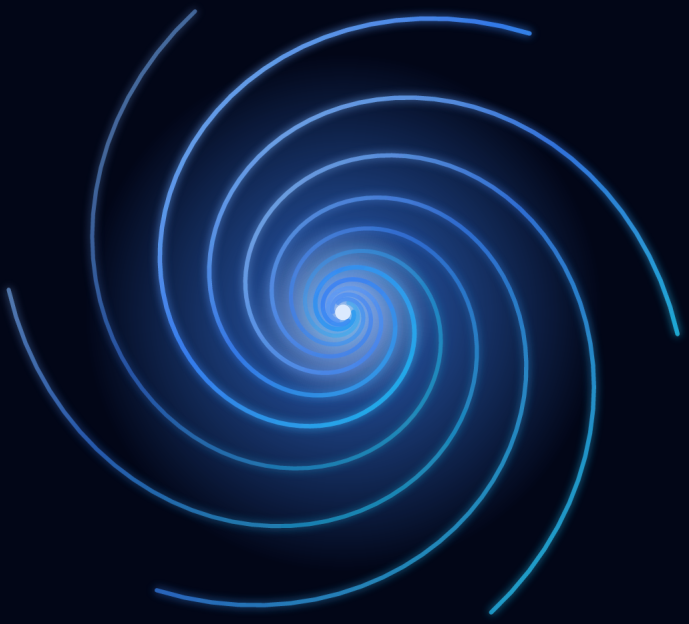
THE VORTEX MANUAL

TYPE-SAFE SCRIPTING WITH INTERPRETED HASKELL



CONTENTS

What is Vortex 3
Writing a DSL 6

WHAT IS VORTEX

Vortex is a toolkit for creating custom shells for use with interpreted Haskell. A Vortex Shell can be compiled with a choice of algebraic effects and corresponding interpreters, providing an execution environment with a heavily restricted set of side effects, whilst maintaining all of the syntactic benefits of Haskell.

Vortex can be used interactively or non-interactively, in a terminal, or over a websocket with SSH. Any do-block sent to a Vortex Shell will first type-check, then execute, and finally return actual Haskell values.

Vortex Shells are so secure that they can be directly exposed to the internet and execute arbitrary untrusted code, without the need for any sandboxing or process isolation.

REPLACE JSON, REST AND BASH WITH A VORTEX PROGRAM

That might seem like an odd statement. JSON, REST and Bash are completely different ideas – how can one idea replace and unify all three? Let's investigate it.

Each of the three meets a different need. JSON holds the config and data a program reads. REST is how one program offers its operations to another. Bash glues those operations together into something useful. A Vortex program meets all three needs at once, because it is a typed do-block, written in a DSL, that a host interprets. Let's take them one at a time.

CONFIG: A PROGRAM INSTEAD OF JSON

When we say JSON we really mean all config formats, so what is a config format? Config is anything you don't want to have to recompile the software in order to change. JSON, YAML, TOML and Dhall are all answers to that one need. The software is built once, and a file it reads at run time tells it what to do.

Config rarely stays plain data for long, though, and there is a tendency to end up "Programming in YAML". Real needs creep in: one setting for staging and another for production, defaults, blocks repeated for every service, a step that should only happen on one branch. So people bolt on templating, anchors, `if` and `when` keys, loops and string interpolation. A made-up example of where that road leads:

```
services:
{% for name in ["web", "worker"] %}
  - name: {{ name }}
    replicas: {% if env == "production" %}4{% else %}1{% endif %}
{% endfor %}
deploy:
  when: ${{ branch == "master" && !draft }}
```

That file is a program now, but in a language with no compiler, no types and no tests. A misspelt key or a condition of the wrong type shows up only when it runs, sometimes halfway through.

If config is going to be a program anyway, let's make it a real one. A Vortex host is compiled once, and the program it reads can change without rebuilding it. nixlab-runner reads its settings from `runner.vx` when it starts. The NixLab forge reads `ci-policy.vx` afresh each time it checks a push or a merge. horizon-config-shell reads a package set's `horizon.vx` and generates the set's Nix from it. Each of them loads the file through `Vortex.Elab.Load` in vortex-elab, which type-checks it with GHC against the config's DSL, then folds and forces it within 10 seconds. A config that does not check is refused as a whole, instead of being half-applied.

Here is the same config as a made-up Vortex program. The loop, the condition and the test are ordinary Haskell:

```
forM_ ["web", "worker"] $ \name ->
  service name $ do
    replicas (if env == Production then 4 else 1)
deploy (branch == "master" && not draft)
```

Nothing here is templated text. `replicas` takes a number, `env` is a value of the config's own type, and `deploy` takes a `Bool`, so writing `replicas "four"`, or comparing `env` with a misspelt name, is refused by GHC before anything runs.

The same idea covers the data a host sends back. A reply holds the `show` text of an ordinary Haskell value whose type derives `Show` and `Read`, so a client that has the DSL's types reads it back with `read` and gets the very value the host returned. Here is one entry of the live NixLab forge's answer to `:commands`:

```
Command {commandName = "branches", commandType = "ProjectPath -> Nix [GitRef]", commandDoc = Just "a project's branch names", commandEffects = ["Repo"]}
```

APIs: a host's effects instead of REST

A REST API fixes a set of endpoints and a message format. Here is a made-up one that asks a forge to keep a merge request rebased:

```
POST /api/projects/nixlab%2Fnixlab/merge_requests/30/auto_rebase

HTTP/1.1 202 Accepted   {"watching": true}
HTTP/1.1 409 Conflict   {"message": "conflicts in ChangeLog.md"}
HTTP/1.1 200 OK         {"watching": false}
```

The answer is really one of three things: the forge is watching, it is blocked by a conflict, or rebasing is off. JSON has no way to say "one of these", so each API invents its own: a status code here, a field that is sometimes there, a `"type"` key elsewhere. Nothing tells the client it has handled every case, and when the server adds a fourth, old clients carry on reading it as one of the first three. Composing calls is no better. Every step is its own round trip, and the client stitches the answers together by hand.

The forge's own operation answers with a sum type:

```
> :type autoRebase
Result "ProjectPath -> Number -> Nix AutoRebaseStatus"
> autoRebase "nixlab/nixlab" 30
Result "RebaseBlocked \"conflicts in ChangeLog.md\""
```

`AutoRebaseStatus` is `RebaseOff | RebaseWatching | RebaseBlocked String`, deriving `Show` and `Read`. A client with the DSL's types reads the reply back as that value and matches on it. With GHC's incomplete-patterns warning on, as it is under werrorwolf, a match that forgets a constructor fails the build, including one added later.

A Vortex host offers a language instead of endpoints. Its operations are effects, and a do-block can combine them however it likes, in a single request:

```
do
  bs <- branches "vortex/vortex-shell"
  pure (length bs)
```

```
Result "5"
```

GHC checks the do-block against the operations' types before anything runs. A do-block that names something the language does not offer, or uses an operation at the wrong type, is refused with GHC's own error and has no effect at all. `branches` takes a `ProjectPath`, so a number will not do:

```
> branches 42
Failed "<hint>:1:20: error: [GHC-39999]
    * No instance for `Num ProjectPath' arising from the literal `42'
```

The language cannot act on the world by itself, either. An effect such as `Roll` in spellbook only describes an operation, and the interpreter the host picks gives it meaning. The same do-block can roll a real die for one caller and a fixed one in a test. `readFile` may well be in scope, but it has the type `IO String`, while a do-block is elaborated at the DSL's own monad, where an `IO` action does not fit. On the live NixLab forge:

```
> readFile "/etc/passwd"
Failed "<hint>:1:11: error: [GHC-83865]
```

```
    * Couldn't match type `IO'
                    with `Polysemy.Internal.Sem
                           [Vortex.Effects.Say.Say, NixLab.Effects.Groups, ...
```

GLUE: A DO-BLOCK INSTEAD OF BASH

Bash is where we usually stitch these pieces together. A made-up loop that counts the tags of every project:

```
for p in $(curl -s "$FORGE/api/projects" | jq -r '.[].path'); do
  n=$(curl -s "$FORGE/api/projects/$p/tags" | jq length)
  echo "$p $n"
done
```

It looks fine, and it goes wrong quietly:

- The project list is often only the first page, and nothing says so.
- A path such as `vortex/livery` needs encoding in a URL. Unencoded, the request asks for the wrong thing.
- When a request fails, the server may send an error page or `{"message": "…"}`. `jq length` then fails with nothing captured, or counts the keys of the error object, and the loop prints a wrong number and carries on.
- Without `pipefail`, a pipe reports only its last command, so a failed `curl` piped into `jq` goes unnoticed.

A lost reply looks like an empty result, a wrong answer looks like a right one, and a script with a typo runs until it reaches the typo. In Vortex the glue is just another do-block, run by the host. This one asks the forge how many tags each project in the `vortex` group has, in one request:

```
do
  ps <- projects
  let inVortex pp = take 7 (projectPathText pp) == "vortex/"
  forM (filter inVortex (map projectPath ps)) $ \pp -> do
    ts <- tags pp
    pure (projectPathText pp, length ts)
```

```
Result "[(\"vortex/livery\",13),(\"vortex/livery-dsl\",5),...]"
```

If the program is wrong, it does not compile, so it never runs halfway. If it is right, the whole answer comes back as one typed value, or the request ends with a `Failed` that says why. Here GHC catches a sum of a number and a string before a single operation is performed:

```
do
  n <- length <$> branches "vortex/manual"
  pure (n + "one")
```

```
Failed "<hint>:3:13: error: [GHC-39999]
    * No instance for `GHC.Internal.Data.String.IsString Int'
```

So that is the one idea behind all three: a Vortex program, which is a typed do-block in a DSL that a host interprets. The config is a program the host checks when it loads it, the API is the DSL's effects, and the glue is a do-block that GHC has checked before it runs. The rest of this manual shows how the libraries fit together to make that work.

WRITING A DSL

A Vortex DSL is the language a do-block is written in. It is made of three things: a set of polysemy effects, the values those effects work on, and one scope module whose exports are the operations a caller can use. The DSL contains no `IO` at all. It describes what to do, and an interpreter, chosen by the host, decides what that means.

EFFECTS

Declare each effect as a GADT and let `makeSem` generate its operations. spellbook declares two of them, in `Spellbook.Effects`:

```
data Dice m a where
  Roll :: Int -> Dice m Int

makeSem ''Dice

data Who m a where
  Whoami :: Who m String

makeSem ''Who

type Spell = Sem '[Say, Dice, Who]
```

`Spell` is the monad every do-block is elaborated at. Its row names only the DSL's own effects, plus `Say` from vortex-effects. There is no `Embed IO` in it, so nothing a caller writes can perform `IO`. The host adds `IO` underneath the row when it interprets a program.

An effect can also refuse, with a typed error. livery-dsl's row, in `Livery.Style.Effects`, carries polysemy's `Error` for this:

```
data Style m a where
  Restyle :: Setting -> Style m ()
  Unstyle :: Style m ()
  Speak :: Line -> Style m ()

makeSem ''Style

type Livery = Sem '[Style, Error LiveryError]
```

THE SCOPE MODULE

The scope module is the DSL's operation surface. The shell imports it, together with the few other modules the shell names, and nothing else resolves. Its export list is everything a do-block may name without a qualifier. spellbook's scope module is `Spellbook`:

```
module Spellbook
  ( Spell
  , say
  , roll
  , whoami
  , help
  ) where

import           Spellbook.Effects  (Spell, roll, whoami)
import           Vortex.Effects.Say (say)
```

The effect constructors stay behind in `Spellbook.Effects`, which a caller never sees. Only the operations `makeSem` generated, and helpers such as `help`, are exported.

A scope module can also leave out the `Prelude` and export exactly the everyday names a do-block needs. `Livery.Dsl` does just that. It exports the `Livery` type, `String`, `Double` and `Int`, its verbs, and `pure`, `show`, `fmap` and `mapM_`, so a livery do-block cannot even name `readFile`. Each verb validates its argument before it reaches the effect:

```
accent :: String -> Livery ()
accent = colourVerb "accent" SetAccent

colourVerb :: String -> (Colour -> Setting) -> String -> Livery ()
colourVerb verb set written = either throw (restyle . set) (parseColour verb written)
```

Pure libraries can join a scope under an alias, as `(module, Just alias)` entries in the shell's scope. spellbook names `Data.List` as `List`, `Data.Char` as `Char`, `Data.Maybe` as `Maybe` and `Data.Map.Strict` as `Map`. `:commands` does not list a module under an alias. A module in the scope without an alias, on the other hand, is part of the surface, and `:commands` lists every function it exports.

VALUES

Give each value an operation takes or returns a type of its own, and derive `Show` and `Read` for every type a do-block can return, because a result travels as its derived `show` text. `OverloadedStrings` is on in every session, so a type with an `IsString` instance, which can validate its text, is written in a do-block as a plain string literal. On the NixLab forge, the `"vortex/vortex-shell"` in `branches "vortex/vortex-shell"` is a `ProjectPath`.

A PURE INTERPRETER

When a DSL has a pure meaning, its program can be folded to a value with no `IO` at all. That fold is the DSL's reference model, and its laws are checked against it. `runLivery` in `Livery.Style.Model` folds a livery do-block over the theme a page already has:

```
runLivery :: Theme -> Livery a -> Either LiveryError (Outcome a)
```

The fold is total, so every do-block yields either an `Outcome`, which holds the theme, the lines said and the value, or a typed `LiveryError`. A do-block that says more than `sayLimit`, which is 32 lines, ends with `TooMuchSaid` rather than a flood.

SETTINGS AS A PROGRAM

A DSL can also be a host's configuration, as the first chapter promised. `Vortex.Ws.Door.Settings` is the scope for a public WebSocket door's settings:

```
do
  origin "https://vortex.openfp.net"
  frameBytes 8192
  sessions 64 4
  peerRate 20 2
```

Its effect, in `Vortex.Ws.Door.Effects`, has a single operation that declares a statement, and the row `Door` is `Sem '[Configure]`. `Vortex.Ws.Door.Model` folds the statements to `DoorSettings`. It fills in a default for every verb except `origin`, and refuses a verb that is given twice. The host settles the program ahead of time with `settled`, which writes the statements as their `show` text, and reads them back with `readSettled`, which folds them again. The program in the file therefore stays the source of the settings, and what the running host reads is generated from it.

`Vortex.Elab.Load` is the general loader for files like this. A `Load` says which scope to use, which type to elaborate at, how source becomes an expression (by default `blockDo` in `Vortex.Elab.Block`, which reads statements written at the left margin as one do-block), how to fold the program and how to force the result. Folding and forcing have a time limit, 10 seconds by default. `loadProgram` answers with the forced value or a `LoadError`, which is one of `Unreadable`, `Unelaborated`, `Threw` or `Unfinished`. An asynchronous exception, such as an interrupt, is never turned into a `LoadError` and propagates as usual.

WHAT VORTEX-EFFECTS PROVIDES

vortex-effects holds two effects and one pure vocabulary that any DSL can share:

`Vortex.Effects.Say` `say :: Member Say r => String -> Sem r ()` sends a line to the caller. `runSay` interprets it through the session's `Emit` and sends at most the first 8,192 characters of each line.

`Vortex.Effects.Deferred` The `Deferred` effect describes subscriptions without running them. `attach`, `cancel` and `inspect` are operations whose interpreter belongs to the host, and `selectDelivery` picks the delivery an event triggers, with an identity that survives retries.

`Vortex.Effects.Atomic` Pure compare-and-swap plans over versioned cells. `spend`, `produce` and `intend` build a `Plan`, `atomically` joins plans together, and `commit` checks every claim before it returns a new `Ledger` or a `CommitError`.

Coming soon.