Writing a Simple REST Web Service in PureScript - Part 1

September 29, 2017
A twenty-three minute read
Tags: purescript, REST, programming, nilenso

At Nilenso, we've been working with a client who has chosen PureScript as their primary programming language. Since I couldn't find any canonical documentation on writing a web service in PureScript, I thought I'd jot down the approach that we took.

The aim of this two-part tutorial is to create a simple JSON REST web service written in PureScript, to run on a node.js server. This assumes that you have basic proficiency with PureScript. We have the following requirements:

1. persisting users into a Postgres database.
2. API endpoints for creating, updating, getting, listing and deleting users.
3. validation of API requests.
4. reading the server and database configs from environment variables.
5. logging HTTP requests and debugging info.

In this part we'll work on setting up the project and on the first two requirements. In the next part we'll work on the rest of the requirements.

Contents

Setting Up

We start with installing PureScript and the required tools. This assumes that we have node and npm installed on our machine.

```
$ mkdir -p ~/.local/
$ npm install -g purescript pulp bower --prefix ~/.local/
```

Pulp is a build tool for PureScript projects and bower is a package manager used to get PureScript libraries. We'll have to add `~/.local/bin` in our `$PATH` (if it is not already added) to access the binaries installed.

Let's create a directory for our project and make Pulp initialize it:

```
$ mkdir ps-simple-rest-service
$ cd ps-simple-rest-service
$ pulp init
$ ls
bower.json  bower_components  src  test
$ cat bower.json
{
  "name": "ps-simple-rest-service",
  "ignore": [
    "**/.*",
    "node_modules",
    "bower_components",
    "output"
  ],
  "dependencies": {
    "purescript-prelude": "^3.1.0",
    "purescript-console": "^3.0.0"
  },
  "devDependencies": {
    "purescript-psci-support": "^3.0.0"
  }
}
$ ls bower_components
purescript-console  purescript-eff  purescript-prelude purescript-psci-support
```

Pulp creates the basic project structure for us. `src` directory will contain the source while the `test` directory will contain the tests. `bower.json` contains the PureScript libraries as dependencies which are downloaded and installed in the `bower_components` directory.

Types First

First, we create the types needed in `src/SimpleService/Types.purs`:

```
module SimpleService.Types where

import Prelude

import Data.Foreign.Class (class Decode, class Encode)
import Data.Foreign.Generic (defaultOptions, genericDecode, genericEncode)
import Data.Generic.Rep (class Generic)
import Data.Generic.Rep.Show (genericShow)

type UserID = Int

newtype User = User
  { id   :: UserID
  , name :: String
  }

derive instance genericUser :: Generic User _

instance showUser :: Show User where
  show = genericShow

instance decodeUser :: Decode User where
  decode = genericDecode $ defaultOptions { unwrapSingleConstructors = true }

instance encodeUser :: Encode User where
  encode = genericEncode $ defaultOptions { unwrapSingleConstructors = true }
```

We are using the generic support for PureScript types from the `purescript-generics-rep` and `purescript-foreign-generic` libraries to encode and decode the `User` type to JSON. We install the library by running the following command:

```
$ bower install purescript-foreign-generic --save
```

Now we can load up the module in the PureScript REPL and try out the JSON conversion features:

```
$ pulp repl
> import SimpleService.Types
> user = User { id: 1, name: "Abhinav"}
> user
(User { id: 1, name: "Abhinav" })

> import Data.Foreign.Generic
> userJSON = encodeJSON user
> userJSON
"{\"name\":\"Abhinav\",\"id\":1}"

> import Data.Foreign
> import Control.Monad.Except.Trans
> import Data.Identity
> dUser = decodeJSON userJSON :: F User
> eUser = let (Identity eUser) = runExceptT $ dUser in eUser
> eUser
(Right (User { id: 1, name: "Abhinav" }))
```

We use `encodeJSON` and `decodeJSON` functions from the `Data.Foreign.Generic` module to encode and decode the `User` instance to JSON. The return type of `decodeJSON` is a bit complicated as it needs to return the parsing errors too. In this case, the decoding returns no errors and we get back a `Right` with the correctly parsed `User` instance.

Persisting It

Next, we add the support for saving a `User` instance to a Postgres database. First, we install the required libraries using bower and npm: `pg` for Javascript bindings to call Postgres, `purescript-aff` for asynchronous processing and `purescript-postgresql-client` for PureScript wrapper over `pg`:

```
$ npm init -y
$ npm install pg@6.4.0 --save
$ bower install purescript-aff --save
$ bower install purescript-postgresql-client --save
```

Before writing the code, we create the database and the `users` table using the command-line Postgres client:

```
$ psql postgres
psql (9.5.4)
Type "help" for help.

postgres=# create database simple_service;
CREATE DATABASE
postgres=# \c simple_service
You are now connected to database "simple_service" as user "abhinav".
simple_service=# create table users (id int primary key, name varchar(100) not null);
CREATE TABLE
simple_service=# \d users
            Table "public.users"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 id     | integer                | not null
 name   | character varying(100) | not null
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
```

Now we add support for converting a `User` instance to-and-from an SQL row by adding the following code in the `src/SimpleService/Types.purs` file:

```
import Data.Array as Array
import Data.Either (Either(..))
import Database.PostgreSQL (class FromSQLRow, class ToSQLRow, fromSQLValue, toSQLValue)

-- code written earlier

instance userFromSQLRow :: FromSQLRow User where
  fromSQLRow [id, name] =
    User <$> ({ id: _, name: _} <$> fromSQLValue id <*> fromSQLValue name)

  fromSQLRow xs = Left $ "Row has " <> show n <> " fields, expecting 2."
    where n = Array.length xs

instance userToSQLRow :: ToSQLRow User where
  toSQLRow (User {id, name}) = [toSQLValue id, toSQLValue name]
```

We can try out the persistence support in the REPL:

```
$ pulp repl
PSCi, version 0.11.6
Type :? for help

import Prelude
>
> import SimpleService.Types
> import Control.Monad.Aff (launchAff, liftEff')
> import Database.PostgreSQL as PG
> user = User { id: 1, name: "Abhinav" }
> databaseConfig = {user: "abhinav", password: "", host: "localhost", port: 5432, database: "simple_service", max: 10, idleTimeoutMillis: 1000}

> :paste
… void $ launchAff do
…   pool <- PG.newPool databaseConfig
…   PG.withConnection pool $ \conn -> do
…     PG.execute conn (PG.Query "insert into users (id, name) values ($1, $2)") user
…
unit

> import Data.Foldable (for_)
> import Control.Monad.Eff.Console (logShow)
> :paste
… void $ launchAff do
…   pool <- PG.newPool databaseConfig
…   PG.withConnection pool $ \conn -> do
…     users :: Array User <- PG.query conn (PG.Query "select id, name from users where id = $1") (PG.Row1 1)
…     liftEff' $ void $ for_ users logShow
…
unit
(User { id: 1, name: "Abhinav" })
```

We create the `databaseConfig` record with the configs needed to connect to the database. Using the record, we create a new Postgres connection pool (`PG.newPool`) and get a connection from it (`PG.withConnection`). We call `PG.execute` with the connection, the SQL insert query for the users table and the `User` instance, to insert the user into the table. All of this is done inside `launchAff` which takes care of sequencing the callbacks correctly to make the asynchronous code look synchronous.

Similarly, in the second part, we query the table using `PG.query` function by calling it with a connection, the SQL select query and the `User` ID as the query parameter. It returns an `Array` of users which we log to the console using the `logShow` function.

We use this experiment to write the persistence related code in the `src/SimpleService/Persistence.purs` file:

```purescript
module SimpleService.Persistence
  ( insertUser
  , findUser
  , updateUser
  , deleteUser
  , listUsers
  ) where

import Prelude

import Control.Monad.Aff (Aff)
import Data.Array as Array
import Data.Maybe (Maybe)
import Database.PostgreSQL as PG
import SimpleService.Types (User(..), UserID)

insertUserQuery :: String
insertUserQuery = "insert into users (id, name) values ($1, $2)"

findUserQuery :: String
findUserQuery = "select id, name from users where id = $1"

updateUserQuery :: String
updateUserQuery = "update users set name = $1 where id = $2"

deleteUserQuery :: String
deleteUserQuery = "delete from users where id = $1"

listUsersQuery :: String
listUsersQuery = "select id, name from users"

insertUser :: forall eff. PG.Connection -> User
           -> Aff (postgreSQL :: PG.POSTGRESQL | eff) Unit
insertUser conn user =
  PG.execute conn (PG.Query insertUserQuery) user

findUser :: forall eff. PG.Connection -> UserID
         -> Aff (postgreSQL :: PG.POSTGRESQL | eff) (Maybe User)
findUser conn userID =
  map Array.head $ PG.query conn (PG.Query findUserQuery) (PG.Row1 userID)

updateUser :: forall eff. PG.Connection -> User
           -> Aff (postgreSQL :: PG.POSTGRESQL | eff) Unit
updateUser conn (User {id, name}) =
  PG.execute conn (PG.Query updateUserQuery) (PG.Row2 name id)

deleteUser :: forall eff. PG.Connection -> UserID
           -> Aff (postgreSQL :: PG.POSTGRESQL | eff) Unit
deleteUser conn userID =
  PG.execute conn (PG.Query deleteUserQuery) (PG.Row1 userID)
```

```
51
52 listUsers :: forall eff. PG.Connection
53           -> Aff (postgreSQL :: PG.POSTGRESQL | eff) (Array User)
54 listUsers conn =
55   PG.query conn (PG.Query listUsersQuery) PG.Row0
```

Serving It

We can now write a simple HTTP API over the persistence layer using Express to provide CRUD functionality for users. Let's install Express and purescript-express, the PureScript wrapper over it:

```
1 $ npm install express --save
2 $ bower install purescript-express --save
```

Getting a User

We do this top-down. First, we change `src/Main.purs` to run the HTTP server by providing the server port and database configuration:

```
1  module Main where
2
3  import Prelude
4
5  import Control.Monad.Eff (Eff)
6  import Control.Monad.Eff.Console (CONSOLE)
7  import Database.PostgreSQL as PG
8  import Node.Express.Types (EXPRESS)
9  import SimpleService.Server (runServer)
10
11 main :: forall eff. Eff ( console :: CONSOLE
12                         , express :: EXPRESS
13                         , postgreSQL :: PG.POSTGRESQL
14                         | eff) Unit
15 main = runServer port databaseConfig
16   where
17     port = 4000
18     databaseConfig = { user: "abhinav"
19                      , password: ""
20                      , host: "localhost"
21                      , port: 5432
22                      , database: "simple_service"
23                      , max: 10
24                      , idleTimeoutMillis: 1000
25                      }
```

Next, we wire up the server routes to the handlers in `src/SimpleService/Server.purs`:

```
module SimpleService.Server (runServer) where

import Prelude

import Control.Monad.Aff (runAff)
import Control.Monad.Eff (Eff)
import Control.Monad.Eff.Class (liftEff)
import Control.Monad.Eff.Console (CONSOLE, log, logShow)
import Database.PostgreSQL as PG
import Node.Express.App (App, get, listenHttp)
import Node.Express.Types (EXPRESS)
import SimpleService.Handler (getUser)

app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
app pool = do
  get "/v1/user/:id" $ getUser pool

runServer :: forall eff.
             Int
          -> PG.PoolConfiguration
          -> Eff ( express :: EXPRESS
                 , postgreSQL :: PG.POSTGRESQL
                 , console :: CONSOLE
                 | eff ) Unit
runServer port databaseConfig =  void $ runAff logShow pure do
  pool <- PG.newPool databaseConfig
  let app' = app pool
  void $ liftEff $ listenHttp app' port \_ -> log $ "Server listening on :" <> show port
```

`runServer` creates a PostgreSQL connection pool and passes it to the `app` function which creates the Express application, which in turn, binds it to the handler `getUser`. Then it launches the HTTP server by calling `listenHttp`.

Finally, we write the actual `getUser` handler in `src/SimpleService/Handler.purs`:

```
module SimpleService.Handler where

import Prelude

import Control.Monad.Aff.Class (liftAff)
import Data.Foreign.Class (encode)
import Data.Int (fromString)
import Data.Maybe (Maybe(..))
import Database.PostgreSQL as PG
import Node.Express.Handler (Handler)
import Node.Express.Request (getRouteParam)
import Node.Express.Response (end, sendJson, setStatus)
import SimpleService.Persistence as P

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

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
```

`getUser` validates the route parameter for valid user ID, sending error HTTP responses in case of failures. It then calls `findUser` to find the user and returns appropriate response.

We can test this on the command-line using HTTPie. We run `pulp --watch run` in one terminal to start the server with file watching, and test it from another terminal:

```
$ pulp --watch run
* Building project in ps-simple-rest-service
* Build successful.
Server listening on :4000
```

```
$ http GET http://localhost:4000/v1/user/1 # should return the user we created earlier
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 25
Content-Type: application/json; charset=utf-8
Date: Sun, 10 Sep 2017 14:32:52 GMT
ETag: W/"19-qmtK9XY+WDrqHTgqtFlV+h+NGOY"
X-Powered-By: Express

{
    "id": 1,
    "name": "Abhinav"
}
```

```
$ http GET http://localhost:4000/v1/user/s
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 38
Content-Type: application/json; charset=utf-8
Date: Sun, 10 Sep 2017 14:36:04 GMT
ETag: W/"26-//tvORl1gGDUMwgSaqbEpJhuadI"
X-Powered-By: Express

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

```
$ http GET http://localhost:4000/v1/user/2
HTTP/1.1 404 Not Found
Connection: keep-alive
Content-Length: 36
Content-Type: application/json; charset=utf-8
Date: Sun, 10 Sep 2017 14:36:11 GMT
ETag: W/"24-IyD5VT4E8/m3kvpwycRBQunI7Uc"
X-Powered-By: Express

{
    "error": "User not found with id: 2"
}
```

Deleting a User

`deleteUser` handler is similar. We add the route in the `app` function in the `src/SimpleService/Server.purs` file:

```
-- previous code
import Node.Express.App (App, delete, get, listenHttp)
import SimpleService.Handler (deleteUser, getUser)
-- previous code

app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
app pool = do
  get "/v1/user/:id" $ getUser pool
  delete "/v1/user/:id" $ deleteUser pool

-- previous code
```

And we add the handler in the `src/SimpleService/Handler.purs` file:

```
deleteUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
deleteUser 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 an integer: " <> sUserId }
    Just 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: " <> sUserId }
```

After the usual validations on the route param, `deleteUser` tries to find the user by the given user ID and if found, it deletes the user. Both the persistence related functions are run inside a single SQL transaction using `PG.withTransaction` function. `deleteUser` return 404 status if the user is not found, else it returns 204 status.

Let’s try it out:

```
$ http GET http://localhost:4000/v1/user/1
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 25
Content-Type: application/json; charset=utf-8
Date: Mon, 11 Sep 2017 05:10:50 GMT
ETag: W/"19-GC9FAtbd81t7CtrQgsNuc8HITXU"
X-Powered-By: Express

{
    "id": 1,
    "name": "Abhinav"
}
```

```
$ http DELETE http://localhost:4000/v1/user/1
HTTP/1.1 204 No Content
Connection: keep-alive
Date: Mon, 11 Sep 2017 05:10:56 GMT
X-Powered-By: Express
```

```
$ http GET http://localhost:4000/v1/user/1
HTTP/1.1 404 Not Found
Connection: keep-alive
Content-Length: 37
Content-Type: application/json; charset=utf-8
Date: Mon, 11 Sep 2017 05:11:03 GMT
ETag: W/"25-Eoc4ZbEF73CyW8EGh6t2jqI8mLU"
X-Powered-By: Express

{
    "error": "User not found with id: 1"
}
```

```
$ http DELETE http://localhost:4000/v1/user/1
HTTP/1.1 404 Not Found
Connection: keep-alive
Content-Length: 37
Content-Type: application/json; charset=utf-8
Date: Mon, 11 Sep 2017 05:11:05 GMT
ETag: W/"25-Eoc4ZbEF73CyW8EGh6t2jqI8mLU"
X-Powered-By: Express

{
    "error": "User not found with id: 1"
}
```

Creating a User

`createUser` handler is a bit more involved. First, we add an Express middleware to parse the body of the request as JSON. We use `body-parser` for this and access it through PureScript FFI. We create a new file `src/SimpleService/Middleware/BodyParser.js` with the content:

```
"use strict";

var bodyParser = require("body-parser");

exports.jsonBodyParser = bodyParser.json({
  limit: "5mb"
});
```

And write a wrapper for it in the file `src/SimpleService/Middleware/BodyParser.purs` with the content:

```
module SimpleService.Middleware.BodyParser where

import Prelude
import Data.Function.Uncurried (Fn3)
import Node.Express.Types (ExpressM, Response, Request)

foreign import jsonBodyParser ::
  forall e. Fn3 Request Response (ExpressM e Unit) (ExpressM e Unit)
```

We also install the `body-parser` npm dependency:

```
$ npm install --save body-parser
```

Next, we change the `app` function in the `src/SimpleService/Server.purs` file to add the middleware and the route:

```
 -- previous code
 import Node.Express.App (App, delete, get, listenHttp, post, useExternal)
 import SimpleService.Handler (createUser, deleteUser, getUser)
 import SimpleService.Middleware.BodyParser (jsonBodyParser)
 -- previous code

 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
```

And finally, we write the handler in the `src/SimpleService/Handler.purs` file:

```
-- previous code
import Data.Either (Either(..))
import Data.Foldable (intercalate)
import Data.Foreign (renderForeignError)
import Node.Express.Request (getBody, getRouteParam)
import SimpleService.Types
-- previous code

createUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
createUser pool = getBody >>= case _ of
  Left errs -> respond 422 { error: intercalate ", " $ map renderForeignError errs}
  Right u@(User user) ->
    if user.id <= 0
      then respond 422 { error: "User ID must be positive: " <> show user.id}
      else if user.name == ""
        then respond 422 { error: "User name must not be empty" }
        else do
          liftAff (PG.withConnection pool $ flip P.insertUser u)
          respondNoContent 201
```

`createUser` calls `getBody` which has type signature `forall e a. (Decode a) => HandlerM (express :: EXPRESS | e) (Either MultipleErrors a)`. It returns either a list of parsing errors or a parsed instance, which in our case is a `User`. In case of errors, we just return the errors rendered as string with a 422 status. If we get a parsed `User` instance, we do some validations on it, returning appropriate error messages. If all validations pass, we create the user in the database by calling `insertUser` from the persistence layer and respond with a status 201.

We can try it out:

```
$ http POST http://localhost:4000/v1/users name="abhinav"
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 97
Content-Type: application/json; charset=utf-8
Date: Mon, 11 Sep 2017 05:51:28 GMT
ETag: W/"61-BgsrMukZpImcdwAJEKCZ+70WBb8"
X-Powered-By: Express

{
    "error": "Error at array index 0: (ErrorAtProperty \"id\" (TypeMismatch \"Int\" \"Undefined\"))"
}
```

```
$ http POST http://localhost:4000/v1/users id:=1 name=""
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 39
Content-Type: application/json; charset=utf-8
Date: Mon, 11 Sep 2017 05:51:42 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:=1 name="abhinav"
HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 0
Date: Mon, 11 Sep 2017 05:52:23 GMT
X-Powered-By: Express
```

```
$ http GET http://localhost:4000/v1/user/1
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 25
Content-Type: application/json; charset=utf-8
Date: Mon, 11 Sep 2017 05:52:30 GMT
ETag: W/"19-GC9FAtbd81t7CtrQgsNuc8HITXU"
X-Powered-By: Express

{
    "id": 1,
    "name": "abhinav"
}
```

First try returns a parsing failure because we didn't provide the `id` field. Second try is a validation failure because the name was empty. Third try is a success which we confirm by doing a `GET` request next.

Updating a User

We want to allow a user's name to be updated through the API, but not the user's ID. So we add a new type to `src/SimpleService/Types.purs` to represent a possible change in user's name:

```
-- previous code
import Data.Foreign.NullOrUndefined (NullOrUndefined)
-- previous code

newtype UserPatch = UserPatch { name :: NullOrUndefined String }

derive instance genericUserPatch :: Generic UserPatch _

instance decodeUserPatch :: Decode UserPatch where
  decode = genericDecode $ defaultOptions { unwrapSingleConstructors = true }
```

`NullOrUndefined` is a wrapper over `Maybe` with added support for Javascript `null` and `undefined` values. We define `UserPatch` as having a possibly null (or undefined) `name` field.

Now we can add the corresponding handler in `src/SimpleService/Handlers.purs`:

```
-- previous code
import Data.Foreign.NullOrUndefined (unNullOrUndefined)
-- previous code

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)
```

After checking for a valid user ID as before, we get the decoded request body as a `UserPatch` instance. If the path does not have the `name` field or has it as `null`, there is nothing to do and we respond with a 204 status. If the user's name is present in the patch, we validate it for non-emptiness. Then, within a database transaction, we try to find the user with the given ID, responding with a 404 status if the user is not found. If the user is found, we update the user's name in the database, and respond with a 200 status and the saved user encoded as the JSON response body.

Finally, we can add the route to our server's router in `src/SimpleService/Server.purs` to make the functionality available:

```
1  -- previous code
2  import Node.Express.App (App, delete, get, http, listenHttp, post, useExternal)
3  import Node.Express.Types (EXPRESS, Method(..))
4  import SimpleService.Handler (createUser, deleteUser, getUser, updateUser)
5  -- previous code
6
7  app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
8  app pool = do
9    useExternal jsonBodyParser
10
11   get "/v1/user/:id"    $ getUser pool
12   delete "/v1/user/:id" $ deleteUser pool
13   post "/v1/users"      $ createUser pool
14   patch "/v1/user/:id"  $ updateUser pool
15   where
16     patch = http (CustomMethod "patch")
```

We can try it out now:

```
1  $ http GET http://localhost:4000/v1/user/1
2  HTTP/1.1 200 OK
3  Connection: keep-alive
4  Content-Length: 26
5  Content-Type: application/json; charset=utf-8
6  Date: Fri, 11 Sep 2017 06:41:10 GMT
7  ETag: W/"1a-hoLBx55zeY8nZFWJh/kM05pXwSA"
8  X-Powered-By: Express
9
10 {
11     "id": 1,
12     "name": "abhinav"
13 }
```

```
$ http PATCH http://localhost:4000/v1/user/1 name=abhinavsarkar
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 31
Content-Type: application/json; charset=utf-8
Date: Fri, 11 Sep 2017 06:41:36 GMT
ETag: W/"1f-EG5i0hq/hYhF0BsuheD9hNXeBpI"
X-Powered-By: Express

{
    "id": 1,
    "name": "abhinavsarkar"
}
```

```
$ http GET http://localhost:4000/v1/user/1
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 31
Content-Type: application/json; charset=utf-8
Date: Fri, 11 Sep 2017 06:41:40 GMT
ETag: W/"1f-EG5i0hq/hYhF0BsuheD9hNXeBpI"
X-Powered-By: Express

{
    "id": 1,
    "name": "abhinavsarkar"
}
```

```
$ http PATCH http://localhost:4000/v1/user/1
HTTP/1.1 204 No Content
Connection: keep-alive
Date: Fri, 11 Sep 2017 06:42:31 GMT
X-Powered-By: Express
```

```
$ http PATCH http://localhost:4000/v1/user/1 name=""
HTTP/1.1 422 Unprocessable Entity
Connection: keep-alive
Content-Length: 39
Content-Type: application/json; charset=utf-8
Date: Fri, 11 Sep 2017 06:43:17 GMT
ETag: W/"27-JQsh12xu/rEFdWy8REF4NMtBUB4"
X-Powered-By: Express

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

Listing all Users

Listing all users is quite simple since it doesn't require us to take any request parameter.

We add the handler to the `src/SimpleService/Handler.purs` file:

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

And the route to the `src/SimpleService/Server.purs` file:

```
-- previous code
import SimpleService.Handler (createUser, deleteUser, getUser, listUsers, updateUser)
-- previous code

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
  where
    patch = http (CustomMethod "patch")
```

And that's it. We can test this endpoint:

```
$ http POST http://localhost:4000/v1/users id:=2 name=sarkarabhinav
HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 0
Date: Fri, 11 Sep 2017 07:06:24 GMT
X-Powered-By: Express
```

```
$ http GET http://localhost:4000/v1/users
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 65
Content-Type: application/json; charset=utf-8
Date: Fri, 11 Sep 2017 07:06:27 GMT
ETag: W/"41-btt9uNdG+9A1RO7SCLOsyMmIyFo"
X-Powered-By: Express

[
    {
        "id": 1,
        "name": "abhinavsarkar"
    },
    {
        "id": 2,
        "name": "sarkarabhinav"
    }
]
```

Conclusion

That concludes the first part of the two-part tutorial. We learned how to set up a PureScript project, how to access a Postgres database and how to create a JSON REST API over the database. The code till the end of this part can be found in github. In the next part, we'll learn how to do API validation, application configuration and logging. Discuss this post in the comments.

