

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:

```
1 $ pulp --watch run
```

```
1  $ http GET http://localhost:4000/v1/random
2  HTTP/1.1 404 Not Found
3  Connection: keep-alive
4  Content-Length: 148
5  Content-Security-Policy: default-src 'self'
6  Content-Type: text/html; charset=utf-8
7  Date: Sat, 30 Sep 2017 08:23:20 GMT
8  X-Content-Type-Options: nosniff
9  X-Powered-By: Express
10
11 <!DOCTYPE html>
12 <html lang="en">
13 <head>
14 <meta charset="utf-8">
15 <title>Error</title>
16 </head>
17 <body>
18 <pre>Cannot GET /v1/random</pre>
19 </body>
20 </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:

```

1  -- previous code
2  import Data.Either (fromRight)
3  import Data.String.Regex (Regex, regex) as Re
4  import Data.String.Regex.Flags (noFlags) as Re
5  import Node.Express.App (App, all, delete, get, http, listenHttp, post, useExternal)
6  import Node.Express.Response (sendJson, setStatus)
7  import Partial.Unsafe (unsafePartial)
8  -- previous code
9
10 allRoutePattern :: Re.Regex
11 allRoutePattern = unsafePartial $ fromRight $ Re.regex "/.*" Re.noFlags
12
13 app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
14 app pool = do
15     useExternal jsonBodyParser
16
17     get "/v1/user/:id"    $ getUser pool
18     delete "/v1/user/:id" $ deleteUser pool
19     post "/v1/users"      $ createUser pool
20     patch "/v1/user/:id"  $ updateUser pool
21     get "/v1/users"       $ listUsers pool
22
23     all allRoutePattern do
24         setStatus 404
25         sendJson {error: "Route not found"}
26     where
27         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:

```

1  $ http GET http://localhost:4000/v1/random
2  HTTP/1.1 404 Not Found
3  Connection: keep-alive
4  Content-Length: 27
5  Content-Type: application/json; charset=utf-8
6  Date: Sat, 30 Sep 2017 08:46:46 GMT
7  ETag: W/"1b-772e0u4nrE48ogbR0KmKfSvrHUE"
8  X-Powered-By: Express
9
10 {
11     "error": "Route not found"
12 }

```

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:

```

1  $ http GET http://localhost:4000/v1/users
2  HTTP/1.1 500 Internal Server Error
3  Connection: keep-alive
4  Content-Length: 372
5  Content-Security-Policy: default-src 'self'
6  Content-Type: text/html; charset=utf-8
7  Date: Sat, 30 Sep 2017 08:53:40 GMT
8  X-Content-Type-Options: nosniff
9  X-Powered-By: Express
10
11 <!DOCTYPE html>
12 <html lang="en">
13 <head>
14 <meta charset="utf-8">
15 <title>Error</title>
16 </head>
17 <body>
18 <pre>Error: connect ECONNREFUSED 127.0.0.1:5432<br> &nbsp; &nbsp; &nbsp;at
   Object._errnoException (util.js:1026:11)<br> &nbsp; &nbsp; &nbsp;at _exceptionWithHostPort
   (util.js:1049:20)<br> &nbsp; &nbsp; &nbsp;at TCPConnectWrap.afterConnect [as oncomplete]
   (net.js:1174:14)</pre>
19 </body>
20 </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:

```

1  -- previous code
2  import Control.Monad.Eff.Exception (message)
3  import Node.Express.App (App, all, delete, get, http, listenHttp, post, useExternal,
   useOnError)
4  -- previous code
5
6  app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
7  app pool = do
8    -- previous code
9    useOnError \err -> do
10      setStatus 500
11      sendJson {error: message err}
12  where
13    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:

```

1  $ http GET http://localhost:4000/v1/users
2  HTTP/1.1 500 Internal Server Error
3  Connection: keep-alive
4  Content-Length: 47
5  Content-Type: application/json; charset=utf-8
6  Date: Sat, 30 Sep 2017 09:01:37 GMT
7  ETag: W/"2f-cJuIW6961YCpo9TWDSZ9VWHLGHE"
8  X-Powered-By: Express
9
10 {
11   "error": "connect ECONNREFUSED 127.0.0.1:5432"
12 }

```

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:

```

1  updateUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
2  updateUser pool = getRouteParam "id" >=> case _ of
3    Nothing -> respond 422 { error: "User ID is required" }
4    Just sUserId -> case fromString sUserId of
5      Nothing -> respond 422 { error: "User ID must be positive: " <> sUserId }
6      Just userId -> getBody >=> case _ of
7        Left errs -> respond 422 { error: intercalate ", " $ map renderForeignError errs}
8        Right (UserPatch userPatch) -> case unNullOrUndefined userPatch.name of
9          Nothing -> respondNoContent 204
10         Just userName -> if userName == ""
11           then respond 422 { error: "User name must not be empty" }
12           else do
13             savedUser <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction
conn do
14               P.findUser conn userId >=> case _ of
15                 Nothing -> pure Nothing
16                 Just (User user) -> do
17                   let user' = User (user { name = userName })
18                   P.updateUser conn user'
19                   pure $ Just user'
20             case savedUser of
21               Nothing -> respond 404 { error: "User not found with id: " <> sUserId }
22               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:

```
1  module SimpleService.Validation
2    (module MoreExports, module SimpleService.Validation) where
3
4  import Prelude
5
6  import Control.Monad.Except (ExceptT, except, runExceptT)
7  import Data.Either (Either(..))
8  import Data.Maybe (Maybe(..))
9  import Node.Express.Handler (HandlerM, Handler)
10 import Node.Express.Response (sendJson, setStatus)
11 import Node.Express.Types (EXPRESS)
12 import Control.Monad.Except (except) as MoreExports
13
14 type Validation eff a = ExceptT String (HandlerM (express :: EXPRESS | eff)) a
15
16 exceptMaybe :: forall e m a. Applicative m => e -> Maybe a -> ExceptT e m a
17 exceptMaybe e a = except $ case a of
18   Just x  -> Right x
19   Nothing -> Left e
20
21 exceptCond :: forall e m a. Applicative m => e -> (a -> Boolean) -> a -> ExceptT e m a
22 exceptCond e cond a = except $ if cond a then Right a else Left e
23
24 withValidation :: forall eff a. Validation eff a -> (a -> Handler eff) -> Handler eff
25 withValidation action handler = runExceptT action >= case _ of
26   Left err -> do
27     setStatus 422
28     sendJson {error: err}
29   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:

```

1  -- previous code
2  import Control.Monad.Trans.Class (lift)
3  import Data.Bifunctor (lmap)
4  import Data.Foreign (ForeignError, renderForeignError)
5  import Data.List.NonEmpty (toList)
6  import Data.List.Types (NonEmptyList)
7  import Data.Tuple (Tuple(..))
8  import SimpleService.Validation as V
9  -- previous code
10
11 renderForeignErrors :: forall a. Either (NonEmptyList ForeignError) a -> Either String a
12 renderForeignErrors = lmap (toList >>> map renderForeignError >>> intercalate ", ")
13
14 updateUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
15 updateUser pool = V.withValidation (Tuple <$> getUserId <*> getUserPatch)
16                               \(Tuple userId (UserPatch userPatch)) ->
17   case unNullOrUndefined userPatch.name of
18     Nothing -> respondNoContent 204
19     Just uName -> V.withValidation (getUserName uName) \userName -> do
20       savedUser <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction conn
21   do
22     P.findUser conn userId >>= case _ of
23       Nothing -> pure Nothing
24       Just (User user) -> do
25         let user' = User (user { name = userName })
26         P.updateUser conn user'
27         pure $ Just user'
28     case savedUser of
29       Nothing -> respond 404 { error: "User not found with id: " <> show userId }
30       Just user -> respond 200 (encode user)
31 where
32   getUserId = lift (getRouteParam "id")
33             >>= V.exceptMaybe "User ID is required"
34             >>= fromString >>> V.exceptMaybe "User ID must be positive"
35
36   getUserPatch = lift getBody >>= V.except <<< renderForeignErrors
37
38   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:

```

1  module SimpleService.Handler where
2
3  import Prelude
4
5  import Control.Monad.Aff.Class (liftAff)
6  import Control.Monad.Trans.Class (lift)
7  import Data.Bifunctor (lmap)
8  import Data.Either (Either)
9  import Data.Foldable (intercalate)
10 import Data.Foreign (ForeignError, renderForeignError)
11 import Data.Foreign.Class (encode)
12 import Data.Foreign.NullOrUndefined (unNullOrUndefined)
13 import Data.Int (fromString)
14 import Data.List.NonEmpty (toList)
15 import Data.List.Types (NonEmptyList)
16 import Data.Maybe (Maybe(..))
17 import Data.Tuple (Tuple(..))
18 import Database.PostgreSQL as PG
19 import Node.Express.Handler (Handler)
20 import Node.Express.Request (getBody, getRouteParam)
21 import Node.Express.Response (end, sendJson, setStatus)
22 import SimpleService.Persistence as P
23 import SimpleService.Validation as V
24 import SimpleService.Types
25
26 getUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
27 getUser pool = V.withValidation getUserId \userId ->
28   liftAff (PG.withConnection pool $ flip P.findUser userId) >=> case _ of
29     Nothing -> respond 404 { error: "User not found with id: " <> show userId }
30     Just user -> respond 200 (encode user)
31
32 deleteUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
33 deleteUser pool = V.withValidation getUserId \userId -> do
34   found <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction conn do
35     P.findUser conn userId >=> case _ of
36       Nothing -> pure false
37       Just _ -> do
38         P.deleteUser conn userId
39         pure true
40   if found
41     then respondNoContent 204
42     else respond 404 { error: "User not found with id: " <> show userId }
43
44 createUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
45 createUser pool = V.withValidation getUser \user@(User _) -> do
46   liftAff (PG.withConnection pool $ flip P.insertUser user)
47   respondNoContent 201
48   where
49     getUser = lift getBody
50     >=> V.except <<< renderForeignErrors

```



```

51     >>= V.exceptCond "User ID must be positive" (\(User user) -> user.id > 0)
52     >>= V.exceptCond "User name must not be empty" (\(User user) -> user.name /= "")
53
54 updateUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
55 updateUser pool = V.withValidation (Tuple <$> getUserId <*> getUserPatch)
56                     \(Tuple userId (UserPatch userPatch)) ->
57     case unNullOrUndefined userPatch.name of
58     Nothing -> respondNoContent 204
59     Just uName -> V.withValidation (getUserName uName) \(userName -> do
60         savedUser <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction conn
61         do
62             P.findUser conn userId >>= case _ of
63             Nothing -> pure Nothing
64             Just (User user) -> do
65                 let user' = User (user { name = userName })
66                 P.updateUser conn user'
67                 pure $ Just user'
68             case savedUser of
69             Nothing -> respond 404 { error: "User not found with id: " <> show userId }
70             Just user -> respond 200 (encode user)
71     where
72         getUserPatch = lift getBody >>= V.except <<< renderForeignErrors
73         getUserName = V.exceptCond "User name must not be empty" (_ /= "")
74
75 listUsers :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
76 listUsers pool = liftAff (PG.withConnection pool P.listUsers) >>= encode >>> respond 200
77
78 getUserId :: forall eff. V.Validation eff Int
79 getUserId = lift (getRouteParam "id")
80     >>= V.exceptMaybe "User ID is required"
81     >>= fromString >>> V.exceptMaybe "User ID must be an integer"
82     >>= V.exceptCond "User ID must be positive" (_ > 0)
83
84 renderForeignErrors :: forall a. Either (NonEmptyList ForeignError) a -> Either String a
85 renderForeignErrors = lmap (toList >>> map renderForeignError >>> intercalate ", ")
86
87 respond :: forall eff a. Int -> a -> Handler eff
88 respond status body = do
89     setStatus status
90     sendJson body
91
92 respondNoContent :: forall eff. Int -> Handler eff
93 respondNoContent status = do
94     setStatus status
95     end

```

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

```
1 $ http POST http://localhost:4000/v1/users id:=3 name=roger
2 HTTP/1.1 201 Created
3 Connection: keep-alive
4 Content-Length: 0
5 Date: Sat, 30 Sep 2017 12:13:37 GMT
6 X-Powered-By: Express
```

```
1 $ http POST http://localhost:4000/v1/users id:=3
2 HTTP/1.1 422 Unprocessable Entity
3 Connection: keep-alive
4 Content-Length: 102
5 Content-Type: application/json; charset=utf-8
6 Date: Sat, 30 Sep 2017 12:13:50 GMT
7 ETag: W/"66-/c4cfoquQZGwtDBUzHjJydJAHJ0"
8 X-Powered-By: Express
9
10 {
11   "error": "Error at array index 0: (ErrorAtProperty \"name\" (TypeMismatch \"String\"
    \"Undefined\"))"
12 }
```

```
1 $ http POST http://localhost:4000/v1/users id:=3 name=""
2 HTTP/1.1 422 Unprocessable Entity
3 Connection: keep-alive
4 Content-Length: 39
5 Content-Type: application/json; charset=utf-8
6 Date: Sat, 30 Sep 2017 12:14:02 GMT
7 ETag: W/"27-JQsh12xu/rEFdwy8REF4NMtBUB4"
8 X-Powered-By: Express
9
10 {
11   "error": "User name must not be empty"
12 }
```

```
1 $ http POST http://localhost:4000/v1/users id:=0 name=roger
2 HTTP/1.1 422 Unprocessable Entity
3 Connection: keep-alive
4 Content-Length: 36
5 Content-Type: application/json; charset=utf-8
6 Date: Sat, 30 Sep 2017 12:14:14 GMT
7 ETag: W/"24-Pvt1L4eGilBmVta0GHlSRJ413E"
8 X-Powered-By: Express
9
10 {
11   "error": "User ID must be positive"
12 }
```

```
1 $ http GET http://localhost:4000/v1/user/3
2 HTTP/1.1 200 OK
3 Connection: keep-alive
4 Content-Length: 23
5 Content-Type: application/json; charset=utf-8
6 Date: Sat, 30 Sep 2017 12:14:28 GMT
7 ETag: W/"17-1scpiB1FT9DBu9s4I1gNWSjH2go"
8 X-Powered-By: Express
9
10 {
11     "id": 3,
12     "name": "roger"
13 }
```

```
1 $ http GET http://localhost:4000/v1/user/asdf
2 HTTP/1.1 422 Unprocessable Entity
3 Connection: keep-alive
4 Content-Length: 38
5 Content-Type: application/json; charset=utf-8
6 Date: Sat, 30 Sep 2017 12:14:40 GMT
7 ETag: W/"26-//tvORl1gGDUMwgSaqbEpJhuadI"
8 X-Powered-By: Express
9
10 {
11     "error": "User ID must be an integer"
12 }
```

```
1 $ http GET http://localhost:4000/v1/user/-1
2 HTTP/1.1 422 Unprocessable Entity
3 Connection: keep-alive
4 Content-Length: 36
5 Content-Type: application/json; charset=utf-8
6 Date: Sat, 30 Sep 2017 12:14:45 GMT
7 ETag: W/"24-Pvt1L4eGilBmVta0GHlSReJ413E"
8 X-Powered-By: Express
9
10 {
11     "error": "User ID must be positive"
12 }
```

It works as expected.

Configuration

Right now our application configuration resides in the main function:

```
1 main = runServer port databaseConfig
2   where
3     port = 4000
4     databaseConfig = { user: "abhinav"
5                        , password: ""
6                        , host: "localhost"
7                        , port: 5432
8                        , database: "simple_service"
9                        , max: 10
10                       , idleTimeoutMillis: 1000
11                       }
```

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.

```
1 $ bower install --save purescript-node-process purescript-config
```

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

```

1  module SimpleService.Config where
2
3  import Data.Config
4  import Prelude
5
6  import Control.Monad.Eff (Eff)
7  import Data.Config.Node (fromEnv)
8  import Data.Either (Either)
9  import Data.Set (Set)
10 import Database.PostgreSQL as PG
11 import Node.Process (PROCESS)
12
13 type ServerConfig =
14   { port          :: Int
15   , databaseConfig :: PG.PoolConfiguration
16   }
17
18 databaseConfig :: Config {name :: String} PG.PoolConfiguration
19 databaseConfig =
20   { user: _, password: _, host: _, port: _, database: _, max: _, idleTimeoutMillis: _ }
21   <$> string {name: "user"}
22   <*> string {name: "password"}
23   <*> string {name: "host"}
24   <*> int    {name: "port"}
25   <*> string {name: "database"}
26   <*> int    {name: "pool_size"}
27   <*> int    {name: "idle_conn_timeout_millis"}
28
29 portConfig :: Config {name :: String} Int
30 portConfig = int {name: "port"}
31
32 serverConfig :: Config {name :: String} ServerConfig
33 serverConfig =
34   { port: _, databaseConfig: _ }
35   <$> portConfig
36   <*> prefix {name: "db"} databaseConfig
37
38 readServerConfig :: forall eff.
39   Eff (process :: PROCESS | eff) (Either (Set String) ServerConfig)
40 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:

```
1 export SS_PORT=4000
2 export SS_DB_USER="abhinav"
3 export SS_DB_PASSWORD=""
4 export SS_DB_HOST="localhost"
5 export SS_DB_PORT=5432
6 export SS_DB_DATABASE="simple_service"
7 export SS_DB_POOL_SIZE=10
8 export SS_DB_IDLE_CONN_TIMEOUT_MILLIS=1000
```

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

```
1 module Main where
2
3 import Prelude
4
5 import Control.Monad.Eff (Eff)
6 import Control.Monad.Eff.Console (CONSOLE, log)
7 import Data.Either (Either(..))
8 import Data.Set (toUnfoldable)
9 import Data.String (joinWith)
10 import Database.PostgreSQL as PG
11 import Node.Express.Types (EXPRESS)
12 import Node.Process (PROCESS)
13 import Node.Process as Process
14 import SimpleService.Config (readServerConfig)
15 import SimpleService.Server (runServer)
16
17 main :: forall eff. Eff ( console :: CONSOLE
18                          , express :: EXPRESS
19                          , postgresql :: PG.POSTGRESQL
20                          , process :: PROCESS
21                          | eff ) Unit
22 main = readServerConfig >=> case _ of
23   Left missingKeys -> do
24     log $ "Unable to start. Missing Env keys: " <> joinWith ", " (toUnfoldable
25       missingKeys)
26     Process.exit 1
27   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:

```
1 $ pulp --watch run
2 * Building project in /Users/abhinav/ps-simple-rest-service
3 * Build successful.
4 Server listening on :4000
5 ^C
6 $ source setenv.sh
7 $ pulp --watch run
8 * Building project in /Users/abhinav/ps-simple-rest-service
9 * Build successful.
10 Server listening on :4000
```

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

```
1 $ pulp run
2 * Building project in /Users/abhinav/ps-simple-rest-service
3 * Build successful.
4 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
5 * 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:

```

1  module SimpleService.Logger
2      ( debug
3      , info
4      , warn
5      , error
6      ) where
7
8  import Prelude
9
10 import Control.Logger as L
11 import Control.Monad.Eff.Class (class MonadEff, liftEff)
12 import Control.Monad.Eff.Console as C
13 import Control.Monad.Eff.Now (NOW, now)
14 import Data.DateTime.Instant (toDateTime)
15 import Data.Either (fromRight)
16 import Data.Formatter.DateTime (Formatter, format, parseFormatString)
17 import Data.Generic.Rep (class Generic)
18 import Data.Generic.Rep.Show (genericShow)
19 import Data.String (toUpper)
20 import Partial.Unsafe (unsafePartial)
21
22 data Level = Debug | Info | Warn | Error
23
24 derive instance eqLevel :: Eq Level
25 derive instance ordLevel :: Ord Level
26 derive instance genericLevel :: Generic Level _
27
28 instance showLevel :: Show Level where
29     show = toUpper <<< genericShow
30
31 type Entry =
32     { level    :: Level
33     , message :: String
34     }
35
36 dtFormatter :: Formatter
37 dtFormatter = unsafePartial $ fromRight $ parseFormatString "YYYY-MM-DD HH:mm:ss.SSS"
38
39 logger :: forall m e. (
40     MonadEff (console :: C.CONSOLE, now :: NOW | e) m) => L.Logger m Entry
41 logger = L.Logger $ \{ level, message } -> liftEff do
42     time <- toDateTime <$> now
43     C.log $ "[" <> format dtFormatter time <> "]" " <> show level <> " " <> message
44
45 log :: forall m e.
46     MonadEff (console :: C.CONSOLE , now :: NOW | e) m
47     => Entry -> m Unit
48 log entry@{level} = L.log (L.cfilter (\e -> e.level == level) logger) entry
49
50 debug :: forall m e.

```



```

51         MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
52 debug message = log { level: Debug, message }
53
54 info :: forall m e.
55         MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
56 info message = log { level: Info, message }
57
58 warn :: forall m e.
59         MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
60 warn message = log { level: Warn, message }
61
62 error :: forall m e.
63         MonadEff (console :: C.CONSOLE , now :: NOW | e) m => String -> m Unit
64 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.

```

1 $ 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:

```

1  -- previous code
2  import Control.Monad.Eff.Console (CONSOLE)
3  import Control.Monad.Eff.Now (NOW)
4  import Data.Maybe (maybe)
5  import Data.String (toUpper)
6  import Node.Express.App (App, all, delete, get, http, listenHttp, post, use,
    useExternal, useOnError)
7  import Node.Express.Handler (Handler, next)
8  import Node.Express.Request (getMethod, getPath)
9  import SimpleService.Logger as Log
10 -- previous code
11
12 requestLogger :: forall eff. Handler (console :: CONSOLE, now :: NOW | eff)
13 requestLogger = do
14   method <- getMethod
15   path    <- getPath
16   Log.debug $ "HTTP: " <> maybe "" id ((toUpper <<< show) <$> method) <> " " <> path
17   next
18
19 app :: forall eff.
20   PG.Pool
21   -> App (postgreSQL :: PG.POSTGRESQL, console :: CONSOLE, now :: NOW | eff)
22 app pool = do
23   useExternal jsonBodyParser
24   use requestLogger
25   -- 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:

```

1 $ pulp --watch run
2 * Building project in /Users/abhinav/ps-simple-rest-service
3 * Build successful.
4 [2017-09-30 16:02:41.634] INFO Server listening on :4000
5 [2017-09-30 16:02:43.494] DEBUG HTTP: PATCH /v1/user/3
6 [2017-09-30 16:02:43.517] DEBUG Updated user: 3
7 [2017-09-30 16:03:46.615] DEBUG HTTP: DELETE /v1/user/3
8 [2017-09-30 16:03:46.635] DEBUG Deleted user 3
9 [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.