PDF

Module system deep dive

Contents

- 2.1. Overview
- 2.2. The empty module
- 2.3. Declaring options
- 2.4. Evaluating modules
- 2.5. Type checking
- 2.6. Interlude: reproducible scripts
- 2.7. Declaring more options
- 2.8. Dependencies between options
- 2.9. Conditional definitions
- 2.10. Default values
- 2.11. Wrapping shell commands
- 2.12. Splitting modules
- 2.13. The `submodule` type
- 2.14. Defining options in other modules
- 2.15. Nested submodules
- 2.16. The `strMatching` type
- 2.17. Functions as submodule arguments
- 2.18. The `either` and `enum` types
- 2.19. The `pathType` submodule
- 2.20. The `between` constraint on integer values
- 2.21. The `pathStyle` submodule
- 2.22. Path styling: color
- 2.23. Further styling
- 2.24. Wrapping up

Or: *Wrapping the world in modules*

Skip to main content

In this tutorial you will follow an extensive demonstration of how to wrap an existing API with Nix modules.

2.1. Overview

This tutorial follows @infinisil's presentation on modules (source) for participants of Summer of Nix 2021.

It may help playing it alongside this tutorial to better keep track of changes to the code you will work on.

2.1.1. What will you learn?

You'll write modules to interact with the Google Maps API, declaring module options which represent map geometry, location pins, and more.

During the tutorial, you will first write some *incorrect* configurations, creating opportunities to discuss the resulting error messages and how to resolve them, particularly when discussing type checking.

2.1.2. What do you need?

You will use two helper scripts for this exercise. Download `map.sh` and `geocode.sh` to your working directory.

Warning

To run the examples in this tutorial, you will need a Google API key in `$XDG_DATA_HOME/google-api/key`.

2.2. The empty module

Write the following into a file called `default.nix`:

default.nix

Skip to main content

```
}
```

2.3. Declaring options

We will need some helper functions, which will come from the Nixpkgs library, which is passed by the module system as `lib`:

default.nix

```
- { ... }:
+ { lib, ... }:
{

}
```

Using `lib.mkOption`, declare the `scripts.output` option to have the type `lines`:

default.nix

```
 { lib, ... }: {

+ options = {
+   scripts.output = lib.mkOption {
+     type = lib.types.lines;
+   };
+ };

 }
```

The `lines` type means that the only valid values are strings, and that multiple definitions should be joined with newlines.

Note

The name and attribute path of the option is arbitrary. Here we use `scripts`, because we will add another script later, and call this one `output`, because it will output the resulting map.

Skip to main content

2.4. Evaluating modules

Write a new file `eval.nix` to call `lib.evalModules` and evaluate the module in `default.nix`:

eval.nix

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
pkgs.lib.evalModules {
  modules = [
    ./default.nix
  ];
}
```

Run the following command:

⚠ **Warning**

This will result in an error.

```
nix-instantiate --eval eval.nix -A config.scripts.output
```

Detailed explanation

The error message indicates that the `scripts.output` option is used but not defined: a value must be set for the option before accessing it. You will do this in the next steps.

2.5. Type checking

As previously mentioned, the `lines` type only permits string values.

⚠ **Warning**

In this section, you will set an invalid value and encounter a type error.

What happens if you instead try to assign an integer to the option?

Skip to main content

default.nix

```
 { lib, ... }: {

  options = {
    scripts.output = lib.mkOption {
      type = lib.types.lines;
    };
  };

+ config = {
+   scripts.output = 42;
+ };
 }
```

Now try to execute the previous command, and witness your first module error:

```
$ nix-instantiate --eval eval.nix -A config.scripts.output
error:
...
       error: A definition for option `scripts.output' is not of type `strings conc
       - In `/home/nix-user/default.nix': 42
```

The definition `scripts.output = 42;` caused a type error: integers are not strings concatenated with the newline character.

To make this module pass the type checks and successfully evaluate the `scripts.output` option, you will now assign a string to `scripts.output`.

In this case, you will assign a shell command that runs the `map` script in the current directory. That in turn calls the Google Maps Static API to generate a world map. The output is passed on to display it with `feh`, a minimalistic image viewer.

Update `default.nix` by changing the value of `scripts.output` to the following string:

default.nix

```
   config = {
-    scripts.output = 42;
+    scripts.output = ''
+      ./map.sh size=640x640 scale=2 | feh -
+    '';
   };
```

Skip to main content

2.6. Interlude: reproducible scripts

That simple command will likely not work as intended on your system, as it may lack the required dependencies (curl and feh). We can solve this by packaging the raw `map` script with `pkgs.writeShellApplication`.

First, make available a `pkgs` argument in your module evaluation by adding a module that sets `config._module.args`:

eval.nix

```
 pkgs.lib.evalModules {
   modules = [
+    ({ config, ... }: { config._module.args = { inherit pkgs; }; })
     ./default.nix
   ];
 }
```

Note

This mechanism is currently only documented in the module system code, and that documentation is incomplete and out of date.

Then change `default.nix` to have the following contents:

default.nix

```
{ pkgs, lib, ... }: {

  options = {
    scripts.output = lib.mkOption {
      type = lib.types.package;
    };
  };

  config = {
    scripts.output = pkgs.writeShellApplication {
      name = "map";
      runtimeInputs = with pkgs; [ curl feh ];
      text = ''
        ${./map.sh} size=640x640 scale=2 | feh -
      '';
    };
  };
}
```

Skip to main content

This will access the previously added `pkgs` argument so we can use dependencies, and copy the `map` file in the current directory into the Nix store so it's available to the wrapped script, which will also live in the Nix store.

Run the script with:

```
nix-build eval.nix -A config.scripts.output
./result/bin/map
```

To iterate more quickly, open a new terminal and set up `entr` to re-run the script whenever any source file in the current directory changes:

```
nix-shell -p entr findutils bash --run \
  "ls *.nix | \
   entr -rs ' \
     nix-build eval.nix -A config.scripts.output --no-out-link \
     | xargs printf -- \"%s/bin/map\" \
     | xargs bash \
   ' \
  "
```

This command does the following:

- List all `.nix` files
- Make `entr` watch them for changes. Terminate the invoked command on each change with `-r`.
- On each change:
  - Run the `nix-build` invocation as above, but without adding a `./result` symlink
  - Take the resulting store path and append `/bin/map` to it
  - Run the executable at the path constructed this way

2.7. Declaring more options

Rather than setting all script parameters directly, we will to do that through the module system. This will not just add some safety through type checking, but also allow to build abstractions to manage growing complexity and changing requirements.

Let's begin by introducing another option, `requestParams`, which will represent the parameters of the request made to the Google Maps API.

Skip to main content

Instead of `lines`, in this case you will want the type of the list elements to be `str`, a generic string type.

The difference between `str` and `lines` is in their merging behavior: Module option types not only check for valid values, but also specify how multiple definitions of an option are to be combined into one.

- For `lines`, multiple definitions get merged by concatenation with newlines.
- For `str`, multiple definitions are not allowed. This is not a problem here, since one can't define a list element multiple times.

Make the following additions to your `default.nix` file:

default.nix

```
     scripts.output = lib.mkOption {
       type = lib.types.package;
     };
+
+    requestParams = lib.mkOption {
+      type = lib.types.listOf lib.types.str;
+    };
   };

   config = {
     scripts.output = pkgs.writeShellApplication {
       name = "map";
       runtimeInputs = with pkgs; [ curl feh ];
       text = ''
         ${./map.sh} size=640x640 scale=2 | feh -
       '';
     };
+
+    requestParams = [
+      "size=640x640"
+      "scale=2"
+    ];
   };
 }
```

2.8. Dependencies between options

A given module generally declares one option that produces a result to be used elsewhere, in this case `scripts.output`.

Options can depend on other options, making it possible to build more useful abstractions.

Skip to main content

Here, we want the `scripts.output` option to use the values of `requestParams` as arguments to the `./map` script.

2.8.1. Accessing option values

To make option values available to a module, the arguments of the function declaring the module must include the `config` attribute.

Update `default.nix` to add the `config` attribute:

default.nix

```
-{ pkgs, lib, ... }: {
+{ pkgs, lib, config, ... }: {
```

When a module that sets options is evaluated, the resulting values can be accessed by their corresponding attribute names under `config`.

Note

Option values can't be accessed directly from the same module.

The module system evaluates all modules it receives, and any of them can define a particular option's value. What happens when an option is set by multiple modules is determined by that option's type.

Warning

The `config` *argument* is **not** the same as the `config` *attribute*:

- The `config` *argument* holds the result of the module system's lazy evaluation, which takes into account all modules passed to `evalModules` and their `imports`.
- The `config` *attribute* of a module exposes that particular module's option values to the module system for evaluation.

Now make the following changes to `default.nix`:

default.nix

Skip to main content

```
    config = {
      scripts.output = pkgs.writeShellApplication {
        name = "map";
        runtimeInputs = with pkgs; [ curl feh ];
        text = ''
-           ${./map} size=640x640 scale=2 | feh -
+           ${./map} ${lib.concatStringsSep " "
+             config.requestParams} | feh -
        '';
```

Here, the value of the `config.requestParams` attribute is populated by the module system based on the definitions in the same file.

Note

Lazy evaluation in the Nix language allows the module system to make a value available in the `config` argument passed to the module which defines that value.

`lib.concatStringsSep " "` is then used to join each list element from the value of `config.requestParams` into a single string, with the list elements of `requestParams` separated by a space character.

The result of this represents the list of command line arguments to pass to the `./map` script.

2.9. Conditional definitions

Sometimes, you will want option values to be, well, optional. This can be useful when defining a value for an option is not required, as in the following case.

You will define a new option, `map.zoom`, to control the zoom level of the map. The Google Maps API will infer a zoom level if no corresponding argument is passed, a situation you can represent with the `nullOr <type>`, which represents values of type `<type>` or `null`. This *does not* automatically mean that when the option isn't defined, the value of such an option is `null` – we still need to define a default value.

Add the `map` attribute set with the `zoom` option into the top-level `options` declaration, like so:

default.nix

```
      requestParams = lib.mkOption {
```

Skip to main content

```
+
+    map = {
+      zoom = lib.mkOption {
+        type = lib.types.nullOr lib.types.int;
+        default = null;
+      };
+    };
   };
```

To make use of this, use the `mkIf <condition> <definition>` function, which only adds the definition if the condition evaluates to `true`. Make the following additions to the `requestParams` list in the `config` block:

default.nix

```
      requestParams = [
        "size=640x640"
        "scale=2"
+       (lib.mkIf (config.map.zoom != null)
+         "zoom=${toString config.map.zoom}")
      ];
    };
```

This will only add a `zoom` parameter to the script invocation if the value of `config.map.zoom` is not `null`.

2.10. Default values

Let's say that in our application we want to have a different default behavior that sets the zoom level to `10`, such that automatic zooming has to be enabled explicitly.

This can be done with the `default` argument to `mkOption`. Its value will be used if the value of the option declaring it is not specified otherwise.

Add the corresponding line:

default.nix

```
      map = {
        zoom = lib.mkOption {
          type = lib.types.nullOr lib.types.int;
+         default = 10;
        };
      };
    };
```

Skip to main content

2.11. Wrapping shell commands

You have now declared options controlling the map dimensions and zoom level, but have not provided a way to specify where the map should be centered.

Add the `center` option now, possibly with your own location as default value:

default.nix

```
          type = lib.types.nullOr lib.types.int;
          default = 10;
        };
+
+       center = lib.mkOption {
+         type = lib.types.nullOr lib.types.str;
+         default = "switzerland";
+       };
      };
    };
```

To implement this behavior, you will use the `geocode` utility, which turns location names into coordinates. There are multiple ways of making a new package accessible, but as an exercise, you will add it as an option in the module system.

First, add a new option to accommodate the package:

default.nix

```
    options = {
      scripts.output = lib.mkOption {
        type = lib.types.package;
      };
+
+     scripts.geocode = lib.mkOption {
+       type = lib.types.package;
+     };
```

Then define the value for that option where you make the raw script reproducible by wrapping a call to it in `writeShellApplication`:

default.nix

```
    config = {
+     scripts.geocode = pkgs.writeShellApplication {
+       name = "geocode";
```

Skip to main content

```
+     };
+
      scripts.output = pkgs.writeShellApplication {
        name = "map";
        runtimeInputs = with pkgs; [ curl feh ];
```

Add another `mkIf` call to the list of `requestParams` now where you access the wrapped package through `config.scripts.geocode`, and run the executable `/bin/geocode` inside:

default.nix

```
          "scale=2"
          (lib.mkIf (config.map.zoom != null)
            "zoom=${toString config.map.zoom}")
+         (lib.mkIf (config.map.center != null)
+           "center=\"$(${config.scripts.geocode}/bin/geocode ${
+             lib.escapeShellArg config.map.center
+           })\"")
        ];
      };
```

This time, you've used `escapeShellArg` to pass the `config.map.center` value as a command-line argument to `geocode`, string interpolating the result back into the `requestParams` string which sets the `center` value.

Wrapping shell command execution in Nix modules is a helpful technique for controlling system changes, as it uses the more ergonomic attributes and values interface rather than dealing with the peculiarities of escaping manually.

2.12. Splitting modules

The module schema includes the `imports` attribute, which allows incorporating further modules, for example to split a large configuration into multiple files.

In particular, this allows you to separate option declarations from where they are used in your configuration.

Create a new module, `marker.nix`, where you can declare options for defining location pins and other markers on the map:

marker.nix

```
{ lib, config, ... }: {
```

Skip to main content

```
}
```

Reference this new file in `default.nix` using the `imports` attribute:

default.nix

```
 { pkgs, lib, config, ... }: {

+  imports = [
+    ./marker.nix
+  ];
+
```

2.13. The `submodule` type

We want to set multiple markers on the map. A marker is a complex type with multiple fields.

This is where one of the most useful types included in the module system's type system comes into play: `submodule`. This type allows you to define nested modules with their own options.

Here, you will define a new `map.markers` option whose type is a list of submodules, each with a nested `location` type, allowing you to define a list of markers on the map.

Each assignment of markers will be type-checked during evaluation of the top-level `config`.

Make the following changes to `marker.nix`:

marker.nix

```
-{ pkgs, lib, config, ... }: {
+{ pkgs, lib, config, ... }:
+let
+  markerType = lib.types.submodule {
+    options = {
+      location = lib.mkOption {
+        type = lib.types.nullOr lib.types.str;
+        default = null;
+      };
+    };
+  };
+in {
+
+  options = {
+    map.markers = lib.mkOption {
```

Skip to main content

```
+     };
+   };
```

2.14. Defining options in other modules

Because of the way the module system composes option definitions, you can freely assign values to options defined in other modules.

In this case, you will use the `map.markers` option to produce and add new elements to the `requestParams` list, making your declared markers appear on the returned map – but from the module declared in `marker.nix`.

To implement this behavior, add the following `config` block to `marker.nix`:

marker.nix

```
+   config = {
+
+     map.markers = [
+       { location = "new york"; }
+     ];
+
+     requestParams = let
+       paramForMarker =
+         builtins.map (marker: "$(${config.scripts.geocode}/bin/geocode ${
+           lib.escapeShellArg marker.location})") config.map.markers;
+     in [ "markers=\"${lib.concatStringsSep "|" paramForMarker}\"" ];
+   };
```

Warning

To avoid confusion with the `map` option setting and the final `config.map` configuration value, here we use the `map` function explicitly as `builtins.map`.

Here, you again used `escapeShellArg` and string interpolation to generate a Nix string, this time producing a pipe-separated list of geocoded location attributes.

The `requestParams` value was also set to the resulting list of strings, which gets appended to the `requestParams` list defined in `default.nix`, thanks to the default merging behavior of the `list` type.

When defining multiple markers, determining an appropriate center or zoom level for the

Skip to main content

To achieve this, make the following additions to `marker.nix`, above the `requestParams` declaration:

marker.nix

```
+    map.center = lib.mkIf
+      (lib.length config.map.markers >= 1)
+      null;
+
+    map.zoom = lib.mkIf
+      (lib.length config.map.markers >= 2)
+      null;
+
     requestParams = let
       paramForMarker = marker:
         let
```

In this case, the default behavior of the Google Maps API when not passed a center or zoom level is to pick the geometric center of all the given markers, and to set a zoom level appropriate for viewing all markers at once.

2.15. Nested submodules

Next, we want to allow multiple named users to define a list of markers each.

For that you'll add a `users` option with type `lib.types.attrsOf <subtype>`, which will allow you to define `users` as an attribute set, whose values have type `<subtype>`.

Here, that subtype will be another submodule which allows declaring a departure marker, suitable for querying the API for the recommended route for a trip.

This will again make use of the `markerType` submodule, giving a nested structure of submodules.

To propagate marker definitions from `users` to the `map.markers` option, make the following changes.

In the `let` block:

marker.nix

```
+  userType = lib.types.submodule {
+    options = {
```

```
+         default = {};
+       };
+     };
+   };
+
  in {
```

This defines a submodule type for a user, with a `departure` option of type `markerType`.

In the `options` block, above `map.markers`:

marker.nix

```
+     users = lib.mkOption {
+       type = lib.types.attrsOf userType;
+     };
```

That allows adding a `users` attribute set to `config` in any submodule that imports `marker.nix`, where each attribute will be of type `userType` as declared in the previous step.

In the `config` block, above `map.center`:

marker.nix

```
    config = {

-     map.markers = [
-       { location = "new york"; }
-     ];
+     map.markers = lib.filter
+       (marker: marker.location != null)
+       (lib.concatMap (user: [
+         user.departure
+       ]) (lib.attrValues config.users));

      map.center = lib.mkIf
        (lib.length config.map.markers >= 1)
```

This takes all the `departure` markers from all users in the `config` argument, and adds them to `map.markers` if their `location` attribute is not `null`.

The `config.users` attribute set is passed to `attrValues`, which returns a list of values of each of the attributes in the set (here, the set of `config.users` you've defined), sorted alphabetically (which is how attribute names are stored in the Nix language).

Back in `default.nix`, the resulting `map.markers` option value is still accessed by

Skip to main content

calls the Google Maps API.

Defining the options in this way allows you to set multiple `users.<name>.departure.location` values and generate a map with the appropriate zoom and center, with pins corresponding to the set of `departure.location` values for *all* `users`.

In the 2021 Summer of Nix, this formed the basis of an interactive multi-person map demo.

2.16. The `strMatching` type

Now that the map can be rendered with multiple markers, it's time to add some style customizations.

To tell the markers apart, add another option to the `markerType` submodule, to allow labeling each marker pin.

The API documentation states that these labels must be either an uppercase letter or a number.

You can implement this with the `strMatching "<regex>"` type, where `<regex>` is a regular expression that will accept any matching values, in this case an uppercase letter or number.

In the `let` block:

marker.nix

```
          type = lib.types.nullOr lib.types.str;
          default = null;
        };
+
+       style.label = lib.mkOption {
+         type = lib.types.nullOr
+           (lib.types.strMatching "[A-Z0-9]");
+         default = null;
+       };
      };
    };
```

Again, `types.nullOr` allows for `null` values, and the default has been set to `null`.

In the `paramForMarker` function:

marker.nix

Skip to main content

```
     requestParams = let
+      paramForMarker = marker:
+        let
+          attributes =
+            lib.optional (marker.style.label != null)
+            "label:${marker.style.label}"
+            ++ [
+              "$(${config.scripts.geocode}/bin/geocode ${
+                lib.escapeShellArg marker.location
+              })"
+            ];
+        in "markers=\"${lib.concatStringsSep "|" attributes}\"";
+      in
+        builtins.map paramForMarker config.map.markers;
```

Note how we now create a unique `marker` for each user by concatenating the `label` and `location` attributes together, and assigning them to the `requestParams`. The label for each `marker` is only propagated to the CLI parameters if `marker.style.label` is set.

2.17. Functions as submodule arguments

Right now, if a label is not explicitly set, none will show up. But since every `users` attribute has a name, we could use that as an automatic value instead.

This `firstUpperAlnum` function allows you to retrieve the first character of the username, with the correct type for passing to `departure.style.label`:

marker.nix

```
{ lib, config, ... }:
 let
+   # Returns the uppercased first letter
+   # or number of a string
+   firstUpperAlnum = str:
+     lib.mapNullable lib.head
+     (builtins.match "[^A-Z0-9]*([A-Z0-9]).*"
+     (lib.toUpper str));

   markerType = lib.types.submodule {
     options = {
```

By transforming the argument to `lib.types.submodule` into a function, you can access arguments within it.

One special argument automatically available to submodules is `name`, which when used in marker.nix

```
-  userType = lib.types.submodule {
+  userType = lib.types.submodule ({ name, ... }: {
     options = {
       departure = lib.mkOption {
         type = markerType;
         default = {};
       };
     };
-  };
```

In this case, you don’t easily have access to the name from the marker submodules `label` option, where you otherwise could set a `default` value.

Instead you can use the `config` section of the `user` submodule to set a default, like so:

marker.nix

```
+
+    config = {
+      departure.style.label = lib.mkDefault
+        (firstUpperAlnum name);
+    };
+  });

 in {
```

Note

Module options have a *priority*, represented as an integer, which determines the precedence for setting the option to a particular value. When merging values, the priority with lowest numeric value wins.

The `lib.mkDefault` modifier sets the priority of its argument value to 1000, the lowest precedence.

This ensures that other values set for the same option will prevail.

2.18. The `either` and `enum` types

For better visual contrast, it would be helpful to have a way to change the *color* of a marker.

Skip to main content

- `either <this> <that>`, which takes two types as arguments, and allows either of them
- `enum [ <allowed values> ]`, which takes a list of allowed values, and allows any of them

In the `let` block, add the following `colorType` option, which can hold strings containing either some given color names or an RGB value add the new compound type:

marker.nix

```
      ...
      (builtins.match "[^A-Z0-9]*([A-Z0-9]).*"
      (lib.toUpper str));

+  # Either a color name or `0xRRGGBB`
+  colorType = lib.types.either
+    (lib.types.strMatching "0x[0-9A-F]{6}")
+    (lib.types.enum [
+      "black" "brown" "green" "purple" "yellow"
+      "blue" "gray" "orange" "red" "white" ]);
+
   markerType = lib.types.submodule {
     options = {
       location = lib.mkOption {
```

This allows either strings that match a 24-bit hexadecimal number or are equal to one of the specified color names.

At the bottom of the `let` block, add the `style.color` option and specify a default value:

marker.nix

```
          (lib.types.strMatching "[A-Z0-9]");
        default = null;
      };
+
+      style.color = lib.mkOption {
+        type = colorType;
+        default = "red";
+      };
    };
  };
```

Now add an entry to the `paramForMarker` list which makes use of the new option:

marker.nix

Skip to main content

```
              ++ [
+                 "color:${marker.style.color}"
                  "$(${config.scripts.geocode}/bin/geocode ${
                    lib.escapeShellArg marker.location
                  })"
```

In case you set many different markers, it would be helpful to have the ability to change their size individually.

Add a new `style.size` option to `marker.nix`, allowing you to choose from the set of pre-defined sizes:

marker.nix

```
          type = colorType;
          default = "red";
        };
+
+       style.size = lib.mkOption {
+         type = lib.types.enum
+           [ "tiny" "small" "medium" "large" ];
+         default = "medium";
+       };
      };
    };
```

Now add a mapping for the size parameter in `paramForMarker`, which selects an appropriate string to pass to the API:

marker.nix

```
      requestParams = let
        paramForMarker = marker:
          let
+           size = {
+             tiny = "tiny";
+             small = "small";
+             medium = "mid";
+             large = null;
+           }.${marker.style.size};
+
```

Finally, add another `lib.optional` call to the `attributes` string, making use of the selected size:

marker.nix

Skip to main content

```
              attributes =
                lib.optional
                  (marker.style.label != null)
                  "label:${marker.style.label}"
+               ++ lib.optional
+                 (size != null)
+                 "size:${size}"
                ++ [
                  "color:${marker.style.color}"
                  "$(${config.scripts.geocode}/bin/geocode ${
```

2.19. The `pathType` submodule

So far, you've created an option for declaring a *departure* marker, as well as several options for configuring the marker's visual representation.

Now we want to compute and display a route from the user's location to some destination.

The new option defined in the next section will allow you to set an *arrival* marker, which together with a departure allows you to draw *paths* on the map using the new module defined below.

To start, create a new `path.nix` file with the following contents:

path.nix

```
{ lib, config, ... }:
let
  pathType = lib.types.submodule {
    options = {
      locations = lib.mkOption {
        type = lib.types.listOf lib.types.str;
      };
    };
  };
in
{
  options = {
    map.paths = lib.mkOption {
      type = lib.types.listOf pathType;
    };
  };
  config = {
    requestParams =
      let
        attrForLocation = loc:
          "$(${config.scripts.geocode}/bin/geocode ${lib.escapeShellArg loc})";
        paramForPath = path:
```

Skip to main content

```
              builtins.map attrForLocation path.locations;
          in
          ''path="${lib.concatStringsSep "|" attributes}"'';
      in
      builtins.map paramForPath config.map.paths;
  };
}
```

The `path.nix` module declares an option for defining a list of paths on our `map`, where each path is a list of strings for geographic locations.

In the `config` attribute we augment the API call by setting the `requestParams` option value with the coordinates transformed appropriately, which will be concatenated with request parameters set elsewhere.

Now import this new `path.nix` module from your `marker.nix` module:

marker.nix

```
 in {

+  imports = [
+    ./path.nix
+  ];
+
   options = {

     users = lib.mkOption {
```

Copy the `departure` option declaration to a new `arrival` option in `marker.nix`, to complete the initial path implementation:

marker.nix

```
         type = markerType;
         default = {};
       };
+
+      arrival = lib.mkOption {
+        type = markerType;
+        default = {};
+      };
     };
```

Next, add an `arrival.style.label` attribute to the `config` block, mirroring the `departure.style.label`:

Skip to main content

marker.nix

```
      config = {
        departure.style.label = lib.mkDefault
          (firstUpperAlnum name);
+       arrival.style.label = lib.mkDefault
+         (firstUpperAlnum name);
      };
    });
```

Finally, update the return list in the function passed to `concatMap` in `map.markers` to also include the `arrival` marker for each user:

marker.nix

```
      map.markers = lib.filter
        (marker: marker.location != null)
        (lib.concatMap (user: [
-         user.departure
+         user.departure user.arrival
        ]) (lib.attrValues config.users));

      map.center = lib.mkIf
```

Now you have the basis to define paths on the map, connecting pairs of departure and arrival points.

In the path module, define a path connecting every user's departure and arrival locations:

path.nix

```
    config = {
+
+     map.paths = builtins.map (user: {
+       locations = [
+         user.departure.location
+         user.arrival.location
+       ];
+     }) (lib.filter (user:
+       user.departure.location != null
+       && user.arrival.location != null
+     ) (lib.attrValues config.users));
+
      requestParams = let
        attrForLocation = loc:
          "$(geocode ${lib.escapeShellArg loc})";
```

Skip to main content

A path is valid only if the `departure` and `arrival` attributes are set for that user.

2.20. The `between` constraint on integer values

Your users have spoken, and they demand the ability to customize the styles of their paths with a `weight` option.

As before, you'll now declare a new submodule for the path style.

While you could also directly declare the `style.weight` option, in this case you should use the submodule to be able reuse the path style type later.

Add the `pathStyleType` submodule option to the `let` block in `path.nix`:

path.nix

```
 { lib, config, ... }:
 let
+
+  pathStyleType = lib.types.submodule {
+    options = {
+      weight = lib.mkOption {
+        type = lib.types.ints.between 1 20;
+        default = 5;
+      };
+    };
+  };
+
   pathType = lib.types.submodule {
```

Note

The `ints.between <lower> <upper>` type allows integers in the given (inclusive) range.

The path weight will default to 5, but can be set to any integer value in the 1 to 20 range, with higher weights producing thicker paths on the map.

Now add a `style` option to the `options` set further down the file:

path.nix

Skip to main content

```
      options = {
        locations = lib.mkOption {
          type = lib.types.listOf lib.types.str;
        };
+
+       style = lib.mkOption {
+         type = pathStyleType;
+         default = {};
+       };
      };

    };
```

Finally, update the `attributes` list in `paramForPath`:

path.nix

```
        paramForPath = path:
          let
            attributes =
-             builtins.map attrForLocation path.locations;
+             [
+               "weight:${toString path.style.weight}"
+             ]
+             ++ builtins.map attrForLocation path.locations;
          in "path=\"${lib.concatStringsSep "|" attributes}\"";
```

2.21. The `pathStyle` submodule

Users still can't actually customize the path style yet. Introduce a new `pathStyle` option for each user.

The module system allows you to declare values for an option multiple times, and if the types permit doing so, takes care of merging each declaration's values together.

This makes it possible to have a definition for the `users` option in the `marker.nix` module, as well as a `users` definition in `path.nix`:

path.nix

```
  in {
    options = {
+
+     users = lib.mkOption {
+       type = lib.types.attrsOf (lib.types.submodule {
```

Skip to main content

```
+          default = {};
+        };
+      });
+    };
+
    map.paths = lib.mkOption {
      type = lib.types.listOf pathType;
    };
```

Then add a line using the `user.pathStyle` option in `map.paths` where each user's paths are processed:

path.nix

```
          user.departure.location
          user.arrival.location
        ];
+       style = user.pathStyle;
    }) (lib.filter (user:
      user.departure.location != null
      && user.arrival.location != null
```

2.22. Path styling: color

As with markers, paths should have customizable colors.

You can accomplish this using types you've already encountered by now.

Add a new `colorType` block to `path.nix`, specifying the allowed color names and RGB/RGBA hexadecimal values:

path.nix

```
{ lib, config, ... }:
let

+  # Either a color name, `0xRRGGBB` or `0xRRGGBBAA`
+  colorType = lib.types.either
+    (lib.types.strMatching "0x[0-9A-F]{6}[0-9A-F]{2}?")
+    (lib.types.enum [
+      "black" "brown" "green" "purple" "yellow"
+      "blue" "gray" "orange" "red" "white"
+    ]);
+
   pathStyleType = lib.types.submodule {
```

Skip to main content

path.nix

```
          type = lib.types.ints.between 1 20;
          default = 5;
        };
+
+       color = lib.mkOption {
+         type = colorType;
+         default = "blue";
+       };
      };
    };
```

Finally, add a line using the `color` option to the `attributes` list:

path.nix

```
            attributes =
              [
                "weight:${toString path.style.weight}"
+               "color:${path.style.color}"
              ]
              ++ map attrForLocation path.locations;
          in "path=${
```

2.23. Further styling

Now that you've got this far, to further improve the aesthetics of the rendered map, add another style option allowing paths to be drawn as *geodesics*, the shortest "as the crow flies" distance between two points on Earth.

Since this feature can be turned on or off, you can do this using the `bool` type, which can be `true` or `false`.

Make the following changes to `path.nix` now:

path.nix

```
          type = colorType;
          default = "blue";
        };
+
+       geodesic = lib.mkOption {
+         type = lib.types.bool;
+         default = false;
```

Skip to main content

```
      };
    };
```

Make sure to also add a line to use that value in `attributes` list, so the option value is included in the API call:

path.nix

```
              [
                "weight:${toString path.style.weight}"
                "color:${path.style.color}"
+               "geodesic:${lib.boolToString path.style.geodesic}"
              ]
              ++ map attrForLocation path.locations;
          in "path=${
```

2.24. Wrapping up

In this tutorial, you've learned how to write custom Nix modules to bring external services under declarative control, with the help of several new utility functions from the Nixpkgs `lib`.

You defined several modules in multiple files, each with separate submodules making use of the module system's type checking.

These modules exposed features of the external API in a declarative way.

You can now conquer the world with Nix.