Channels

Snowfall Lib makes use of a core package set to build systems, packages, and more. This package set is taken from the input on your flake named `nixpkgs`. However, it is common to provide additional configuration for NixPkgs before using it. In order to do this, you can use the `channels-config` option.

Snowfall

On this page › Overview

```
        snowfall-lib = {
            url = "github:snowfallorg/lib";
            inputs.nixpkgs.follows = "nixpkgs";
        };
    };

    outputs = inputs:
        inputs.snowfall-lib.mkFlake {
            inherit inputs;
            src = ./.;

            # The attribute set specified here will be passed directly to NixPkg
            # instantiating the package set.
            channels-config = {
                # Allow unfree packages.
                allowUnfree = true;

                # Allow certain insecure packages
                permittedInsecurePackages = [
                    "firefox-100.0.0"
                ];

                # Additional configuration for specific packages.
                config = {
                    # For example, enable smartcard support in Firefox.
                    firefox.smartcardSupport = true;
                };
            };
        };
}
```

Snowfall

On this page Overview

Generic

Sometimes you want to put something on your flake output that isn't fully managed by Snowfall Lib. See the following two sections for the best ways to handle generic flake outputs.

Outputs Builder

Snowfall Lib extends flake-utils-plus which means you can make use of `outputs-builder` to construct flake outputs for each supported system.

```
        snowfall-lib = {
            url = "github:snowfallorg/lib";
            inputs.nixpkgs.follows = "nixpkgs";
        };
    };

    outputs = inputs:
        inputs.snowfall-lib.mkFlake {
            inherit inputs;
            src = ./.;

            # The outputs builder receives an attribute set of your available Ni
            # These are every input that points to a NixPkgs instance (even fork
            # case, the only channel available in this flake is `channels.nixpkg
            outputs-builder = channels: {
                # Outputs in the outputs builder are transformed to support each
                # entry will be turned into multiple different outputs like `for
                formatter = channels.nixpkgs.alejandra;
            };
        };
}
```

Custom

If you can't use `outputs-builder` then it is also possible to merge your flake outputs with another attribute set to provide custom entries.

Merging in this way is destructive and will overwrite things generated by Snowfall Lib that share the same names as the attributes you add.

```
        snowfall-lib = {
            url = "github:snowfallorg/lib";
            inputs.nixpkgs.follows = "nixpkgs";
        };
    };

    outputs = inputs:
        # Generate outputs from Snowfall Lib.
        (inputs.snowfall-lib.mkFlake {
            inherit inputs;
            src = ./.;
        })
        # And merge some attributes with it.
        // {
            my-custom-output = "hello world";
        };
}
```

Previous
Shells

Next
Channels

Getting Started

New flake

If your project does not have a flake yet:

```
nix flake init -t github:hercules-ci/flake-par
```

Existing flake

Otherwise, add the input,

```
flake-parts.url = "github:hercules-ci/flake-pa
```

then slide `mkFlake` between your outputs function h

```
outputs = inputs@{ flake-parts, ... }:
  flake-parts.lib.mkFlake { inherit inputs; }
    flake = {
      # Put your original flake attributes her
    };
    systems = [
      # systems for which you want to build th
      "x86_64-linux"
      # ...
    ];
    perSystem = { config, ... }: {
    };
  };
```

Now you can start using the flake-parts options.

Homes

To create a new home, add a new directory to your `homes` directory.

> **Note**
>
> Remember to run `git add` when creating new files!

```
# Create a directory in the `homes` directory for a new home. This should follow
# Snowfall Lib's required home target format to ensure that the correct architec
# and output are used.
mkdir -p ./homes/x86_64-linux/user@my-home
```

Now create the Nix file for the home at `homes/x86_64-linux/user@my-home/default.nix`.

```
    # An instance of `pkgs` with your overlays and packages applied is also avai
    pkgs,
    # You also have access to your flake's inputs.
    inputs,

    # Additional metadata is provided by Snowfall Lib.
    home, # The home architecture for this host (eg. `x86_64-linux`).
    target, # The Snowfall Lib target for this home (eg. `x86_64-home`).
    format, # A normalized name for the home target (eg. `home`).
    virtual, # A boolean to determine whether this home is a virtual target usin
    host, # The host name for this home.

    # All other arguments come from the home home.
    config,
    ...
}:
{
    # Your configuration.
}
```

This home will be made available on your flake's `homeConfigurations` output with the same name as the directory that you created.

Homes can have additional `specialArgs` and `modules` configured within your call to `mkFlake`. See the following for an example which adds a Home Manager module to a specific host and sets a custom value in `specialArgs`.

Snowfall

On this page › Overview

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
        inputs.snowfall-lib.mkFlake {
            inherit inputs;
            src = ./.;

            # Add modules to all homes.
            homes.modules = with inputs; [
                # my-input.homeModules.my-module
            ];

            # Add modules to a specific home.
            homes.users."my-user@my-host".modules = with inputs; [
                # my-input.homeModules.my-module
            ];

            # Add modules to a specific home.
            homes.users."my-user@my-host".specialArgs = {
                my-custom-value = "my-value";
            };
        };
}
```

Snowfall

On this page Overview

Library

Snowfall Lib automatically passes your merged library to all other parts of your flake. This means that you can access your own library with `lib.my-namespace` or any library from your flake inputs with `lib.my-input`. The namespace for your library and packages can be configured with `snowfall.namespace`.

To create a library, add a new directory to your `lib` directory or use the base `lib` directory.

> **Note**
>
> Remember to run `git add` when creating new files!

```
# Create a directory in the `lib` directory for library methods that will be mer
mkdir -p ./lib/my-lib
```

Now create the Nix file for the lib at `lib/my-lib/default.nix` (or `lib/default.nix`).

```
    # Your flake inputs are also available.
    inputs,

    # Additionally, Snowfall Lib's own inputs are passed. You probably don't nee
    snowfall-inputs,
}:
{
    # This will be available as `lib.my-namespace.my-helper-function`.
    my-helper-function = x: x;

    my-scope = {
        # This will be available as `lib.my-namespace.my-scope.my-scoped-helper-
        my-scoped-helper-function = x: x;
    };
}
```

This library will be made available on your flake's `lib` output.

Previous
Homes

Next
Shells

Lib

Usage

`snowfall-lib` provides two utilities directly on the flake itself.

`mkLib`

The library generator function. This is the entrypoint for `snowfall-lib` and is how you access all of its features. See the following Nix Flake example for how to create a library instance with `mkLib`.

Snowfall

On this page › Overview

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;

        # You can optionally place your Snowfall-related files in another
        # directory.
        snowfall.root = ./nix;
      };
    in
    # We'll cover what to do here next.
    { };
}
```

For information on how to use `lib`, see the `lib` section. Or skip directly to `lib.mkFlake` to see how to configure your flake's outputs.

`mkFlake`

A convenience wrapper for writing the following.

```
    # You can optionally place your Snowfall-related files in another
    # directory.
    snowfall.root = ./nix;
  };
in lib.mkFlake {
}
```

Instead, with `mkFlake` you can combine these calls into one like the following.

```
inputs.snowfall-lib.mkFlake {
  inherit inputs;
  src = ./.;
};
```

See `lib.mkFlake` for information on how to configure your flake's outputs.

`lib`

Snowfall Lib provides utilities for creating flake outputs as well as some necessary helpers. In addition, `lib` is an extension of `nixpkgs.lib` and every flake input that contains a `lib` attribute. This means that you can use `lib` directly for all of your needs, whether they're Snowfall-related, NixPkgs-related, or for one of the other flake inputs.

The way that `mkLib` merges libraries is by starting with the base `nixpkgs.lib` and then me each flake input's `lib` attribute, namespaced by the name of the input. For example, if you have the input `flake-utils-plus` then you will be able to use `lib.flake-utils-plus` instead of having to keep a reference to the input's lib at `inputs.flake-utils-plus.lib`.

If you have your own library in a `lib/` directory at your flake's root, definitions in there will automatically be imported and merged as well.

`lib.mkFlake`

The `lib.mkFlake` function creates full flake outputs. For most cases you will only need to use this helper and the Snowfall `lib` will take care of everything else.

Flake Structure

Snowfall Lib has opinions about how a flake's files are laid out. This lets `lib` do all of the busy work for you and allows you to focus on creating. Here is the structure that `lib` expects to find at the root of your flake.

```
|
| Your Nix flake.
├─ flake.nix
|
| An optional custom library.
├─ lib/
|  |
|  | A Nix function called with `inputs`, `snowfall-inputs`, and `lib`.
|  | The function should return an attribute set to merge with `lib`.
|  ├─ default.nix
|  |
|  | Any (nestable) directory name.
|  └─ **/
|     |
|     | A Nix function called with `inputs`, `snowfall-inputs`, and `lib`.
|     | The function should return an attribute set to merge with `lib`.
|     └─ default.nix
|
| An optional set of packages to export.
├─ packages/
|  |
|  | Any (nestable) directory name. The name of the directory will be the
|  | name of the package.
|  └─ **/
|     |
|     | A Nix package to be instantiated with `callPackage`. This file
|     | should contain a function that takes an attribute set of packages
|     | and *required* `lib` and returns a derivation.
|     └─ default.nix
|
|
├─ modules/ (optional modules)
|  |
|  | A directory named after the `platform` type that will be used for modules w
```

Snowfall

On this page › Overview

```
│  │ - home
│  └ <platform>/
│     │
│     │ Any (nestable) directory name. The name of the directory will be the
│     │ name of the module.
│     └ **/
│        │
│        │ A NixOS module.
│        └ default.nix
│
├ overlays/ (optional overlays)
│  │
│  │ Any (nestable) directory name.
│  └ **/
│     │
│     │ A custom overlay. This file should contain a function that takes three a
│     │   - An attribute set of your flake's inputs and a `channels` attribute c
│     │     all of your available channels (eg. nixpkgs, unstable).
│     │   - The final set of `pkgs`.
│     │   - The previous set of `pkgs`.
│     │
│     │ This function should return an attribute set to merge onto `pkgs`.
│     └ default.nix
│
├ systems/ (optional system configurations)
│  │
│  │ A directory named after the `system` type that will be used for all mach
│  │
│  │ The architecture is any supported architecture of NixPkgs, for example:
│  │  - x86_64
│  │  - aarch64
│  │  - i686
│  │
│  │ The format is any supported NixPkgs format *or* a format provided by either
```

Snowfall

On this page › Overview

```
│  │  - darwin
│  │  - iso
│  │  - install-iso
│  │  - do
│  │  - vmware
│  │
│  │ With the architecture and format together (joined by a hyphen), you get the
│  │ directory for the system type.
│  └─ <architecture>-<format>/
│     │
│     │ A directory that contains a single system's configuration. The directory
│     │ will be the name of the system.
│     └─ <system-name>/
│        │
│        │ A NixOS module for your system's configuration.
│        └─ default.nix
│
├─ homes/ (optional homes configurations)
│  │
│  │ A directory named after the `home` type that will be used for all homes wit
│  │
│  │ The architecture is any supported architecture of NixPkgs, for example:
│  │  - x86_64
│  │  - aarch64
│  │  - i686
│  │
│  │ The format is any supported NixPkgs format *or* a format provided by eit
│  │ or nixos-generators. However, in order to build systems with nix-darwin or
│  │ you must add `darwin` and `nixos-generators` inputs to your flake respectiv
│  │ are some example formats:
│  │  - linux
│  │  - darwin
│  │  - iso
│  │  - install-iso
```

```
|  | directory for the home type.
|  └─ <architecture>-<format>/
|     |
|     | A directory that contains a single home's configuration. The directory n
|     | will be the name of the home.
|     └─ <home-name>/
|        |
|        | A NixOS module for your home's configuration.
|        └─ default.nix
```

Default Flake

Without any extra input, `lib.mkFlake` will generate outputs for all systems, modules, packages, overlays, and shells specified by the Flake Structure section.

Snowfall

On this page › Overview

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    # This is an example and in your actual flake you can use `snowfall-lib.mkFl
    # directly unless you explicitly need a feature of `lib`.
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;
      };
    in
      lib.mkFlake { };
}
```

Snowfall Configuration

Snowfall Lib supports configuring some functionality and interopability with other tools via the `snowfall` attribute passed to `mkLib`.

Snowfall

On this page › Overview

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    # This is an example and in your actual flake you can use `snowfall-lib.mkFl
    # directly unless you explicitly need a feature of `lib`.
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;

        snowfall = {
        namespace = "my-namespace";
          meta = {
            # Your flake's preferred name in the flake registry.
            name = "my-flake";
            # A pretty name for your flake.
            title = "My Flake";
          };
        };
      };
    in
      lib.mkFlake { };
}
```

Snowfall

On this page Overview

Snowfall

On this page › Overview

```
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    home-manager = {
      url = "github:nix-community/home-manager";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    # This is an example and in your actual flake you can use `snowfall-lib.mkFl
    # directly unless you explicitly need a feature of `lib`.
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;
      };
    in
      lib.mkFlake {
        # Add overlays for the `nixpkgs` channel.
        overlays = with inputs; [
          # my-inputs.overlays.my-overlay
        ];

        # Add modules to all NixOS systems.
        systems.modules.nixos = with inputs; [
          # my-input.nixosModules.my-module
        ];
```

```
        ];

        # Add modules to a specific system.
        systems.hosts.my-host = with inputs; [
          # my-input.nixosModules.my-module
        ];

        # Add modules to all homes.
        homes.modules = with inputs; [
          # my-input.homeModules.my-module
        ];

        # Add modules to a specific home.
        homes.users."my-user@my-host".modules = with inputs; [
          # my-input.homeModules.my-module
        ];
      };
}
```

Internal Packages And Outputs

Packages created from your `packages/` directory are automatically made available via an overlay for your `nixpkgs` channel. System configurations can access these packages directly on `pkgs` and consumers of your flake can use the generated `<your-flake>.overlays` attributes.

Snowfall

On this page › Overview

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    comma = {
      url = "github:nix-community/comma";
      inputs.nixpkgs.follows = "unstable";
    };
  };

  outputs = inputs:
    # This is an example and in your actual flake you can use `snowfall-lib.mkFl
    # directly unless you explicitly need a feature of `lib`.
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;
      };
    in
      lib.mkFlake {
        # Optionally place all packages under a namespace when used in an overla
        # Instead of accessing packages with `pkgs.<name>`, your internal pac
        # will be available at `pkgs.<namespace>.<name>`.
        package-namespace = "my-namespace";

        # You can also pass through external packages or dynamically create new
        # in addition to the ones that `lib` will create from your `packages/` d
        outputs-builder = channels: {
          packages = {
```

```
}
```

Default Packages And Shells

Snowfall Lib will create packages and shells based on your `packages/` and `shells` directories. However, it is common to additionally map one of those packages or shells to be their respective default. This can be achieved by setting an `alias` and mapping the `default` package or shell to the name of the one you want.

Snowfall

On this page › Overview

```
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    # This is an example and in your actual flake you can use `snowfall-lib.mkFl
    # directly unless you explicitly need a feature of `lib`.
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;
      };
    in
      lib.mkFlake {
        alias = {
          packages = {
            default = "my-package";
          };

          shells = {
            default = "my-shell";
          };

          modules = {
            default = "my-module";
          };

          templates = {
```

```
}
```

Darwin And NixOS Generators

Snowfall Lib has support for configuring macOS systems and building any output supported by NixOS Generators. In order to use these features, your flake must include `darwin` and/or `nixos-generators` as inputs.

Snowfall

On this page › Overview

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    # In order to configure macOS systems.
    darwin = {
      url = "github:lnl7/nix-darwin";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    # In order to build system images and artifacts supported by nixos-generator
    nixos-generators = {
      url = "github:nix-community/nixos-generators";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    # This is an example and in your actual flake you can use `snowfall-lib.mkFl
    # directly unless you explicitly need a feature of `lib`.
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;
      };
    in
      # No additional configuration is required to use this feature, you only
      # have to add darwin or nixos-generators to your flake inputs.
      lib.mkFlake { };
```

flake at `<format>Configurations` where `<format>` is the name of the generator type. See the following table for a list of supported formats from NixOS Generators.

format	description
amazon	Amazon EC2 image
azure	Microsoft azure image (Generation 1 / VHD)
cloudstack	qcow2 image for cloudstack
do	Digital Ocean image
gce	Google Compute image
hyperv	Hyper-V Image (Generation 2 / VHDX)
install-iso	Installer ISO
install-iso-hyperv	Installer ISO with enabled hyper-v support
iso	ISO
kexec	kexec tarball (extract to / and run /kexec_nixos)
kexec-bundle	Same as before, but it's just an executable
kubevirt	KubeVirt image
lxc	Create a tarball which is importable as an lxc container, use together with lxc-metadata
lxc-metadata	The necessary metadata for the lxc image to start
openstack	qcow2 image for openstack
proxmox	VMA file for proxmox
qcow	qcow2 image

raw-efi	Raw image with efi support
sd-aarch64	Like sd-aarch64-installer, but does not use default installer image config.
sd-aarch64-installer	create an installer sd card for aarch64
vagrant-virtualbox	VirtualBox image for Vagrant
virtualbox	virtualbox VM
vm	Only used as a qemu-kvm runner
vm-bootloader	Same as vm, but uses a real bootloader instead of netbooting
vm-nogui	Same as vm, but without a GUI
vmware	VMWare image (VMDK)

Home Manager

Snowfall Lib supports configuring Home Manager for both standalone use and for use as a module with NixOS or nix-darwin. To use this feature, your flake must include `home-manager` as an input.

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    # In order to use Home Manager.
    home-manager = {
      url = "github:nix-community/home-manager";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
    # This is an example and in your actual flake you can use `snowfall-lib.mkFl
    # directly unless you explicitly need a feature of `lib`.
    let
      lib = inputs.snowfall-lib.mkLib {
        # You must pass in both your flake's inputs and the root directory of
        # your flake.
        inherit inputs;
        src = ./.;
      };
    in
      # No additional configuration is required to use this feature, you only
      # have to add home-manager to your flake inputs.
      lib.mkFlake { };
}
```

lib.snowfall.flake

Snowfall

On this page › Overview

Remove the `self` attribute from an attribute set.

Type: `Attrs -> Attrs`

Usage:

```
without-self { self = {}; x = true; }
```

Result:

```
{ x = true; }
```

`lib.snowfall.flake.without-src`

Remove the `src` attribute from an attribute set.

Type: `Attrs -> Attrs`

Usage:

```
without-src { src = {}; x = true; }
```

Result:

```
{ x = true; }
```

`lib.snowfall.flake.without-snowfall-inputs`

Remove the `src` and `self` attributes from an attribute set.

Type: `Attrs -> Attrs`

Usage:

```
{ x = true; }
```

lib.snowfall.flake.get-libs

Transform an attribute set of inputs into an attribute set where the values are the inputs' `lib` attribute. Entries without a `lib` attribute are removed.

Type: `Attrs -> Attrs`

Usage:

```
get-lib { x = nixpkgs; y = {}; }
```

Result:

```
{ x = nixpkgs.lib; }
```

lib.snowfall.path

lib.snowfall.path.split-file-extension

Split a file name and its extension.

Type: `String -> [String]`

Usage:

```
split-file-extension "my-file.md"
```

Result:

Snowfall

On this page › Overview

Check if a file name has a file extension.

Type: `String -> Bool`

Usage:

```
has-any-file-extension "my-file.txt"
```

Result:

```
true
```

`lib.snowfall.path.get-file-extension`

Get the file extension of a file name.

Type: `String -> String` Usage:

```
get-file-extension "my-file.final.txt"
```

Result:

```
"txt"
```

`lib.snowfall.path.has-file-extension`

Check if a file name has a specific file extension.

Type: `String -> String -> Bool`

Usage:

```
true
```

lib.snowfall.path.get-parent-directory

Get the parent directory for a given path.

Type: `Path -> Path`

Usage:

```
get-parent-directory "/a/b/c"
```

Result:

```
"/a/b"
```

lib.snowfall.path.get-file-name-without-extension

Get the file name of a path without its extension.

Type: `Path -> String`

Usage:

```
get-file-name-without-extension ./some-directory/my-file.pdf
```

Result:

```
"my-file"
```

`lib.snowfall.fs.is-file-kind`

`lib.snowfall.fs.is-symlink-kind`

`lib.snowfall.fs.is-directory-kind`

`lib.snowfall.fs.is-unknown-kind`

Matchers for file kinds. These are often used with `readDir`.

Type: `String -> Bool`

Usage:

```
is-file-kind "directory"
```

Result:

```
false
```

`lib.snowfall.fs.get-file`

Get a file path relative to the user's flake.

Type: `Path -> Path`

Usage:

```
get-file "systems"
```

Result:

```
"/user-source/systems"
```

Type: ~~Path -> Path~~

Usage:

```
get-snowfall-file "systems"
```

Result:

```
"/user-source/snowfall-dir/systems"
```

`lib.snowfall.fs.internal-get-file`

Get a file relative to the Snowfall Lib flake. You probably shouldn't use this!

Type: `Path -> Path`

Usage:

```
get-file "systems"
```

Result:

```
"/snowfall-lib-source/systems"
```

`lib.snowfall.fs.safe-read-directory`

Safely read from a directory if it exists.

Type: `Path -> Attrs`

Usage:

```
{ "my-file.txt" = "regular"; }
```

lib.snowfall.fs.get-directories

Get directories at a given path.

Type: `Path -> [Path]`

Usage:

```
get-directories ./something
```

Result:

```
[ "./something/a-directory" ]
```

lib.snowfall.fs.get-files

Get files at a given path.

Type: `Path -> [Path]`

Usage:

```
get-files ./something
```

Result:

```
[ "./something/a-file" ]
```

lib.snowfall.fs.get-files-recursive

Usage:

```
get-files-recursive ./something
```

Result:

```
[ "./something/some-directory/a-file" ]
```

`lib.snowfall.fs.get-nix-files`

Get nix files at a given path.

Type: `Path -> [Path]`

Usage:

```
get-nix-files "./something"
```

Result:

```
[ "./something/a.nix" ]
```

`lib.snowfall.fs.get-nix-files-recursive`

Get nix files at a given path, traversing any directories within.

Type: `Path -> [Path]`

Usage:

```
get-nix-files "./something"
```

`lib.snowfall.fs.get-default-nix-files`

Get nix files at a given path named “default.nix”.

Type: `Path -> [Path]`

Usage:

```
get-default-nix-files "./something"
```

Result:

```
[ "./something/default.nix" ]
```

`lib.snowfall.fs.get-default-nix-files-recursive`

Get nix files at a given path named “default.nix”, traversing any directories within.

Type: `Path -> [Path]`

Usage:

```
get-default-nix-files-recursive "./something"
```

Result:

```
[ "./something/some-directory/default.nix" ]
```

`lib.snowfall.fs.get-non-default-nix-files`

Get nix files at a given path not named “default.nix”.

```
get-non-default-nix-files "./something"
```

Result:

```
[ "./something/a.nix" ]
```

`lib.snowfall.fs.get-non-default-nix-files-recursive`

Get nix files at a given path not named “default.nix”, traversing any directories within.

Type: `Path -> [Path]`

Usage:

```
get-non-default-nix-files-recursive "./something"
```

Result:

```
[ "./something/some-directory/a.nix" ]
```

`lib.snowfall.module`

Utilities for working with NixOS modules.

`lib.snowfall.module.create-modules`

Create flake output modules.

Type: `Attrs -> Attrs`

Usage:

```
{ another-module = ...; my-module = ...; default = ...; }
```

lib.snowfall.attrs

Utilities for working with attribute sets.

lib.snowfall.attrs.map-concat-attrs-to-list

Map and flatten an attribute set into a list.

Type: `(a -> b -> [c]) -> Attrs -> [c]`

Usage:

```
map-concat-attrs-to-list (name: value: [name value]) { x = 1; y = 2; }
```

Result:

```
[ "x" 1 "y" 2 ]
```

lib.snowfall.attrs.merge-deep

Recursively merge a list of attribute sets.

Type: `[Attrs] -> Attrs`

Usage:

```
merge-deep [{ x = 1; } { x = 2; }]
```

Result:

Snowfall

On this page › Overview

Merge the root of a list of attribute sets.

Type: `[Attrs] -> Attrs`

Usage:

```
merge-shallow [{ x = 1; } { x = 2; }]
```

Result:

```
{ x = 2; }
```

`lib.snowfall.attrs.merge-shallow-packages`

Merge shallow for packages, but allow one deeper layer of attributes sets.

Type: `[Attrs] -> Attrs`

Usage:

```
merge-shallow-packages [
  {
    inherit (pkgs) vim;
    namespace.first = 1;
  }
  {
    inherit (unstable) vim;
    namespace.second = 2;
  }
]
```

Result:

```
    second = 2;
  };
}
```

lib.snowfall.system

lib.snowfall.system.is-darwin

Check whether a named system is macOS.

Type: `String -> Bool`

Usage:

```
is-darwin "x86_64-linux"
```

Result:

```
false
```

lib.snowfall.system.is-linux

Check whether a named system is Linux.

Type: `String -> Bool`

Usage:

```
is-linux "x86_64-linux"
```

Result:

Check whether a named system is virtual.

Type: `String -> Bool`

Usage:

```
is-linux "x86_64-iso"
```

Result:

```
true
```

`lib.snowfall.system.get-virtual-system-type`

Get the virtual system type of a system target.

Type: `String -> String`

Usage:

```
get-virtual-system-type "x86_64-iso"
```

Result:

```
"iso"
```

`lib.snowfall.system.get-inferred-system-name`

Get the name of a system based on its file path.

Type: `Path -> String`

Result:

```
"my-system"
```

`lib.snowfall.system.get-target-systems-metadata`

Get structured data about all systems for a given target.

Type: `String -> [Attrs]`

Usage:

```
get-target-systems-metadata "x86_64-linux"
```

Result:

```
[ { target = "x86_64-linux"; name = "my-machine"; path = "/systems/x86_64-linux/
```

`lib.snowfall.system.get-system-builder`

Get the system builder for a given target.

Type: `String -> Function`

Usage:

```
get-system-builder "x86_64-iso"
```

Result:

```
(args: <system>)
```

Type: ~~`String -> String`~~

Usage:

```
get-system-output "aarch64-darwin"
```

Result:

```
"darwinConfigurations"
```

`lib.snowfall.system.get-resolved-system-target`

Get the resolved (non-virtual) system target.

Type: `String -> String`

Usage:

```
get-resolved-system-target "x86_64-iso"
```

Result:

```
"x86_64-linux"
```

`lib.snowfall.system.create-system`

Create a system.

Type: `Attrs -> Attrs`

Usage:

```
<flake-utils-plus-system-configuration>
```

lib.snowfall.system.create-systems

Create all available systems.

Type: `Attrs -> Attrs`

Usage:

```
create-systems { hosts.my-host.specialArgs.x = true; modules.nixos = [ my-shared
```

Result:

```
{ my-host = <flake-utils-plus-system-configuration>; }
```

lib.snowfall.home

lib.snowfall.home.split-user-and-host

Get the user and host from a combined string.

Type: `String -> Attrs`

Usage:

```
split-user-and-host "myuser@myhost"
```

Result:

```
{ user = "myuser"; host = "myhost"; }
```

Type: ~~Attrs -> Attrs~~

Usage:

```
create-home { path = ./homes/my-home; }
```

Result:

```
<flake-utils-plus-home-configuration>
```

`lib.snowfall.home.create-homes`

Create all available homes.

Type: `Attrs -> Attrs`

Usage:

```
create-homes { users."my-user@my-system".specialArgs.x = true; modules = [ my-sh
```

Result:

```
{ "my-user@my-system" = <flake-utils-plus-home-configuration>; }
```

`lib.snowfall.home.get-target-homes-metadata`

Get structured data about all homes for a given target.

Type: `String -> [Attrs]`

Usage:

```
[ { system = "x86_64-linux"; name = "my-home"; path = "/homes/x86_64-linux/my-ho
```

lib.snowfall.home.create-home-system-modules

Create system modules for home-manager integration.

Type: `Attrs -> [Module]`

Usage:

```
create-home-system-modules { users."my-user@my-system".specialArgs.x = true; mod
```

Result:

```
[Module]
```

lib.snowfall.package

Utilities for working with flake packages.

lib.snowfall.package.create-packages

Create flake output packages.

Type: `Attrs -> Attrs`

Usage:

```
create-packages { inherit channels; src = ./my-packages; overrides = { inherit a
```

Result:

Utilities for working with flake dev shells.

lib.snowfall.shell.create-shell

Create flake output packages.

Type: `Attrs -> Attrs`

Usage:

```
create-shells { inherit channels; src = ./my-shells; overrides = { inherit anoth
```

Result:

```
{ another-shell = ...; my-shell = ...; default = ...; }
```

lib.snowfall.overlay

Utilities for working with channel overlays.

lib.snowfall.overlay.create-overlays-builder

Create a flake-utils-plus overlays builder.

Type: `Attrs -> Attrs -> [(a -> b -> c)]`

Usage:

```
create-overlays-builder { src = ./my-overlays; package-namespace = "my-packages"
```

Result:

Create overlays to be used for flake outputs.

Type: `Attrs -> Attrs`

Usage:

```
create-overlays {
  src = ./my-overlays;
  packages-src = ./my-packages;
  package-namespace = "my-namespace";
  extra-overlays = {
    my-example = final: prev: {};
  };
}
```

Result:

```
{
  default = final: prev: {};
  my-example = final: prev: {};
  some-overlay = final: prev: {};
}
```

`lib.snowfall.template`

Utilities for working with flake templates.

`lib.snowfall.template.create-templates`

Create flake templates.

Type: `Attrs -> Attrs`

Result:

```
{ another-template = ...; my-template = ...; default = ...; }
```

Previous
Channels

Next
v2

Modules

Snowfall Lib automatically applies all of your modules to your systems. This means that all NixOS modules will be imported for your NixOS systems, all Darwin modules will be imported for your Darwin systems, and all Home Manager modules will be imported for your Home configurations.

To create a new module, add a new directory to your `modules` directory.

> **Note**
>
> Remember to run `git add` when creating new files!

```
# Create a directory in the `modules/nixos`, `modules/darwin`, or `modules/home`
# directory for a new module.
mkdir -p ./modules/nixos/my-module
```

Now create the Nix file for the module at `modules/nixos/my-module/default.nix`.

Snowfall

On this page › Overview

```
    # An instance of `pkgs` with your overlays and packages applied is also avai
    pkgs,
    # You also have access to your flake's inputs.
    inputs,

    # Additional metadata is provided by Snowfall Lib.
    system, # The system architecture for this host (eg. `x86_64-linux`).
    target, # The Snowfall Lib target for this system (eg. `x86_64-iso`).
    format, # A normalized name for the system target (eg. `iso`).
    virtual, # A boolean to determine whether this system is a virtual target us
    systems, # An attribute map of your defined hosts.

    # All other arguments come from the module system.
    config,
    ...
}:
{
    # Your configuration.
}
```

This module will be made available on your flake's `nixosModules`, `darwinModules`, or `homeModules` output with the same name as the directory that you created.

Previous
Overlays

Next
Systems

Overlays

Snowfall Lib automatically exports your overlays and applies them to your NixPkgs instance used within your flake. This includes making the overlaid packages available to packages in your flake, NixOS systems, Darwin systems, Home Manager, modules, and overlays.

To create a new overlay, add a new directory to your `overlays` directory.

> **Note**
>
> Remember to run `git add` when creating new files!

```
# Create a directory in the `overlays` directory for a new overlay.
mkdir -p ./overlays/my-overlay
```

Now create the Nix file for the overlay at `overlays/my-overlay/default.nix`.

```
# These channels are system-specific instances of NixPkgs that can be used to qu
# pull packages into your overlay.
#
# All other arguments for this function are your flake inputs.
{ channels, my-input, ... }:

final: prev: {
    # For example, to pull a package from unstable NixPkgs make sure you have th
    # input `unstable = "github:nixos/nixpkgs/nixos-unstable"` in your flake.
    inherit (channels.unstable) chromium;

    my-package = my-input.packages.${prev.system}.my-package;
}
```

This overlay will be made available on your flake's `overlays` output with the same name as the directory that you created.

Previous
Packages

Next
Modules

Packages

Snowfall Lib automatically exports your packages on your flake and makes them available to all other parts of your flake. This includes making these packages available to other packages in your flake, NixOS systems, Darwin systems, Home Manager, modules, and overlays.

To create a new package, add a new directory your `packages` directory.

> **Note**
>
> Remember to run `git add` when creating new files!

```
# Create a directory in the `packages` directory for a new package.
mkdir -p ./packages/my-package
```

Now create the Nix file for the package at `packages/my-package/default.nix`.

```
    # You also have access to your flake's inputs.
    inputs,

    # All other arguments come from NixPkgs. You can use `pkgs` to pull packages
    # programmatically or you may add the named attributes as arguments here.
    pkgs,
    stdenv,
    ...
}:

stdenv.mkDerivation {
    # Create your package
}
```

This package will be made available on your flake's `packages` output with the same name as the directory that you created.

Previous
Quickstart

Next
Overlays

Quickstart

Snowfall Lib is a library that makes it easy to manage your Nix flake by imposing an opinionated file structure.

Create a flake if you don't have one already

Snowfall Lib generates your Nix flake outputs for you. If you don't already have a Nix flake, you can create one using the following command.

```
# Create a flake in the current directory.
nix flake init
```

Add Snowfall Lib to your flake

To start using Snowfall Lib, import the library in your Nix flake by adding it to your flake's inputs.

```
        # The name "snowfall-lib" is required due to how Snowfall Lib processes
        # flake's inputs.
        snowfall-lib = {
            url = "github:snowfallorg/lib";
            inputs.nixpkgs.follows = "nixpkgs";
        };
    };

    # We will handle this in the next section.
    outputs = inputs: {};
}
```

Create your flake outputs

```
        snowfall-lib = {
            url = "github:snowfallorg/lib";
            inputs.nixpkgs.follows = "nixpkgs";
        };
    };

    outputs = inputs:
        inputs.snowfall-lib.mkFlake {
            # You must provide our flake inputs to Snowfall Lib.
            inherit inputs;

            # The `src` must be the root of the flake. See configuration
            # in the next section for information on how you can move your
            # Nix files to a separate directory.
            src = ./.;
        };
}
```

Configure Snowfall Lib

Snowfall Lib offers some customization options. The following example details a few popular settings. For a full list see Snowfall Lib Reference.

Snowfall

On this page › Overview

```
        snowfall-lib = {
            url = "github:snowfallorg/lib";
            inputs.nixpkgs.follows = "nixpkgs";
        };
    };

    outputs = inputs:
        inputs.snowfall-lib.mkFlake {
            inherit inputs;
            src = ./.;

            # Configure Snowfall Lib, all of these settings are optional.
            snowfall = {
                # Tell Snowfall Lib to look in the `./nix/` directory for your
                # Nix files.
                root = ./nix;

                # Choose a namespace to use for your flake's packages, library,
                # and overlays.
                namespace = "my-namespace";

                # Add flake metadata that can be processed by tools like Snowfal
                meta = {
                    # A slug to use in documentation when displaying things like
                    name = "my-awesome-flake";

                    # A title to show for your flake, typically the name.
                    title = "My Awesome Flake";
                };
            };
        };
}
```

technical information.

Next
Packages

Shells

To create a new shell, add a new directory your `shells` directory.

> **Note**
>
> Remember to run `git add` when creating new files!

```
# Create a directory in the `shells` directory for a new shell.
mkdir -p ./shells/my-shell
```

Now create the Nix file for the shell at `shells/my-shell/default.nix`.

```
    # You also have access to your flake's inputs.
    inputs,

    # All other arguments come from NixPkgs. You can use `pkgs` to pull shells o
    # programmatically or you may add the named attributes as arguments here.
    pkgs,
    stdenv,
    ...
}:

stdenv.mkDerivation {
    # Create your shell
}
```

This shell will be made available on your flake's `devShells` output with the same name as the directory that you created.

Previous
Library

Next
Generic

Snowfall Lib v2 Migration

Snowfall Lib v2 adds a large amount of features and has made a few breaking changes that are in the benefit of overall user experience. To migrate from v1 to v2, see the following steps.

Namespace

The `overlay-package-namespace` option has been removed in favor of using `snowfall.namespace`.

```
inputs.snowfall-lib.mkFlake {
    # Before:
    # overlay-package-namespace = "my-namespace";

    # After:
    snowfall.namespace = "my-namespace";
}
```

In addition, packages and your flake library now default to the `internal` namespace.

Aliases

Output aliases are no longer automatically remapped from strings returned from `outputs-builder`. Instead, use the new `alias` option to configure aliases.

```
    # };

    # After:
    alias.packages.default = "my-package";
}
```

External Modules

Modules must now be added to a specific system or system type. Previously Snowfall Lib assumed modules were all compatible NixOS modules. This has been extended to also support Darwin modules.

> **Note**
>
> Modules may still be added to specific systems via `systems.hosts.<my-host>.modules`.

my-input.nixosModules.my-module

```
inputs.snowfall-lib.mkFlake {
    # Before:
    # systems.modules = with inputs; [
    # };

    # After:
    systems.modules.nixos = with inputs; [
        my-input.nixosModules.my-module
    ];
    systems.modules.darwin = with inputs; [
        my-input.darwinModules.my-module
    ];
}
```

that you are now expecting to use the most recent, new structure imposed by Snowfall Lib.

Previous
Reference

Systems

To create a new system, add a new directory to your `systems` directory.

> **Note**
>
> Remember to run `git add` when creating new files!

```
# Create a directory in the `systems` directory for a new system. This should fo
# Snowfall Lib's required system target format to ensure that the correct archit
# and output are used.
mkdir -p ./systems/x86_64-linux/my-system
```

Now create the Nix file for the system at `systems/x86_64-linux/my-system/default.nix`.

Snowfall

On this page › Overview

```
    # An instance of `pkgs` with your overlays and packages applied is also avai
    pkgs,
    # You also have access to your flake's inputs.
    inputs,

    # Additional metadata is provided by Snowfall Lib.
    system, # The system architecture for this host (eg. `x86_64-linux`).
    target, # The Snowfall Lib target for this system (eg. `x86_64-iso`).
    format, # A normalized name for the system target (eg. `iso`).
    virtual, # A boolean to determine whether this system is a virtual target us
    systems, # An attribute map of your defined hosts.

    # All other arguments come from the system system.
    config,
    ...
}:
{
    # Your configuration.
}
```

This system will be made available on your flake's `nixosConfigurations`, `darwinConfigurations`, or one of Snowfall Lib's virtual `*Configurations` outputs with the same name as the directory that you created.

Systems can have additional `specialArgs` and `modules` configured within your call to `mkFlake`. See the following for an example which adds a NixOS module to a specific host and sets a custom value in `specialArgs`.

Snowfall

On this page › Overview

```
    nixpkgs.url = "github:nixos/nixpkgs/nixos-23.05";

    snowfall-lib = {
      url = "github:snowfallorg/lib";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = inputs:
        inputs.snowfall-lib.mkFlake {
            inherit inputs;
            src = ./.;

            # Add modules to all NixOS systems.
            systems.modules.nixos = with inputs; [
                # my-input.nixosModules.my-module
            ];

            # If you wanted to configure a Darwin (macOS) system.
            # systems.modules.darwin = with inputs; [
            #   my-input.darwinModules.my-module
            # ];

            # Add a module to a specific host.
            systems.hosts.my-host.modules = with inputs; [
                # my-input.nixosModules.my-module
            ];

            # Add a custom value to `specialArgs`.
            system.hosts.my-host.specialArgs = {
                my-custom-value = "my-value";
            };
        };
}
```

Modules

Homes