Working with local files

Contents

To build a local project in a Nix derivation, source files must be accessible to its `builder` executable. Since by default, the `builder` runs in an isolated environment that only allows reading from the Nix store, the Nix language has built-in features to copy local files to the store and expose the resulting store paths.

Using these features directly can be tricky however:

- Coercion of paths to strings, such as the wide-spread pattern of `src = ./.`, makes the derivation dependent on the name of the current directory. Furthermore, it always adds the entire directory to the store, including unneeded files, which causes unnecessary new builds when they change.
- The `builtins.path` function (and equivalently `lib.sources.cleanSourceWith`) can address these problems. However, it's often hard to express the desired path selection using the `filter` function interface.

Skip to main content

In this tutorial you'll learn how to use the Nixpkgs `lib.fileset` library to work with local files in derivations. It abstracts over built-in functionality and offers a safer and more convenient interface.

File sets

A *file set* is a data type representing a collection of local files. File sets can be created, composed, and manipulated with the various functions of the library.

You can explore and learn about the library with `nix repl`:

```
$ nix repl -f channel:nixos-23.11
...
nix-repl> fs = lib.fileset
```

The `trace` function pretty-prints the files included in a given file set:

```
nix-repl> fs.trace ./. null
trace: /home/user (all files in directory)
null
```

All functions that expect a file set for an argument can also accept a path. Such path arguments are then implicitly turned into sets that contain *all* files under the given path. In the previous trace this is indicated by `(all files in directory)`.

Tip

The `trace` function pretty-prints its first argument and returns its second argument. But since you often just need the pretty-printing in `nix repl`, you can omit the second argument:

```
nix-repl> fs.trace ./.
trace: /home/user (all files in directory)
«lambda @ /nix/store/1czr278x24s3bl6qdnifpvm5z03wfi2p-nixpkgs-src/lib/fileset
```

Even though file sets conceptually contain local files, these files are *never* added to the Nix store

Skip to main content

copying secrets into the world-readable store.

In this example, although we pretty-printed the home directory, no files were copied. This is in contrast to coercion of paths to strings such as in `"${./.}"`, which copies the whole directory to the Nix store on evaluation!

> ⚠ **Warning**
>
> When using the `flakes` and `nix-command` experimental features, a local directory within a Flake is always copied into the Nix store *completely* unless it is a Git repository!

This implicit coercion also works for files:

```
$ touch some-file
```

```
nix-repl> fs.trace ./some-file
trace: /home/user
trace: - some-file (regular)
```

In addition to the included file, this also prints its file type.

Example project

To further experiment with the library, make a sample project. Create a new directory, enter it, and set up `npins` to pin the Nixpkgs dependency:

```
$ mkdir fileset
$ cd fileset
$ nix-shell -p npins --run "npins init --bare; npins add github nixos nixpkgs --branch
```

Then create a `default.nix` file with the following contents:

default.nix

Skip to main content

```
  sources ? import ./npins,
}:
let
  pkgs = import sources.nixpkgs {
    config = { };
    overlays = [ ];
    inherit system;
  };
in
pkgs.callPackage ./build.nix { }
```

Add two source files to work with:

```
$ echo hello > hello.txt
$ echo world > world.txt
```

Adding files to the Nix store

Files in a given file set can be added to the Nix store with `toSource`. The argument to this function requires a `root` attribute to determine which source directory to copy to the store. Only the files in the `fileset` attribute are included in the result.

Define `build.nix` as follows:

build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
  sourceFiles = ./hello.txt;
in

fs.trace sourceFiles

stdenv.mkDerivation {
  name = "fileset";
  src = fs.toSource {
    root = ./.;
    fileset = sourceFiles;
  };
  postInstall = ''
    mkdir $out
    cp -v hello.txt $out
  '';
```

Skip to main content

The call to `fs.trace` prints the file set that will be used as a derivation input.

Try building it:

> **Note**
>
> It will take a while to fetch Nixpkgs the first time around.

```
$ nix-build
trace: /home/user/fileset
trace: - hello.txt (regular)
this derivation will be built:
  /nix/store/3ci6avmjaijx5g8jhb218i183xi7bi2n-fileset.drv
...
'hello.txt' -> '/nix/store/sa4g6h13v0zbpfw6pzva860kp5aks44n-fileset/hello.txt'
...
/nix/store/sa4g6h13v0zbpfw6pzva860kp5aks44n-fileset
```

But the real benefit of the file set library comes from its facilities for composing file sets in different ways.

Difference

To be able to copy both files `hello.txt` and `world.txt` to the output, add the whole project directory as a source again:

build.nix

```
 { stdenv, lib }:
 let
   fs = lib.fileset;
-  sourceFiles = ./hello.txt;
+  sourceFiles = ./.;
 in

 fs.trace sourceFiles

 stdenv.mkDerivation {
   name = "fileset";
   src = fs.toSource {
     root = ./.;
     fileset = sourceFiles;
```

Skip to main content

```
      mkdir $out
-     cp -v hello.txt $out
+     cp -v {hello,world}.txt $out
    '';
 }
```

This will work as expected:

```
$ nix-build
trace: /home/user/fileset (all files in directory)
this derivation will be built:
  /nix/store/fsihp8872vv9ngbkc7si5jcbigs81727-fileset.drv
...
'hello.txt' -> '/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset/hello.txt'
'world.txt' -> '/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset/world.txt'
...
/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset
```

However, if you run `nix-build` again, the output path will be different!

```
$ nix-build
trace: /home/user/fileset (all files in directory)
this derivation will be built:
  /nix/store/nlh7ismrf27xsnl3m20vfz6rvwlbbbca-fileset.drv
...
'hello.txt' -> '/nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset/hello.txt'
'world.txt' -> '/nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset/world.txt'
...
/nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset
```

The problem here is that `nix-build` by default creates a `result` symlink in the working directory, which points to the store path just produced:

```
$ ls -l result
result -> /nix/store/xknflcvjaa8dj6a6vkg629zmcrgz10rh-fileset
```

Since `src` refers to the whole directory, and its contents change when `nix-build` succeeds, Nix will have to start over every time.

> **Note**
>
> This will also happen without the file set library, e.g. when setting `src = ./.;` directly.

Skip to main content

The `difference` function subtracts one file set from another. The result is a new file set that contains all files from the first argument that aren't in the second argument.

Use it to filter out `./result` by changing the `sourceFiles` definition:

build.nix

```
 { stdenv, lib }:
 let
   fs = lib.fileset;
-  sourceFiles = ./.;
+  sourceFiles = fs.difference ./. ./result;
 in
```

Building this, the file set library will specify which files are taken from the directory:

```
$ nix-build
trace: /home/user/fileset
trace: - build.nix (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - world.txt (regular)
this derivation will be built:
  /nix/store/zr19bv51085zz005yk7pw4s9sglmafvn-fileset.drv
...
'hello.txt' -> '/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset/hello.txt'
'world.txt' -> '/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset/world.txt'
...
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

An attempt to repeat the build will re-use the existing store path:

```
$ nix-build
trace: /home/user/fileset
trace: - build.nix (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - world.txt (regular)
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

Skip to main content

Missing files

Removing the `./result` symlink creates a new problem, though:

```
$ rm result
$ nix-build
error: lib.fileset.difference: Second argument (negative set)
  (/home/user/fileset/result) is a path that does not exist.
  To create a file set from a path that may not exist, use `lib.fileset.maybeMissing`.
```

Follow the instructions in the error message, and use `maybeMissing` to create a file set from a path that may not exist (in which case the file set will be empty):

build.nix

```
 { stdenv, lib }:
 let
   fs = lib.fileset;
-  sourceFiles = fs.difference ./. ./result;
+  sourceFiles = fs.difference ./. (fs.maybeMissing ./result);
 in
```

This now works, using the whole directory since `./result` is not present:

```
$ nix-build
trace: /home/user/fileset (all files in directory)
this derivation will be built:
  /nix/store/zr19bv51085zz005yk7pw4s9sglmafvn-fileset.drv
...
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

Another build attempt will produce a different trace, but the same output path:

```
$ nix-build
trace: /home/user/fileset
trace: - build.nix (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - world.txt (regular)
/nix/store/vhyhk6ij39gjapqavz1j1x3zbiy3qc1a-fileset
```

Skip to main content

Union (explicitly exclude files)

There is still a problem: Changing *any* of the included files causes the derivation to be built again, even though it doesn't depend on those files.

Append an empty line to `build.nix`:

```
$ echo >> build.nix
```

Again, Nix will start from scratch:

```
$ nix-build
trace: /home/user/fileset
trace: - default.nix (regular)
trace: - npins (all files in directory)
trace: - build.nix (regular)
trace: - string.txt (regular)
this derivation will be built:
  /nix/store/zmgpqlpfz2jq0w9rdacsnpx8ni4n77cn-filesets.drv
...
/nix/store/6pffjljjy3c7kla60nljk3fad4q4kkzn-filesets
```

One way to fix this is to use `unions`.

Create a file set containing a union of the files to exclude (`fs.unions [ ... ]`), and subtract it (`difference`) from the complete directory (`./.`):

build.nix

```
  sourceFiles =
    fs.difference
      ./.
      (fs.unions [
        (fs.maybeMissing ./result)
        ./default.nix
        ./build.nix
        ./npins
      ]);
```

Changing any of the excluded files now doesn't necessarily cause a new build anymore:

Skip to main content

```
$ nix-build
trace: /home/user/fileset
trace: - hello.txt (regular)
trace: - world.txt (regular)
/nix/store/ckn40y7hgqphhbhyrq64h9r6rvdh973r-fileset
```

Filter

The `fileFilter` function allows filtering file sets such that each included file satisfies the given criteria.

Use it to select all files with a name ending in `.nix`:

build.nix

```
   sourceFiles =
     fs.difference
       ./.
       (fs.unions [
         (fs.maybeMissing ./result)
-        ./default.nix
-        ./build.nix
+        (fs.fileFilter (file: file.hasExt "nix") ./.)
         ./nix
       ]);
```

This does not change the result, even if we add a new `.nix` file.

```
$ nix-build
trace: /home/user/fileset
trace: - hello.txt (regular)
trace: - world.txt (regular)
/nix/store/ckn40y7hgqphhbhyrq64h9r6rvdh973r-fileset
```

Notably, the approach of using `difference ./.` explicitly selects the files to *exclude*, which means that new files added to the source directory are included by default. Depending on your project, this might be a better fit than the alternative in the next section.

Skip to main content

Union (explicitly include files)

To contrast the previous approach, `unions` can also be used to select only the files to *include*. This means that new files added to the current directory would be ignored by default.

Create some additional files:

```
$ mkdir src
$ touch build.sh src/select.{c,h}
```

Then create a file set from only the files to be included explicitly:

build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
  sourceFiles = fs.unions [
    ./hello.txt
    ./world.txt
    ./build.sh
    (fs.fileFilter
      (file: file.hasExt "c" || file.hasExt "h")
      ./src
    )
  ];
in

fs.trace sourceFiles

stdenv.mkDerivation {
  name = "fileset";
  src = fs.toSource {
    root = ./.;
    fileset = sourceFiles;
  };
  postInstall = ''
    cp -vr . $out
  '';
}
```

The `postInstall` script is simplified to rely on the sources to be pre-filtered appropriately:

```
$ nix-build
trace: /home/user/fileset
```

Skip to main content

```
trace: - src (all files in directory)
trace: - world.txt (regular)
this derivation will be built:
  /nix/store/sjzkn07d6a4qfp60p6dc64pzvmmdafff-fileset.drv
...
'.' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset'
'./build.sh' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/build.sh'
'./hello.txt' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/hello.txt'
'./world.txt' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/world.txt'
'./src' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/src'
'./src/select.c' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/src/select.c'
'./src/select.h' -> '/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset/src/select.h'
...
/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset
```

Only the specified files are used, even when a new one is added:

```
$ touch src/select.o README.md

$ nix-build
trace: - build.sh (regular)
trace: - hello.txt (regular)
trace: - src
trace:   - select.c (regular)
trace:   - select.h (regular)
trace: - world.txt (regular)
/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset
```

Matching files tracked by Git

If a directory is part of a Git repository, passing it to `gitTracked` gives you a file set that only includes files tracked by Git.

Create a local Git repository and add all files except `src/select.o` and `./result` to it:

```
$ git init
Initialized empty Git repository in /home/user/fileset/.git/
$ git add -A
$ git reset src/select.o result
```

Re-use this selection of files with `gitTracked`:

Skip to main content

```
  sourceFiles = fs.gitTracked ./.;
```

Build it again:

```
$ nix-build
warning: Git tree '/home/user/fileset' is dirty
trace: /home/vg/src/nix.dev/fileset
trace: - README.md (regular)
trace: - build.nix (regular)
trace: - build.sh (regular)
trace: - default.nix (regular)
trace: - hello.txt (regular)
trace: - npins (all files in directory)
trace: - src
trace:   - select.c (regular)
trace:   - select.h (regular)
trace: - world.txt (regular)
this derivation will be built:
  /nix/store/p9aw3fl5xcjbgg9yagykywvskzgrmk5y-fileset.drv
...
/nix/store/cw4bza1r27iimzrdbfl4yn5xr36d6k5l-fileset
```

This includes too much though, as not all of these files are needed to build the derivation as originally intended.

> **Note**
>
> When using the `flakes` and `nix-command` experimental features, this function isn't needed, because `nix build` by default only allows access to files tracked by Git. However, in order to provide the same developer experience for stable Nix, use of this function is nevertheless recommended.

Intersection

This is where `intersection` comes in. It allows creating a file set that consists only of files that are in *both* of two given file sets.

Select all files that are both tracked by Git *and* relevant for the build:

```
build.nix
```

Skip to main content

```
  sourceFiles =
    fs.intersection
      (fs.gitTracked ./.)
      (fs.unions [
        ./hello.txt
        ./world.txt
        ./build.sh
        ./src
      ]);
```

This will produce the same output as in the other approach and therefore re-use a previous build result:

```
$ nix-build
warning: Git tree '/home/user/fileset' is dirty
trace: - build.sh (regular)
trace: - hello.txt (regular)
trace: - src
trace:   - select.c (regular)
trace:   - select.h (regular)
trace: - world.txt (regular)
/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset
```

Conclusion

We have shown some examples on how to use all of the fundamental file set functions. For more complex use cases, they can be composed as needed.

For the complete list and more details, see the `lib.fileset` reference documentation.