

# Modularize Your NixOS Configuration

At this point, the skeleton of the entire system is configured. The current configuration structure in `/etc/nixos` should be as follows:

```
1  $ tree
2  .
3  ├── flake.lock
4  ├── flake.nix
5  ├── home.nix
6  └── configuration.nix
```

The functions of these four files are:

- `flake.lock` : An automatically generated version-lock file that records all input sources, hash values, and version numbers of the entire flake to ensure reproducibility.
- `flake.nix` : The entry file that will be recognized and deployed when executing `sudo nixos-rebuild switch`. See [Flakes - NixOS Wiki](#) for all options of `flake.nix`.
- `configuration.nix` : Imported as a Nix module in `flake.nix`, all system-level configuration is currently written here. See [Configuration - NixOS Manual](#) for all options of `configuration.nix`.
- `home.nix` : Imported by Home-Manager as the configuration of the user `ryan` in `flake.nix`, containing all of `ryan`'s configuration and managing `ryan`'s home folder. See [Appendix A. Configuration Options - Home-Manager](#) for all options of `home.nix`.

By modifying these files, you can declaratively change the system and home directory status.

However, as the configuration grows, relying solely on `configuration.nix` and `home.nix` can lead to bloated and difficult-to-maintain files. A better solution is to use the Nix module system to split the configuration into multiple Nix modules and write them in a classified manner.

The Nix module system provides a parameter, `imports`, which accepts a list of `.nix` files and merges all the configuration defined in these files into the current Nix module. Note that `imports` will not simply overwrite duplicate configuration but handle it more reasonably. For example, if `program.packages = [...]` is defined in multiple modules, then `imports` will merge all `program.packages` defined in all Nix modules into one list. Attribute sets can also be merged correctly. The specific behavior can be explored by yourself.

I only found a description of `imports` in [Nixpkgs-Unstable Official Manual - evalModules](#)

**Parameters:** A list of modules. These are merged together to form the final configuration. It's a bit ambiguous...

With the help of `imports`, we can split `home.nix` and `configuration.nix` into multiple Nix modules defined in different `.nix` files. Lets look at an example module `packages.nix`:

```
1  {
2    config,
3    pkgs,
4    ...
5  }: {
6    imports = [
7      (import ./special-fonts-1.nix {inherit config pkgs;}) # (1)
8      ./special-fonts-2.nix # (2)
9    ];
10
11   fontconfig.enable = true;
12 }
```

This module loads two other modules in the `imports` section, namely `special-fonts-1.nix` and `special-fonts-2.nix`. Both files are modules themselves and look similar to this.

```
1  { config, pkgs, ... }: {
2    # Configuration stuff ...
3  }
```

Both import statements above are equivalent in the parameters they receive:

- Statement (1) imports the function in `special-fonts-1.nix` and calls it by passing `{config = config; pkgs = pkgs}`. Basically using the return value of the call (another partial configuration [attribute set]) inside the `imports` list.
- Statement (2) defines a path to a module, whose function Nix will load *automatically* when assembling the configuration `config`. It will pass all matching arguments from the function in `packages.nix` to the loaded function in `special-fonts-2.nix` which results in `import ./special-fonts-2.nix {config = config; pkgs = pkgs}`.

Here is a nice starter example of modularizing the configuration, Highly recommended:

- [Misterio77/nix-starter-configs](#)

A more complicated example, [ryan4yin/nix-config/i3-kickstarter](#) is the configuration of my previous NixOS system with the i3 window manager. Its structure is as follows:

shell

```
1  └─ flake.lock
2  └─ flake.nix
3  └─ home
4  │   └─ default.nix      # here we import all submodules by imports = [
5  │   └─ fcitx5          # fcitx5 input method's configuration
6  │   │   └─ default.nix
7  │   │   └─ rime-data-flypy
8  │   └─ i3              # i3 window manager's configuration
9  │   │   └─ config
10 │   │   └─ default.nix
11 │   │   └─ i3blocks.conf
12 │   │   └─ keybindings
13 │   │   └─ scripts
14 │   └─ programs
15 │   │   └─ browsers.nix
16 │   │   └─ common.nix
17 │   │   └─ default.nix # here we import all modules in programs folder b
18 │   │   └─ git.nix
19 │   │   └─ media.nix
20 │   │   └─ vscode.nix
21 │   │   └─ xdg.nix
22 │   └─ rofi            # rofi launcher's configuration
23 │   │   └─ configs
24 │   │   │   └─ arc_dark_colors.rasi
25 │   │   │   └─ arc_dark_transparent_colors.rasi
26 │   │   │   └─ power-profiles.rasi
27 │   │   │   └─ powermenu.rasi
28 │   │   │   └─ rofidmenu.rasi
29 │   │   │   └─ rofikeyhint.rasi
30 │   │   └─ default.nix
31 │   └─ shell          # shell/terminal related configuration
32 │       └─ common.nix
33 │       └─ default.nix
34 │       └─ nushell
35 │       │   └─ config.nu
36 │       │   └─ default.nix
37 │       │   └─ env.nu
38 │       └─ starship.nix
39
```

```

40 |         └─ terminals.nix
41 | └─ hosts
42 |   └─ msi-rtx4090      # My main machine's configuration
43 |     └─ default.nix   # This is the old configuration.nix, but most of tl
44 |       └─ hardware-configuration.nix # hardware & disk related configura
45 |     └─ my-nixos      # my test machine's configuration
46 |       └─ default.nix
47 |         └─ hardware-configuration.nix
48 | └─ modules           # some common NixOS modules that can be reused
49 |   └─ i3.nix
50 |     └─ system.nix
    └─ wallpaper.jpg   # wallpaper

```

There is no need to follow the above structure, you can organize your configuration in any way you like. The key is to use `imports` to import all the submodules into the main module.

---

## lib.mkOverride, lib.mkDefault, and lib.mkForce

In Nix, some people use `lib.mkDefault` and `lib.mkForce` to define values. These functions are designed to set default values or force values of options.

You can explore the source code of `lib.mkDefault` and `lib.mkForce` by running `nix repl -f '<nixpkgs>'` and then entering `:e lib.mkDefault`. To learn more about `nix repl`, type `:?` for the help information.

Here's the source code:

```

1         # .....
2
3         mkOverride = priority: content:
4             { _type = "override";
5               inherit priority content;
6             };
7
8         mkOptionDefault = mkOverride 1500; # priority of option defaults
9         mkDefault = mkOverride 1000; # used in config sections of non-user module
10        mkImageMediaOverride = mkOverride 60; # image media profiles can be deriv
11        mkForce = mkOverride 50;
12        mkVMOverride = mkOverride 10; # used by 'nixos-rebuild build-vm'
13

```

```
# .....
```

In summary, `lib.mkDefault` is used to set default values of options with a priority of 1000 internally, and `lib.mkForce` is used to force values of options with a priority of 50 internally. If you set a value of an option directly, it will be set with a default priority of 1000, the same as `lib.mkDefault`.

The lower the `priority` value, the higher the actual priority. As a result, `lib.mkForce` has a higher priority than `lib.mkDefault`. If you define multiple values with the same priority, Nix will throw an error.

Using these functions can be very helpful for modularizing the configuration. You can set default values in a low-level module (base module) and force values in a high-level module.

For example, in my configuration at [ryan4yin/nix-config/blob/c515ea9/modules/nixos/core-server.nix](https://github.com/ryan4yin/nix-config/blob/c515ea9/modules/nixos/core-server.nix), I define default values like this:

nix

```
1  { lib, pkgs, ... }:
2
3  {
4    # .....

```

Then, for my desktop machine, I override the value in [ryan4yin/nix-config/blob/c515ea9/modules/nixos/core-desktop.nix](https://github.com/ryan4yin/nix-config/blob/c515ea9/modules/nixos/core-desktop.nix) like this:

nix

```
1  { lib, pkgs, ... }:
2
3  {
4    # import the base module
5    imports = [
6      ./core-server.nix
7    ];
8
9    # override the default value defined in the base module
10
```

```
10     nixpkgs.config.allowUnfree = lib.mkForce true;
11
12
13     # .....
    }
```

---

## lib.mkOrder , lib.mkBefore , and lib.mkAfter

In addition to `lib.mkDefault` and `lib.mkForce` , there are also `lib.mkBefore` and `lib.mkAfter` , which are used to set the merge order of **list-type options**. These functions further contribute to the modularization of the configuration.

I haven't found the official documentation for list-type options, but I simply understand that they are types whose merge results are related to the order of merging. According to this understanding, both `list` and `string` types are list-type options, and these functions can indeed be used on these two types in practice.

As mentioned earlier, when you define multiple values with the same **override priority**, Nix will throw an error. However, by using `lib.mkOrder` , `lib.mkBefore` , or `lib.mkAfter` , you can define multiple values with the same override priority, and they will be merged in the order you specify.

To examine the source code of `lib.mkBefore` , you can run `nix repl -f '<nixpkgs>'` and then enter `:e lib.mkBefore` . To learn more about `nix repl` , type `:?` for the help information:

```
                                                                    nix
1     # .....
2
3     mkOrder = priority: content:
4         { _type = "order";
5           inherit priority content;
6         };
7
8     mkBefore = mkOrder 500;
9     defaultOrderPriority = 1000;
10    mkAfter = mkOrder 1500;
11
12    # .....
```

Therefore, `lib.mkBefore` is a shorthand for `lib.mkOrder 500`, and `lib.mkAfter` is a shorthand for `lib.mkOrder 1500`.

To test the usage of `lib.mkBefore` and `lib.mkAfter`, let's create a simple Flake project:

nix

```
1 # flake.nix
2 {
3   inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";
4   outputs = {nixpkgs, ...}: {
5     nixosConfigurations = {
6       "my-nixos" = nixpkgs.lib.nixosSystem {
7         system = "x86_64-linux";
8
9         modules = [
10          ({lib, ...}: {
11            programs.bash.shellInit = lib.mkBefore ''
12              echo 'insert before default'
13            '';
14            programs.zsh.shellInit = lib.mkBefore "echo 'insert before defa
15            nix.settings.substituters = lib.mkBefore [
16              "https://nix-community.cachix.org"
17            ];
18          })
19
20          ({lib, ...}: {
21            programs.bash.shellInit = lib.mkAfter ''
22              echo 'insert after default'
23            '';
24            programs.zsh.shellInit = lib.mkAfter "echo 'insert after defaul
25            nix.settings.substituters = lib.mkAfter [
26              "https://ryan4yin.cachix.org"
27            ];
28          })
29
30          ({lib, ...}: {
31            programs.bash.shellInit = ''
32              echo 'this is default'
33            '';
34            programs.zsh.shellInit = "echo 'this is default'";
35            nix.settings.substituters = [
36              "https://nix-community.cachix.org"
37            ];
38          })
39
40          ]
41        }
42      }
43    }
44  }
```

```

39         ];
40     };
41 };
42 };
43 }

```

The flake above contains the usage of `lib.mkBefore` and `lib.mkAfter` on multiline strings, single-line strings, and lists. Let's test the results:

```

                                                                    bash
1   # Example 1: multiline string merging
2   > echo $(nix eval .#nixosConfigurations.my-nixos.config.programs.bash.shellI
3   trace: warning: system.stateVersion is not set, defaulting to 23.11. Read w
4   n.
5   "echo 'insert before default'
6
7   echo 'this is default'
8
9   if [ -z \"$_NIXOS_SET_ENVIRONMENT_DONE\" ]; then
10  . /nix/store/60882lm9znqdmbsxqsd5bgnb7gybaf2-set-environment
11  fi
12
13
14
15  echo 'insert after default'
16  "
17
18  # example 2: single-line string merging
19  > echo $(nix eval .#nixosConfigurations.my-nixos.config.programs.zsh.shellI
20  "echo 'insert before default';
21  echo 'this is default';
22  echo 'insert after default';"
23
24  # Example 3: list merging
25  > nix eval .#nixosConfigurations.my-nixos.config.nix.settings.substituters
26  [ "https://nix-community.cachix.org" "https://nix-community.cachix.org" "ht

```

As you can see, `lib.mkBefore` and `lib.mkAfter` can define the order of merging of multiline strings, single-line strings, and lists. The order of merging is the same as the order of definition.

For a deeper introduction to the module system, see [Module System & Custom Options](#).

## References

- [Nix modules: Improving Nix's discoverability and usability](#)
  - [Module System - Nixpkgs](#)
-