Writing a Simple REST Web Service in PureScript - Part 2

October 1, 2017
A twenty minute read
Tags: purescript, REST, programming, nilenso

To recap, in the first part of this two-part tutorial, we built a simple JSON REST web service in PureScript to create, update, get, list and delete users, backed by a Postgres database. In this part we'll work on the rest of the features. The requirements are:

1. validation of API requests.
2. reading the server and database configs from environment variables.
3. logging HTTP requests and debugging info.

Contents

1. Bugs!
2. Validation
3. Configuration
4. Logging
5. Conclusion

But first,

Bugs!

What happens if we hit a URL on our server which does not exist? Let's fire up the server and test it:

```
$ pulp --watch run
$ http GET http://localhost:4000/v1/random
HTTP/1.1 404 Not Found
Connection: keep-alive
Content-Length: 148
Content-Security-Policy: default-src 'self'
Content-Type: text/html; charset=utf-8
Date: Sat, 30 Sep 2017 08:23:20 GMT
X-Content-Type-Options: nosniff
X-Powered-By: Express

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot GET /v1/random</pre>
</body>
</html>
```

We get back a default HTML response with a 404 status from Express. Since we are writing a JSON API, we should return a JSON response in this case too. We add the following code in the `src/SimpleService/Server.purs` file to add a catch-all route and send a 404 status with a JSON error message:

```
-- previous code
import Data.Either (fromRight)
import Data.String.Regex (Regex, regex) as Re
import Data.String.Regex.Flags (noFlags) as Re
import Node.Express.App (App, all, delete, get, http, listenHttp, post,
useExternal)
import Node.Express.Response (sendJson, setStatus)
import Partial.Unsafe (unsafePartial)
-- previous code

allRoutePattern :: Re.Regex
allRoutePattern = unsafePartial $ fromRight $ Re.regex "/.*" Re.noFlags

app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
app pool = do
  useExternal jsonBodyParser

  get "/v1/user/:id"    $ getUser pool
  delete "/v1/user/:id" $ deleteUser pool
  post "/v1/users"      $ createUser pool
  patch "/v1/user/:id"  $ updateUser pool
  get "/v1/users"       $ listUsers pool

  all allRoutePattern do
    setStatus 404
    sendJson {error: "Route not found"}
  where
    patch = http (CustomMethod "patch")
```

`allRoutePattern` matches all routes because it uses a `"/.*"` regular expression. We place it as the last route to match all the otherwise unrouted requests. Let's see what is the result:

```
$ http GET http://localhost:4000/v1/random
HTTP/1.1 404 Not Found
Connection: keep-alive
Content-Length: 27
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 08:46:46 GMT
ETag: W/"1b-772e0u4nrE48ogbR0KmKfSvrHUE"
X-Powered-By: Express

{
    "error": "Route not found"
}
```

Now we get a nicely formatted JSON response.

Another scenario is when our application throws some uncaught error. To simulate this, we shut down our postgres database and hit the server for listing users:

```
$ http GET http://localhost:4000/v1/users
HTTP/1.1 500 Internal Server Error
Connection: keep-alive
Content-Length: 372
Content-Security-Policy: default-src 'self'
Content-Type: text/html; charset=utf-8
Date: Sat, 30 Sep 2017 08:53:40 GMT
X-Content-Type-Options: nosniff
X-Powered-By: Express

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Error: connect ECONNREFUSED 127.0.0.1:5432<br> &nbsp; &nbsp;at
Object._errnoException (util.js:1026:11)<br> &nbsp; &nbsp;at
_exceptionWithHostPort (util.js:1049:20)<br> &nbsp; &nbsp;at
TCPConnectWrap.afterConnect [as oncomplete] (net.js:1174:14)</pre>
</body>
</html>
```

We get another default HTML response from Express with a 500 status. Again, in this case we'd like to return a JSON response. We add the following code to the `src/SimpleService/Server.purs` file:

```
-- previous code
import Control.Monad.Eff.Exception (message)
import Node.Express.App (App, all, delete, get, http, listenHttp, post,
useExternal, useOnError)
-- previous code

app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
app pool = do
  -- previous code
  useOnError \err -> do
    setStatus 500
    sendJson {error: message err}
  where
    patch = http (CustomMethod "patch")
```

We add the `useOnError` handler which comes with `purescript-express` to return the error message as a JSON response. Back on the command-line:

```
$ http GET http://localhost:4000/v1/users
HTTP/1.1 500 Internal Server Error
Connection: keep-alive
Content-Length: 47
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 09:01:37 GMT
ETag: W/"2f-cJuIW6961YCpo9TWDSZ9VWHLGHE"
X-Powered-By: Express

{
    "error": "connect ECONNREFUSED 127.0.0.1:5432"
}
```

It works! Bugs are fixed now. We proceed to add next features.

Validation

Let's recall the code to update a user from the `src/SimpleService/Handler.purs` file:

```
updateUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
updateUser pool = getRouteParam "id" >>= case _ of
  Nothing -> respond 422 { error: "User ID is required" }
  Just sUserId -> case fromString sUserId of
    Nothing -> respond 422 { error: "User ID must be positive: " <> sUserId }
    Just userId -> getBody >>= case _ of
      Left errs -> respond 422 { error: intercalate ", " $ map renderForeignError
errs}
      Right (UserPatch userPatch) -> case unNullOrUndefined userPatch.name of
        Nothing -> respondNoContent 204
        Just userName -> if userName == ""
          then respond 422 { error: "User name must not be empty" }
          else do
            savedUser <- liftAff $ PG.withConnection pool \conn ->
PG.withTransaction conn do
              P.findUser conn userId >>= case _ of
                Nothing -> pure Nothing
                Just (User user) -> do
                  let user' = User (user { name = userName })
                  P.updateUser conn user'
                  pure $ Just user'
            case savedUser of
              Nothing -> respond 404 { error: "User not found with id: " <>
sUserId }
              Just user -> respond 200 (encode user)
```

As we can see, the actual request handling logic is obfuscated by the request validation logic for the user id and the user name patch parameters. We also notice that we are using three constructs for validation here: `Maybe`, `Either` and `if-then-else`. However, we can use just `Either` to subsume all these cases as it can "carry" a failure as well as a success case. `Either` also comes with a nice monad transformer `ExceptT` which provides the `do` syntax for failure propagation. So we choose `ExceptT` as the base construct for our validation framework and write functions to upgrade `Maybe` and `if-then-else` to it. We add the following code to the `src/SimpleService/Validation.purs` file:

```
module SimpleService.Validation
  (module MoreExports, module SimpleService.Validation) where

import Prelude

import Control.Monad.Except (ExceptT, except, runExceptT)
import Data.Either (Either(..))
import Data.Maybe (Maybe(..))
import Node.Express.Handler (HandlerM, Handler)
import Node.Express.Response (sendJson, setStatus)
import Node.Express.Types (EXPRESS)
import Control.Monad.Except (except) as MoreExports

type Validation eff a = ExceptT String (HandlerM (express :: EXPRESS | eff)) a

exceptMaybe :: forall e m a. Applicative m => e -> Maybe a -> ExceptT e m a
exceptMaybe e a = except $ case a of
  Just x  -> Right x
  Nothing -> Left e

exceptCond :: forall e m a. Applicative m => e -> (a -> Boolean) -> a -> ExceptT
e m a
exceptCond e cond a = except $ if cond a then Right a else Left e

withValidation :: forall eff a. Validation eff a -> (a -> Handler eff) -> Handler
eff
withValidation action handler = runExceptT action >>= case _ of
  Left err -> do
    setStatus 422
    sendJson {error: err}
  Right x  -> handler x
```

We re-export `except` from the `Control.Monad.Except` module. We also add a `withValidation` function which runs an `ExceptT` based validation and either returns an error response with a 422 status in case of a failed validation or runs the given action with the valid value in case of a successful validation.

Using these functions, we now write `updateUser` in the `src/SimpleService/Handler.purs` file as:

```
-- previous code
import Control.Monad.Trans.Class (lift)
import Data.Bifunctor (lmap)
import Data.Foreign (ForeignError, renderForeignError)
import Data.List.NonEmpty (toList)
import Data.List.Types (NonEmptyList)
import Data.Tuple (Tuple(..))
import SimpleService.Validation as V
-- previous code

renderForeignErrors :: forall a. Either (NonEmptyList ForeignError) a -> Either
String a
renderForeignErrors = lmap (toList >>> map renderForeignError >>> intercalate ",
")

updateUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
updateUser pool = V.withValidation (Tuple <$> getUserId <*> getUserPatch)
                                   \(Tuple userId (UserPatch userPatch)) ->
    case unNullOrUndefined userPatch.name of
      Nothing -> respondNoContent 204
      Just uName -> V.withValidation (getUserName uName) \userName -> do
        savedUser <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction
conn do
          P.findUser conn userId >>= case _ of
            Nothing -> pure Nothing
            Just (User user) -> do
              let user' = User (user { name = userName })
              P.updateUser conn user'
              pure $ Just user'
        case savedUser of
          Nothing -> respond 404 { error: "User not found with id: " <> show
userId }
          Just user -> respond 200 (encode user)
  where
    getUserId = lift (getRouteParam "id")
      >>= V.exceptMaybe "User ID is required"
      >>= fromString >>> V.exceptMaybe "User ID must be positive"

    getUserPatch = lift getBody >>= V.except <<< renderForeignErrors

    getUserName = V.exceptCond "User name must not be empty" (_ == "")
```

The validation logic has been extracted out in separate functions now which are composed using Applicative. The validation steps are composed using the `ExceptT` monad. We are now free to express the core logic of the function clearly. We rewrite the `src/SimpleService/Handler.purs` file using the validations:

```
module SimpleService.Handler where

import Prelude

import Control.Monad.Aff.Class (liftAff)
import Control.Monad.Trans.Class (lift)
import Data.Bifunctor (lmap)
import Data.Either (Either)
import Data.Foldable (intercalate)
import Data.Foreign (ForeignError, renderForeignError)
import Data.Foreign.Class (encode)
import Data.Foreign.NullOrUndefined (unNullOrUndefined)
import Data.Int (fromString)
import Data.List.NonEmpty (toList)
import Data.List.Types (NonEmptyList)
import Data.Maybe (Maybe(..))
import Data.Tuple (Tuple(..))
import Database.PostgreSQL as PG
import Node.Express.Handler (Handler)
import Node.Express.Request (getBody, getRouteParam)
import Node.Express.Response (end, sendJson, setStatus)
import SimpleService.Persistence as P
import SimpleService.Validation as V
import SimpleService.Types

getUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
getUser pool = V.withValidation getUserId \userId ->
  liftAff (PG.withConnection pool $ flip P.findUser userId) >>= case _ of
    Nothing -> respond 404 { error: "User not found with id: " <> show userId }
    Just user -> respond 200 (encode user)

deleteUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
deleteUser pool = V.withValidation getUserId \userId -> do
  found <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction conn do
    P.findUser conn userId >>= case _ of
      Nothing -> pure false
      Just _  -> do
        P.deleteUser conn userId
        pure true
  if found
    then respondNoContent 204
    else respond 404 { error: "User not found with id: " <> show userId }

createUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
createUser pool = V.withValidation getUser \user@(User _) -> do
  liftAff (PG.withConnection pool $ flip P.insertUser user)
  respondNoContent 201
  where
    getUser = lift getBody
      >>= V.except <<< renderForeignErrors
      >>= V.exceptCond "User ID must be positive" (\(User user) -> user.id > 0)
      >>= V.exceptCond "User name must not be empty" (\(User user) -> user.name
/= "")

updateUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
updateUser pool = V.withValidation (Tuple <$> getUserId <*> getUserPatch)
                                   \(Tuple userId (UserPatch userPatch)) ->
    case unNullOrUndefined userPatch.name of
      Nothing -> respondNoContent 204
      Just uName -> V.withValidation (getUserName uName) \userName -> do
        savedUser <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction
conn do
          P.findUser conn userId >>= case _ of
            Nothing -> pure Nothing
            Just (User user) -> do
              let user' = User (user { name = userName })
              P.updateUser conn user'
              pure $ Just user'
        case savedUser of
          Nothing -> respond 404 { error: "User not found with id: " <> show
userId }
          Just user -> respond 200 (encode user)
  where
    getUserPatch = lift getBody >>= V.except <<< renderForeignErrors
    getUserName = V.exceptCond "User name must not be empty" (_ /= "")

listUsers :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
listUsers pool = liftAff (PG.withConnection pool P.listUsers) >>= encode >>>
respond 200

getUserId :: forall eff. V.Validation eff Int
getUserId = lift (getRouteParam "id")
  >>= V.exceptMaybe "User ID is required"
  >>= fromString >>> V.exceptMaybe "User ID must be an integer"
  >>= V.exceptCond "User ID must be positive" (_ > 0)

renderForeignErrors :: forall a. Either (NonEmptyList ForeignError) a -> Either
String a
renderForeignErrors = lmap (toList >>> map renderForeignError >>> intercalate ",
")

respond :: forall eff a. Int -> a -> Handler eff
respond status body = do
  setStatus status
  sendJson body

respondNoContent :: forall eff. Int -> Handler eff
respondNoContent status = do
  setStatus status
  end
```

The code is much cleaner now. Let's try out a few test cases:

```
$ http POST http://localhost:4000/v1/users id:=3 name=roger
HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 0
Date: Sat, 30 Sep 2017 12:13:37 GMT
X-Powered-By: Express


$ http POST http://localhost:4000/v1/users id:=3
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 102
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 12:13:50 GMT
ETag: W/"66-/c4cfoquQZGwtDBUzHjJydJAHJ0"
X-Powered-By: Express

{
    "error": "Error at array index 0: (ErrorAtProperty \"name\" (TypeMismatch
\"String\" \"Undefined\"))"
}


$ http POST http://localhost:4000/v1/users id:=3 name=""
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 39
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 12:14:02 GMT
ETag: W/"27-JQsh12xu/rEFdWy8REF4NMtBUB4"
X-Powered-By: Express

{
    "error": "User name must not be empty"
}


$ http POST http://localhost:4000/v1/users id:=0 name=roger
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 36
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 12:14:14 GMT
ETag: W/"24-Pvt1L4eGilBmVtaOGHlSReJ413E"
X-Powered-By: Express

{
    "error": "User ID must be positive"
}
```

```
$ http GET http://localhost:4000/v1/user/3
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 23
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 12:14:28 GMT
ETag: W/"17-1scpiB1FT9DBu9s4I1gNWSjH2go"
X-Powered-By: Express

{
    "id": 3,
    "name": "roger"
}


$ http GET http://localhost:4000/v1/user/asdf
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 38
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 12:14:40 GMT
ETag: W/"26-//tvORl1gGDUMwgSaqbEpJhuadI"
X-Powered-By: Express

{
    "error": "User ID must be an integer"
}


$ http GET http://localhost:4000/v1/user/-1
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 36
Content-Type: application/json; charset=utf-8
Date: Sat, 30 Sep 2017 12:14:45 GMT
ETag: W/"24-Pvt1L4eGilBmVtaOGHlSReJ413E"
X-Powered-By: Express

{
    "error": "User ID must be positive"
}
```

It works as expected.

Configuration

Right now our application configuration resides in the `main` function:

```
main = runServer port databaseConfig
  where
    port = 4000
    databaseConfig = { user: "abhinav"
                     , password: ""
                     , host: "localhost"
                     , port: 5432
                     , database: "simple_service"
                     , max: 10
                     , idleTimeoutMillis: 1000
                     }
```

We are going to extract it out of the code and read it from the environment variables using the `purescript-config` package. First, we install the required packages using bower.

```
$ bower install --save purescript-node-process purescript-config
```

Now, we write the following code in the `src/SimpleService/Config.purs` file:

```
module SimpleService.Config where

import Data.Config
import Prelude

import Control.Monad.Eff (Eff)
import Data.Config.Node (fromEnv)
import Data.Either (Either)
import Data.Set (Set)
import Database.PostgreSQL as PG
import Node.Process (PROCESS)

type ServerConfig =
  { port           :: Int
  , databaseConfig :: PG.PoolConfiguration
  }

databaseConfig :: Config {name :: String} PG.PoolConfiguration
databaseConfig =
  { user: _, password: _, host: _, port: _, database: _, max: _,
idleTimeoutMillis: _ }
  <$> string {name: "user"}
  <*> string {name: "password"}
  <*> string {name: "host"}
  <*> int    {name: "port"}
  <*> string {name: "database"}
  <*> int    {name: "pool_size"}
  <*> int    {name: "idle_conn_timeout_millis"}

portConfig :: Config {name :: String} Int
portConfig = int {name: "port"}

serverConfig :: Config {name :: String} ServerConfig
serverConfig =
  { port: _, databaseConfig: _}
  <$> portConfig
  <*> prefix {name: "db"} databaseConfig

readServerConfig :: forall eff.
                    Eff (process :: PROCESS | eff) (Either (Set String)
ServerConfig)
readServerConfig = fromEnv "SS" serverConfig
```

We use the applicative DSL provided in `Data.Config` module to build a description of our configuration. This description contains the keys and types of the configuration, for consumption by various interpreters. Then we use the `fromEnv` interpreter to read the config from the environment variables derived from the `name` fields in the records in the description in the `readServerConfig` function. We also write a bash script to set those environment variables in the development environment in the `setenv.sh` file:

```
export SS_PORT=4000
export SS_DB_USER="abhinav"
export SS_DB_PASSWORD=""
export SS_DB_HOST="localhost"
export SS_DB_PORT=5432
export SS_DB_DATABASE="simple_service"
export SS_DB_POOL_SIZE=10
export SS_DB_IDLE_CONN_TIMEOUT_MILLIS=1000
```

Now we rewrite our `src/Main.purs` file to use the `readServerConfig` function:

```
module Main where

import Prelude

import Control.Monad.Eff (Eff)
import Control.Monad.Eff.Console (CONSOLE, log)
import Data.Either (Either(..))
import Data.Set (toUnfoldable)
import Data.String (joinWith)
import Database.PostgreSQL as PG
import Node.Express.Types (EXPRESS)
import Node.Process (PROCESS)
import Node.Process as Process
import SimpleService.Config (readServerConfig)
import SimpleService.Server (runServer)

main :: forall eff. Eff ( console :: CONSOLE
                        , express :: EXPRESS
                        , postgreSQL :: PG.POSTGRESQL
                        , process :: PROCESS
                        | eff ) Unit
main = readServerConfig >>= case _ of
  Left missingKeys -> do
    log $ "Unable to start. Missing Env keys: " <> joinWith ", " (toUnfoldable
missingKeys)
    Process.exit 1
  Right { port, databaseConfig } -> runServer port databaseConfig
```

If `readServerConfig` fails, we print the missing keys to the console and exit the process. Else we run the server with the read config.

To test this, we stop the server we ran in the beginning, source the config, and run it again:

```
$ pulp --watch run
* Building project in /Users/abhinav/ps-simple-rest-service
* Build successful.
Server listening on :4000
^C
$ source setenv.sh
$ pulp --watch run
* Building project in /Users/abhinav/ps-simple-rest-service
* Build successful.
Server listening on :4000
```

It works! We test the failure case by opening another terminal which does not have the environment variables set:

```
$ pulp run
* Building project in /Users/abhinav/ps-simple-rest-service
* Build successful.
Unable to start. Missing Env keys: SS_DB_DATABASE, SS_DB_HOST,
SS_DB_IDLE_CONN_TIMEOUT_MILLIS, SS_DB_PASSWORD, SS_DB_POOL_SIZE, SS_DB_PORT,
SS_DB_USER, SS_PORT
* ERROR: Subcommand terminated with exit code 1
```

Up next, we add logging to our application.

Logging

For logging, we use the `purescript-logging` package. We write a logger which logs to `stdout`; in the `src/SimpleService/Logger.purs` file:

```
module SimpleService.Logger
  ( debug
  , info
  , warn
  , error
  ) where

import Prelude

import Control.Logger as L
import Control.Monad.Eff.Class (class MonadEff, liftEff)
import Control.Monad.Eff.Console as C
import Control.Monad.Eff.Now (NOW, now)
import Data.DateTime.Instant (toDateTime)
import Data.Either (fromRight)
import Data.Formatter.DateTime (Formatter, format, parseFormatString)
import Data.Generic.Rep (class Generic)
import Data.Generic.Rep.Show (genericShow)
import Data.String (toUpper)
import Partial.Unsafe (unsafePartial)

data Level = Debug | Info | Warn | Error

derive instance eqLevel :: Eq Level
derive instance ordLevel :: Ord Level
derive instance genericLevel :: Generic Level _

instance showLevel :: Show Level where
  show = toUpper <<< genericShow

type Entry =
  { level   :: Level
  , message :: String
  }

dtFormatter :: Formatter
dtFormatter = unsafePartial $ fromRight $ parseFormatString "YYYY-MM-DD
HH:mm:ss.SSS"

logger :: forall m e. (
          MonadEff (console :: C.CONSOLE, now :: NOW | e) m) => L.Logger m Entry
logger = L.Logger $ \{ level, message } -> liftEff do
  time <- toDateTime <$> now
  C.log $ "[" <> format dtFormatter time <> "] " <> show level <> " " <> message

log :: forall m e.
        MonadEff (console :: C.CONSOLE , now :: NOW | e) m
     => Entry -> m Unit
log entry@{level} = L.log (L.cfilter (\e -> e.level == level) logger) entry
```

```
debug :: forall m e.
         MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
debug message = log { level: Debug, message }

info :: forall m e.
        MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
info message = log { level: Info, message }

warn :: forall m e.
        MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
warn message = log { level: Warn, message }

error :: forall m e.
         MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
error message = log { level: Error, message }
```

`purescript-logging` lets us define our own logging levels and loggers. We define four log levels, and a log entry type with the log level and the message. Then we write the logger which will print the log entry to `stdout` along with the current time as a well formatted string. We define convenience functions for each log level.

Before we proceed, let's install the required dependencies.

```
$ bower install --save purescript-logging purescript-now purescript-formatters
```

Now we add a request logger middleware to our server in the `src/SimpleService/Server.purs` file:

```
-- previous code
import Control.Monad.Eff.Console (CONSOLE)
import Control.Monad.Eff.Now (NOW)
import Data.Maybe (maybe)
import Data.String (toUpper)
import Node.Express.App (App, all, delete, get, http, listenHttp, post, use,
useExternal, useOnError)
import Node.Express.Handler (Handler, next)
import Node.Express.Request (getMethod, getPath)
import SimpleService.Logger as Log
-- previous code

requestLogger :: forall eff. Handler (console :: CONSOLE, now :: NOW | eff)
requestLogger = do
  method <- getMethod
  path   <- getPath
  Log.debug $ "HTTP: " <> maybe "" id ((toUpper <<< show) <$> method) <> " " <>
path
  next

app :: forall eff.
       PG.Pool
    -> App (postgreSQL :: PG.POSTGRESQL, console :: CONSOLE, now :: NOW | eff)
app pool = do
  useExternal jsonBodyParser
  use requestLogger
  -- previous code
```

We also convert all our previous logging statements which used `Console.log` to use `SimpleService.Logger` and add logs in our handlers. We can see logging in effect by restarting the server and hitting it:

```
$ pulp --watch run
* Building project in /Users/abhinav/ps-simple-rest-service
* Build successful.
[2017-09-30 16:02:41.634] INFO Server listening on :4000
[2017-09-30 16:02:43.494] DEBUG HTTP: PATCH /v1/user/3
[2017-09-30 16:02:43.517] DEBUG Updated user: 3
[2017-09-30 16:03:46.615] DEBUG HTTP: DELETE /v1/user/3
[2017-09-30 16:03:46.635] DEBUG Deleted user 3
[2017-09-30 16:05:03.805] DEBUG HTTP: GET /v1/users
```

Conclusion

In this tutorial we learned how to create a simple JSON REST web service written in PureScript with persistence, validation, configuration and logging. The complete code for this tutorial can be found in github. Discuss this post in the comments.

© 2017-2018, Abhinav Sarkar