!!YPER

Type-safe, statically checked composition of HTTP servers

0.9.0

May 8, 2019

Contents

Type-safe, statically checked composition of HTTP servers

Hyper is an experimental middleware architecture for HTTP servers written in PureScript. Its main focus is correctness and type-safety, using type-level information to enforce correct composition and abstraction for web servers. The Hyper project is also a breeding ground for higher-level web server constructs, which tend to fall under the "framework" category.

Note: Until recently, most work on extensions and higher-level constructs have started their life in the main repository, but have then been separated into their own packages. More things might be separated in the near future, so prepare for unstability. Hyper itself should be considered experimental, for now.

This documentation is divided into sections, each being suited for different types of information you might be looking for.

- *Introduction* (page 3) describes the project itself; its motivations, goals, and relevant information for contributors to the project.

- *Core API* (page 5) is an introduction and reference of the Core API of Hyper, on which the higher-level features build.
- *Topic Guides* (page 8) explain how to solve specifics tasks of writing web servers using features of Hyper.
- *Tutorials* (page 12) are step-by-step guides on how to build complete web applications.
- *Extensions* (page 14) provides an overview of extensions to Hyper.
- *Frequently Asked Questions* (page 15) lists some of the questions and considerations about Hyper that come up frequently, answered for future reference.

Chapter 1

Introduction

Composing middleware in NodeJS is a risky business. They mutate the HTTP request and response objects freely, and are often dependent on each others' side-effects. There are no guarantees that you have stacked the middleware functions in a sensible order, and it is often the case, in the author's experience, that misconfigured middleware takes a lot of time and effort to debug.

1.1 Goals

The goal of *Hyper* is to make use of row polymorphism and other tasty type system features in PureScript to enforce correctly stacked middleware in HTTP server applications. All effects of middleware should be reflected in the types to ensure that common mistakes cannot be made. A few examples of such mistakes could be:

- Incorrect ordering of header and body writing
- Writing incomplete responses
- Writing multiple responses
- Trying to consume a non-parsed request body
- Consuming a request body parsed as the wrong type
- Incorrect ordering of, or missing, error handling middleware
- Incorrect ordering of middleware for sessions, authentication, authorization
- Missing authentication and/or authorization checks
- Linking, in an HTML anchor, to a resource that is not routed
- Posting, in an HTML form, to a resource that is not routed

Hyper aims to solve these issues, in part through its Core API for middleware, but also through a number of extensions for building safely composable and maintainable web applications.

1.2 Contributing

While Hyper is currently an experiment, and in constant flux, you are welcome to contribute. Please post ideas and start discussions using the issue tracker on GitHub[1]. You can also contact Oskar Wickström[2] directly for design discussions. If this project grows, we can setup a mailing list, or some other means of communication.

[1] https://github.com/owickstrom/hyper/issues
[2] https://wickstrom.tech/about.html

Please note that sending pull requests without first discussing the design is probably a waste of time, if not only fixing simple things like typos.

Chapter 2

Core API

This chapter explains the central components of Hyper, called the *Core API*. While focusing heavily on safety, Hyper tries to provide an open API that can support multiple PureScript backends, and different styles of web applications.

The design of Hyper is inspired by a number of projects. The middleware chain lends much from *Plug*, an abstract HTTP interface in Elixir, that enables various HTTP libraries to inter-operate. You might also find similarities with *connect* in NodeJS. On the type system side, Hyper tries to bring in ideas from *Haskell* and *Idris*, specifically the use of phantom types and GADTs to lift invariants to the type level and increase safety.

The central components of the Core API are:

2.1 Conn

A *Conn*, short for "connection", models the entirety of a connection between the HTTP server and the user agent, both request and response.

```
type Conn req res components =
  { request :: req
  , response :: res
  , components :: components
  }
```

The `request` and `response` hold the values representing the HTTP request and response, respectively. The purpose of the `components` field, however, is not that obvious. It is used for things not directly related to HTTP, but nonetheless related to the act of responding to the HTTP request. A middleware can add information to the Conn using components, like providing authentication or authorization values. The types of these components then becomes part of the Conn type, and you get stronger static guarantees when using the middleware.

2.2 Middleware

A *middleware* is an *indexed monadic action* transforming one `Conn` to another `Conn`. It operates in some base monad `m`, and is indexed by `i` and `o`, the *input* and *output* Conn types of the middleware action.

```
newtype Middleware m i o a = ...
```

The input and output type parameters are used to ensure that a Conn is transformed, and that side-effects are performed, correctly, throughout the middleware chain.

Being able to parameterize `Middleware` with some type `m`, you can customize the chain depending on the needs of your middleware and handlers. Applications can use monad transformers to track state, provide configuration, gather metrics, and much more, in the chain of middleware.

Middleware are composed using `ibind`, the indexed monadic version of `bind`. The simplest way of composing middleware is by chaining them with `:*>`, from `Control.Monad.Indexed`. See purescript-indexed-monad[3] for more information.

```
writeStatus statusOK
:*> closeHeaders
:*> respond "We're composing middleware!"
```

If you want to feed the return value of one middleware into another, use `:>>=`, the infix operator alias for `ibind`.

```
getUser :>>= renderUser
```

The *qualified do* syntax allows you to use `ibind` implicitly instead of the regular `bind`.

```
Middleware.do
  user <- getUser
  writeStatus statusOK
  closeHeaders
  respond ("User: " <> user.name)
```

2.3 Response State Transitions

The `response` field in the Conn is a value provided by the server backend. Middleware often constrain the `response` field to be a value implementing the `Response` type class. This makes it possible to use response-writing operations without depending on a specific server backend.

The state of a response is tracked in its last type parameter. This state tracking, and the type-indexed middleware using the response, guarantee correctness in response handling, preventing incorrect ordering of headers and body writes, incomplete responses, or other such mistakes. Let us have a look at the type signatures of some of response-writing functions in `Hyper.Response`.

We see that `headers` takes a foldable collection of headers, and gives back a middleware that, given a connection where headers are ready to be written (`HeadersOpen`), writes all specified headers, writes the separating CRLF before the HTTP body, and marks the state of the response as being ready to write the body (`BodyOpen`).

```
headers
  :: forall t m req res b c
  . ( Foldable f
    , Monad m
    , Response res m b
    )
  => f Header
  -> Middleware
    m
    (Conn req (res HeadersOpen) c)
    (Conn req (res BodyOpen) c)
    Unit
```

To be used in combination with `headers`, the `respond` function takes some `ResponseWritable b m r`, and gives back a middleware that, given a connection *where all headers have been written*, writes a response, and *marks the state of the response as ended*.

[3] https://pursuit.purescript.org/packages/purescript-indexed-monad/1.0.0

```
respond
  :: forall m r b req res c
   . ( Monad m
     , ResponseWritable b m r
     , Response res m b
     )
  => r
  -> Middleware
     m
     (Conn req (res BodyOpen) c)
     (Conn req (res ResponseEnded) c)
     Unit
```

The `ResponseWritable` type class describes types that can be written as responses. It takes three type parameters, where `b` is the target type, `m` is a base monad for the Middleware returned, and `r` is the original response type,

```
class ResponseWritable b m r where
  toResponse :: forall i. r -> Middleware m i i b
```

This mechanism allows servers to provide specific types for the response body, along with instances for common response types. When using the Node server, which has a response body type wrapping `Buffer`, you can still respond with a `String` or `HTML` value directly.

Aside from convenience in having a single function for most response types and servers, the polymorphism of `respond` lets middleware be decoupled from specific servers. It only requires an instance matching the response type used by the middleware and the type required by the server.

2.4 Servers

Although Hyper middleware can be applied directly to Conn values using `runMiddleware`, you likely want a *server* to run your middleware. Hyper tries to be as open as possible when it comes to servers – your application, and the middleware it depends on, should not be tied to a specific server. This allows for greater reuse and the ability to test entire applications without running the "real" server. Currently Hyper bundles a NodeJS server, described in *NodeJS* (page 10), as well as a test server, described in *Testing* (page 10).

Chapter 3

Topic Guides

The topic guides explain how to solve specifics tasks of writing web servers using features of Hyper. They are not full-length tutorials, but try to cover the details, and possible considerations, of a single subject. The following topic guides are available:

3.1 Request Body Reading

The `ReadableBody` type class has one operation, `readBody`, which supports different servers to provide different types of request body values.

```
class ReadableBody req m b | req -> b where
  readBody
    :: forall res c
     . Middleware
       m
       (Conn req res c)
       (Conn req res c)
       b
```

Given that there is an instance for the body `b`, and the return type `r`, we can use this middleware together with other middleware, like so:

```
onPost =
  readBody :>>=
  case _ of
    "" ->
      writeStatus statusBadRequest
      :*> closeHeaders
      :*> respond "... anyone there?"
    msg ->
      writeStatus statusBadRequest
      :*> closeHeaders
      :*> respond ("You said: " <> msg)
```

3.2 Forms

When working with form data, we often want to serialize and deserialize forms as custom data types, instead of working with the key-value pairs directly. The `ToForm` and `FromForm` type classes abstracts serialization and deserialization to form data, respectively.

We first declare our data types, and some instance which we will need later.

```
data MealType = Vegan | Vegetarian | Omnivore | Carnivore

derive instance genericMealType :: Generic MealType _
instance eqMealType :: Eq MealType where eq = genericEq
instance showMealType :: Show MealType where show = genericShow

newtype Order = Order { beers :: Int, meal :: MealType }
```

In this example we will only deserialize forms, and thus we only need the `FromForm` instance.

```
instance fromFormOrder :: FromForm Order where
  fromForm form = do
    beers <- required "beers" form >>= parseBeers
    meal <- required "meal" form >>= parseMealType
    pure (Order { beers: beers, meal: meal })
    where
      parseBeers s =
        maybe
        (throwError ("Invalid number: " <> s))
        pure
        (Int.fromString s)

      parseMealType =
        case _ of
          "Vegan" -> pure Vegan
          "Vegetarian" -> pure Vegetarian
          "Omnivore" -> pure Omnivore
          "Carnivore" -> pure Carnivore
          s -> throwError ("Invalid meal type: " <> s)
```

Now we are ready to write our handler. We use `parseFromForm` to get a value of type `Either String Order`, where the `String` explains parsing errors. By pattern matching using record field puns, we extract the `beers` and `meal` values, and respond based on those values.

```
onPost =
  parseFromForm :>>=
  case _ of
    Left err ->
      writeStatus statusBadRequest
      :*> closeHeaders
      :*> respond (err <> "\n")
    Right (Order { beers, meal })
      | meal == Omnivore || meal == Carnivore ->
        writeStatus statusBadRequest
        :*> closeHeaders
        :*> respond "Sorry, we do not serve meat here.\n"
      | otherwise ->
        writeStatus statusBadRequest
        :*> closeHeaders
        :*> respond ("One " <> show meal <> " meal and "
                     <> show beers <> " beers coming up!\n")
```

Let's try this server out at the command line.

```
$ curl -X POST -d 'beers=6' http://localhost:3000
Missing field: meal
$ curl -X POST -d 'meal=Vegan&beers=foo' http://localhost:3000
Invalid number: foo
$ curl -X POST -d 'meal=Omnivore&beers=6' http://localhost:3000
Sorry, we do not serve meat here.
$ curl -X POST -d 'meal=Vegetarian&beers=6' http://localhost:3000
One Vegetarian meal and 6 beers coming up!
```

3.3 NodeJS

The server in `Hyper.Node.Server` wraps the `http` module in NodeJS, and serves middleware using the `Aff` monad. Here is how you can start a Node server:

```
let
  app =
    writeStatus (Tuple 200 "OK")
    :*> closeHeaders
    :*> respond "Hello there!"
in runServer defaultOptions {} app
```

As seen above, `runServer` takes a record of options, an initial *components* record, and your application middleware. If you want to do logging on server startup, and on any request handling errors, use `defaultOptionsWithLogging`.

3.3.1 Monad Transformers

You might want to use a monad transformer stack in your application, for instance as a way to pass configuration, or to accumulate some state in the chain of middleware. The underlying monad of `Middleware` is parameterized for this exact purpose. When running the NodeJS server with monad transformers, you need to use `runServer'` instead of the regular `runServer`, and pass a function that runs your monad and returns an `Aff` value.

The following code runs a middleware using the `ReaderT` monad transformer. Note that the `runAppM` function might need to be defined at the top-level to please the type checker.

```
type MyConfig = { thingToSay :: String }

runAppM
  :: forall a.
    ReaderT MyConfig Aff a
  -> Aff a
runAppM = flip runReaderT { thingToSay: "Hello, ReaderT!" }

main :: Effect Unit
main =
  let app =
        lift' ask :>>= \config ->
          writeStatus statusOK
          :*> closeHeaders
          :*> respond config.thingToSay
  in runServer' defaultOptionsWithLogging {} runAppM app
```

In a real-world application the configuration type `MyConfig` could hold a database connection pool, or settings read from the environment, for example.

3.4 Testing

When running tests you might not want to start a full HTTP server and send requests using an HTTP client. Instead you can use the server in `Hyper.Test.TestServer`. It runs your middleware directly on `Conn` values, and collects the response using a Writer monad. You get back a `TestResponse` from which you can extract the status code, headers, and the response body.

```
it "responds with a friendly message" do
  conn <- { request: TestRequest defaultRequest
          , response: TestResponse Nothing [] []
          , components: {}
          }
```

(continues on next page)

(continued from previous page)

```
        # evalMiddleware app
        # testServer
testStatus conn `shouldEqual` Just statusOK
testStringBody conn `shouldEqual` "Hello there!"
```

Chapter 4

Tutorials

This chapter includes step-by-step guides on how to build complete web applications using Hyper.

4.1 Getting Started with Hyper

Welcome to *Getting Started with Hyper*! The purpose of this tutorial is for you to get a minimal Hyper project up and running. It assumes only that you have working PureScript development environment, with `node`, `psc`, `pulp`, and `bower` installed. If you do not have those tools already, follow the installation steps described in Getting Started[4] in the PureScript documentation.

Note: Hyper 0.9.0 requires PureScript version 0.12.2[5] or higher.

Start off by generating an empty project by entering the following commands in your terminal:

```
mkdir hello-hyper
cd hello-hyper
pulp init
```

Then install `purescript-hyper`, and add it as a project dependency, by running:

```
bower i purescript-hyper --save
```

You now have what you need to write a server. Edit `src/Main.purs` to look like this:

```
module Main where

import Prelude
import Control.Monad.Indexed ((:*>))
import Effect (Effect)
import Hyper.Node.Server (defaultOptionsWithLogging, runServer)
import Hyper.Response (closeHeaders, respond, writeStatus)
import Hyper.Status (statusOK)

main :: Effect Unit
main =
  let app = writeStatus statusOK
            :*> closeHeaders
            :*> respond "Hello, Hyper!"
  in runServer defaultOptionsWithLogging {} app
```

[4] https://github.com/purescript/documentation/blob/master/guides/Getting-Started.md
[5] https://github.com/purescript/purescript/releases/tag/v0.12.2

The *main* function defines a value *app*, which is a *Middleware* (page 5), responding with the HTTP status "200 OK", no extra headers, and "Hello, Hyper!" as the response body. The use of *runServer* creates a NodeJS server running our middleware.

Now build and run the program:

```
pulp run
```

You should see output similar to the following:

```
* Building project in /tmp/hello-hyper
* Build successful.
Listening on http://localhost:3000
```

Open http://localhost:3000 in your web browser, and you should see "Hello, Hyper!" rendered. Congratulations, you have written your first web server using Hyper!

Note: There is only one tutorial here right now, but more will be written. In the meantime, check out the runnable examples at GitHub[6].

---

[6] https://github.com/owickstrom/hyper/tree/master/examples

Chapter 5

Extensions

There are a number of extensions built on top of the Core API, providing higher-level abstractions. They are, for technical reasons, usually hosted as separate Git repositories.

5.1 Type-Level Routing

The Trout[7] package provides *type-level routing*. Its API, inspired heavily by the Haskell library Servant[8], lets us express web application routing at the type-level using *routing types*.

By using routing types we get static guarantees about having handled all cases and having correctly serializing and deserializing data. We also get a lot of stuff for free, such as type-safe parameters for handlers, generated type-safe URIs to endpoints, and generated clients and servers.

5.1.1 Packages

The following packages are available for type-level routing with Hyper:

Trout[9]
: Provides the core types used in routing types. It does not depend on Hyper, and can be used for other libraries and frameworks, theoretically.

Hypertrout[10]
: Used to create routers based on routing types, which are Hyper middleware. It can be seen as the equivalent of *servant-server*.

Trout Client[11]
: Derive client-side accessor functions for doing AJAX requests, based on Trout routing types. Use this together with *Hypertrout* to get an all-PureScript project, with safe routing between client and server.

[7] https://github.com/owickstrom/purescript-trout
[8] https://haskell-servant.github.io
[9] https://github.com/owickstrom/purescript-trout
[10] https://github.com/owickstrom/purescript-hypertrout
[11] https://github.com/owickstrom/purescript-trout-client

Chapter 6

Frequently Asked Questions

This document lists some of the questions and considerations about Hyper that come up frequently, and answers that hopefully serves future readers well. Please, do not regard this as a complete rationale of the project, but rather a casual summary of discussions that have been held in the past.

6.1 Why PureScript, and not Haskell?

This project started out as a curiosity around expressing statically typed middleware using extensible records[12]. While Haskell has a record construct, extensible records is not a built-in feature, and the libraries implementing extensible records for Haskell seemed too clunky to build a library upon. PureScript's extensible records, and row typing, was a very good match for middleware typing, and the cognitive overhead and expressiveness was reasonable.

Another concern, which might not be shared by all readers, is the NodeJS deployment target. While the project author would love to see more support for Haskell deployments in PaaS solutions, the current situation seem to favour deployment on NodeJS, JVM, Ruby, and Python. Also, many companies and developers might have invested in NodeJS infrastructure and libraries of their own, and so PureScript provides a gradual migration path to statically typed functional programming in web development.

The third point to consider is that PureScript is gaining traction on the client side, competing, and perhaps living in symbiosis, with frontend frameworks like React, Angular, and Ember. Having the option to share data types between client and server, and write them both in a language like PureScript, is something Hyper emphasizes.

[12] https://www.microsoft.com/en-us/research/publication/extensible-records-with-scoped-labels/