {  
    Cmd = [ "hello" ];  
  };  
}
```


Using CI/CD

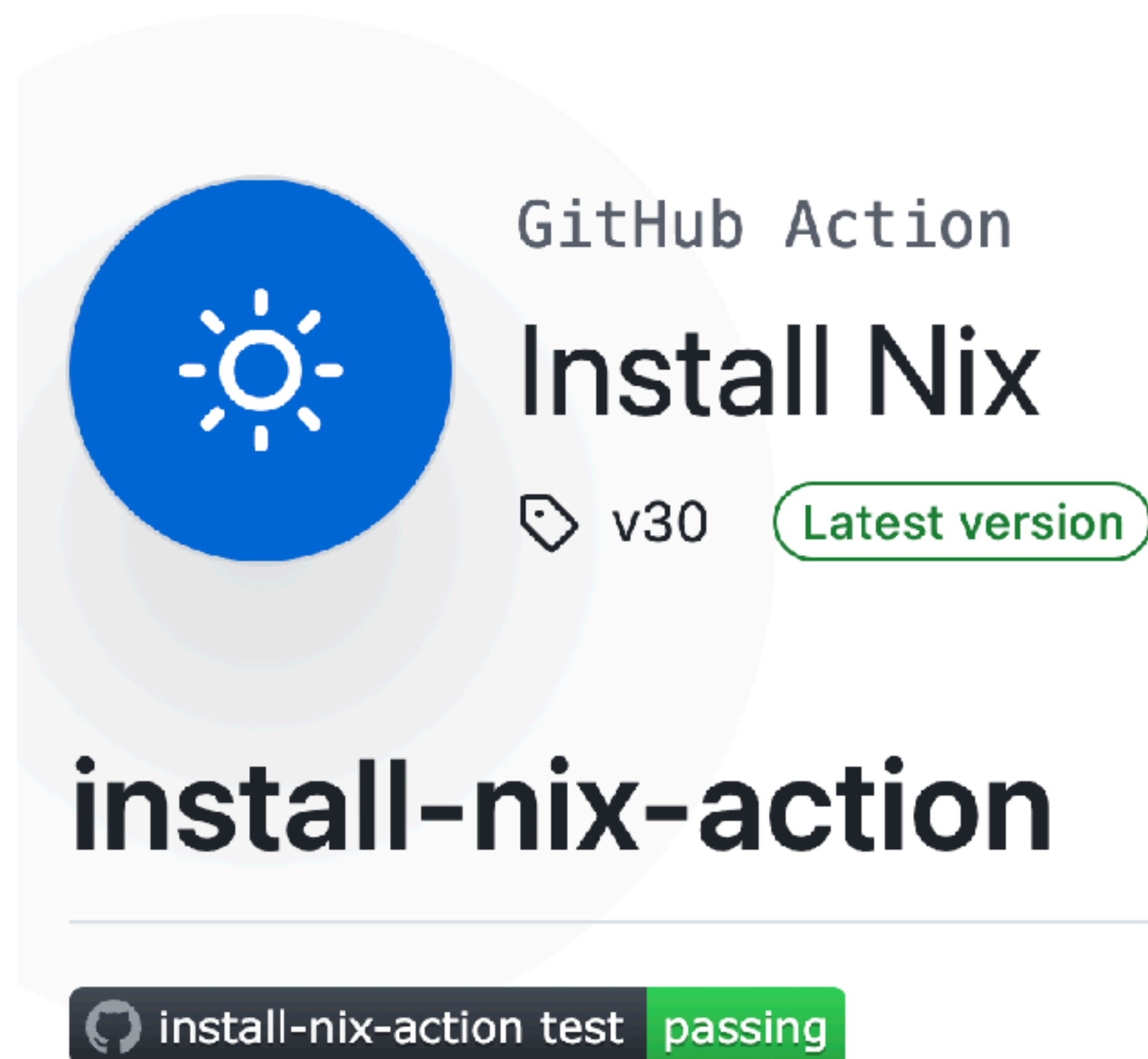


Using CI/CD with Nix

- Your remote agents can run anything, each agent can have its own nix store

```
stages {  
  stage('Setup') {  
    steps {  
      sh '''  
        . /etc/profile.d/nix.sh  
        nix-channel --update  
        '''  
    }  
  }  
  stage('Build') {  
    steps {  
      sh '''  
        . /etc/profile.d/nix.sh  
        nix-build ./default.nix  
        '''  
    }  
  }  
}
```


Github Actions



Installs [Nix](#) on GitHub Actions for the supported platforms: Linux and macOS.

By default it has no nixpkgs configured, you have to set `nix_path` by [picking a channel](#) or [pin nixpkgs yourself](#) (see also [pinning tutorial](#)).

Nix Flakes



What are Nix Flakes?

- An experimental feature that is aimed to improve the main features of Nix derivations
- Flakes aim to make Nix configurations more reproducible, shareable, and composable. Here are the key aspects of Nix Flakes
- Flakes represent an evolution in the Nix ecosystem, aiming to solve long-standing issues around reproducibility and usability.
- They're particularly appealing for developers seeking to create reproducible development environments and for Nix users who need to manage complex system configurations.

What's great about Flakes?

- A flake.nix file offers a uniform schema
- Other flakes can be referenced as dependencies providing Nix language code or other files.
- The values produced by the Nix expressions in flake.nix are structured according to pre-defined use cases.
- References to other flakes can be specified using a dedicated URL-like syntax.
- A flake registry allows using symbolic identifiers for further brevity.
- References can be automatically locked to their current specific version and later updated programmatically.
- A new command line interface, implemented as a separate experimental feature, leverages flakes by accepting flake references in order to build, run, or deploy software defined as a flake.

What's great about Flakes?

- The flake.nix file is checked for schema validity.
- In particular, the metadata fields cannot be arbitrary Nix expressions. This is to prevent complex, possibly non-terminating computations while querying the metadata.
- The entire flake directory is copied to Nix store before evaluation.
- This allows for effective evaluation caching, which is relevant for large expressions such as Nixpkgs, but also requires copying the entire flake directory again on each change.
- No external variables, parameters, or impure language values are allowed.
- It means full reproducibility of a Nix expression, and, by extension, the resulting build instructions by default, but also prohibits parameterization of results by consumers.

Demo: Nix Flakes



- Let's take a look at a nix flake

NixOS



What is NixOS?

- A Linux distribution that can be configured and is based on Nix and Nixpkgs.
- Its underlying modular configuration system is written in the Nix language, and uses packages from Nixpkgs.
- The operating system environment and services it provides are configured with the Nix languages

About this System — Info Center

Search...

Basic Information

About this System

System Monitor

Energy


Detailed Information

Devices

Graphics

Network

About this System



NixOS 23.05
<https://nixos.org/>

Software

KDE Plasma Version: 5.27.5
KDE Frameworks Version: 5.106.0
Qt Version: 5.15.9
Kernel Version: 6.1.31 (64-bit)
Graphics Platform: X11

Hardware

Processors: 12 × AMD Ryzen 5 4600H with Radeon Graphics
Memory: 3.8 GiB of RAM
Graphics Processor: llvmpipe

Copy to Clipboard

9105LINUX.COM

Thank You



- Email: dhinojosa@evolutionnext.com
- Github: <https://www.github.com/dhinojosa>
- Mastodon: <https://mastodon.social/@dhinojosa>
- Linked In: <http://www.linkedin.com/in/dhevolutionnext>