Writing a Simple REST Web Service in PureScript - Part 1

✎ September 29, 2017

🕒 A twenty-three minute read

🏷 Tags: purescript, REST, programming, nilenso

A t 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.

```
1  $ mkdir -p ~/.local/
2  $ 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`:

```purescript
1   module SimpleService.Types where
2
3   import Prelude
4
5   import Data.Foreign.Class (class Decode, class Encode)
6   import Data.Foreign.Generic (defaultOptions, genericDecode, genericEncode)
7   import Data.Generic.Rep (class Generic)
8   import Data.Generic.Rep.Show (genericShow)
9
10  type UserID = Int
11
12  newtype User = User
13    { id   :: UserID
14    , name :: String
15    }
16
17  derive instance genericUser :: Generic User _
18
19  instance showUser :: Show User where
20    show = genericShow
21
22  instance decodeUser :: Decode User where
23    decode = genericDecode $ defaultOptions { unwrapSingleConstructors = true }
24
25  instance encodeUser :: Encode User where
26    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:

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

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

```
1   $ pulp repl
2   > import SimpleService.Types
3   > user = User { id: 1, name: "Abhinav"}
4   > user
5   (User { id: 1, name: "Abhinav" })
6
7   > import Data.Foreign.Generic
8   > userJSON = encodeJSON user
9   > userJSON
10  "{\"name\":\"Abhinav\",\"id\":1}"
11
12  > import Data.Foreign
13  > import Control.Monad.Except.Trans
14  > import Data.Identity
15  > dUser = decodeJSON userJSON :: F User
16  > eUser = let (Identity eUser) = runExceptT $ dUser in eUser
17  > eUser
18  (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`:

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

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

```
1   $ psql postgres
2   psql (9.5.4)
3   Type "help" for help.
4
5   postgres=# create database simple_service;
6   CREATE DATABASE
7   postgres=# \c simple_service
8   You are now connected to database "simple_service" as user "abhinav".
9   simple_service=# create table users (id int primary key, name varchar(100) not null);
10  CREATE TABLE
11  simple_service=# \d users
12              Table "public.users"
13   Column |          Type          | Modifiers
14  --------+------------------------+-----------
15   id     | integer                | not null
16   name   | character varying(100) | not null
17  Indexes:
18      "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:

```
1   import Data.Array as Array
2   import Data.Either (Either(..))
3   import Database.PostgreSQL (class FromSQLRow, class ToSQLRow, fromSQLValue, toSQLValue)
4
5   -- code written earlier
6
7   instance userFromSQLRow :: FromSQLRow User where
8     fromSQLRow [id, name] =
9       User <$> ({ id: _, name: _} <$> fromSQLValue id <*> fromSQLValue name)
10
11    fromSQLRow xs = Left $ "Row has " <> show n <> " fields, expecting 2."
12      where n = Array.length xs
13
14  instance userToSQLRow :: ToSQLRow User where
15    toSQLRow (User {id, name}) = [toSQLValue id, toSQLValue name]
```

We can try out the persistence support in the REPL:

```
1   $ pulp repl
2   PSCi, version 0.11.6
3   Type :? for help
4
5   import Prelude
6   >
7   > import SimpleService.Types
8   > import Control.Monad.Aff (launchAff, liftEff')
9   > import Database.PostgreSQL as PG
10  > user = User { id: 1, name: "Abhinav" }
11  > databaseConfig = {user: "abhinav", password: "", host: "localhost", port: 5432,
    database: "simple_service", max: 10, idleTimeoutMillis: 1000}
12
13  > :paste
14  … void $ launchAff do
15  …    pool <- PG.newPool databaseConfig
16  …    PG.withConnection pool $ \conn -> do
17  …      PG.execute conn (PG.Query "insert into users (id, name) values ($1, $2)") user
18  …
19  unit
20
21  > import Data.Foldable (for_)
22  > import Control.Monad.Eff.Console (logShow)
23  > :paste
24  … void $ launchAff do
25  …    pool <- PG.newPool databaseConfig
26  …    PG.withConnection pool $ \conn -> do
27  …      users :: Array User <- PG.query conn (PG.Query "select id, name from users where
    id = $1") (PG.Row1 1)
28  …      liftEff' $ void $ for_ users logShow
29  …
30  unit
31  (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:

```
1   module SimpleService.Server (runServer) where
2
3   import Prelude
4
5   import Control.Monad.Aff (runAff)
6   import Control.Monad.Eff (Eff)
7   import Control.Monad.Eff.Class (liftEff)
8   import Control.Monad.Eff.Console (CONSOLE, log, logShow)
9   import Database.PostgreSQL as PG
10  import Node.Express.App (App, get, listenHttp)
11  import Node.Express.Types (EXPRESS)
12  import SimpleService.Handler (getUser)
13
14  app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
15  app pool = do
16    get "/v1/user/:id" $ getUser pool
17
18  runServer :: forall eff.
19                Int
20            -> PG.PoolConfiguration
21            -> Eff ( express :: EXPRESS
22                   , postgreSQL :: PG.POSTGRESQL
23                   , console :: CONSOLE
24                   | eff ) Unit
25  runServer port databaseConfig =  void $ runAff logShow pure do
26    pool <- PG.newPool databaseConfig
27    let app' = app pool
28    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:

```
1  module SimpleService.Handler where
2
3  import Prelude
4
5  import Control.Monad.Aff.Class (liftAff)
6  import Data.Foreign.Class (encode)
7  import Data.Int (fromString)
8  import Data.Maybe (Maybe(..))
9  import Database.PostgreSQL as PG
10 import Node.Express.Handler (Handler)
11 import Node.Express.Request (getRouteParam)
12 import Node.Express.Response (end, sendJson, setStatus)
13 import SimpleService.Persistence as P
14
15 getUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
16 getUser pool = getRouteParam "id" >>= case _ of
17   Nothing -> respond 422 { error: "User ID is required" }
18   Just sUserId -> case fromString sUserId of
19     Nothing -> respond 422 { error: "User ID must be an integer: " <> sUserId }
20     Just userId -> liftAff (PG.withConnection pool $ flip P.findUser userId) >>= case _
   of
21       Nothing -> respond 404 { error: "User not found with id: " <> sUserId }
22       Just user -> respond 200 (encode user)
23
24 respond :: forall eff a. Int -> a -> Handler eff
25 respond status body = do
26   setStatus status
27   sendJson body
28
29 respondNoContent :: forall eff. Int -> Handler eff
30 respondNoContent status = do
31   setStatus status
32   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:

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

```
1   $ http GET http://localhost:4000/v1/user/1 # should return the user we created earlier
2   HTTP/1.1 200 OK
3   Connection: keep-alive
4   Content-Length: 25
5   Content-Type: application/json; charset=utf-8
6   Date: Sun, 10 Sep 2017 14:32:52 GMT
7   ETag: W/"19-qmtK9XY+WDrqHTgqtFlV+h+NGOY"
8   X-Powered-By: Express
9
10  {
11      "id": 1,
12      "name": "Abhinav"
13  }
```

```
1   $ http GET http://localhost:4000/v1/user/s
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: Sun, 10 Sep 2017 14:36:04 GMT
7   ETag: W/"26-//tvORl1gGDUMwgSaqbEpJhuadI"
8   X-Powered-By: Express
9
10  {
11      "error": "User ID must be an integer: s"
12  }
```

```
1   $ http GET http://localhost:4000/v1/user/2
2   HTTP/1.1 404 Not Found
3   Connection: keep-alive
4   Content-Length: 36
5   Content-Type: application/json; charset=utf-8
6   Date: Sun, 10 Sep 2017 14:36:11 GMT
7   ETag: W/"24-IyD5VT4E8/m3kvpwycRBQunI7Uc"
8   X-Powered-By: Express
9
10  {
11      "error": "User not found with id: 2"
12  }
```

Deleting a User

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

```
1   -- previous code
2   import Node.Express.App (App, delete, get, listenHttp)
3   import SimpleService.Handler (deleteUser, getUser)
4   -- previous code
5
6   app :: forall eff. PG.Pool -> App (postgreSQL :: PG.POSTGRESQL | eff)
7   app pool = do
8     get "/v1/user/:id" $ getUser pool
9     delete "/v1/user/:id" $ deleteUser pool
10
11  -- previous code
```

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

```
1   deleteUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
2   deleteUser 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 an integer: " <> sUserId }
6       Just userId -> do
7         found <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction conn do
8           P.findUser conn userId >>= case _ of
9             Nothing -> pure false
10            Just _  -> do
11              P.deleteUser conn userId
12              pure true
13        if found
14          then respondNoContent 204
15          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:

```
1  $ http GET http://localhost:4000/v1/user/1
2  HTTP/1.1 200 OK
3  Connection: keep-alive
4  Content-Length: 25
5  Content-Type: application/json; charset=utf-8
6  Date: Mon, 11 Sep 2017 05:10:50 GMT
7  ETag: W/"19-GC9FAtbd81t7CtrQgsNuc8HITXU"
8  X-Powered-By: Express
9
10 {
11     "id": 1,
12     "name": "Abhinav"
13 }
```

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

```
1  $ http GET http://localhost:4000/v1/user/1
2  HTTP/1.1 404 Not Found
3  Connection: keep-alive
4  Content-Length: 37
5  Content-Type: application/json; charset=utf-8
6  Date: Mon, 11 Sep 2017 05:11:03 GMT
7  ETag: W/"25-Eoc4ZbEF73CyW8EGh6t2jqI8mLU"
8  X-Powered-By: Express
9
10 {
11     "error": "User not found with id: 1"
12 }
```

```
1  $ http DELETE http://localhost:4000/v1/user/1
2  HTTP/1.1 404 Not Found
3  Connection: keep-alive
4  Content-Length: 37
5  Content-Type: application/json; charset=utf-8
6  Date: Mon, 11 Sep 2017 05:11:05 GMT
7  ETag: W/"25-Eoc4ZbEF73CyW8EGh6t2jqI8mLU"
8  X-Powered-By: Express
9
10 {
11     "error": "User not found with id: 1"
12 }
```

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:

```
1  "use strict";
2
3  var bodyParser = require("body-parser");
4
5  exports.jsonBodyParser = bodyParser.json({
6    limit: "5mb"
7  });
```

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

```
1  module SimpleService.Middleware.BodyParser where
2
3  import Prelude
4  import Data.Function.Uncurried (Fn3)
5  import Node.Express.Types (ExpressM, Response, Request)
6
7  foreign import jsonBodyParser ::
8    forall e. Fn3 Request Response (ExpressM e Unit) (ExpressM e Unit)
```

We also install the body-parser npm dependency:

```
1  $ 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:

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

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

```
1   -- previous code
2   import Data.Either (Either(..))
3   import Data.Foldable (intercalate)
4   import Data.Foreign (renderForeignError)
5   import Node.Express.Request (getBody, getRouteParam)
6   import SimpleService.Types
7   -- previous code
8
9   createUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
10  createUser pool = getBody >>= case _ of
11    Left errs -> respond 422 { error: intercalate ", " $ map renderForeignError errs}
12    Right u@(User user) ->
13      if user.id <= 0
14        then respond 422 { error: "User ID must be positive: " <> show user.id}
15        else if user.name == ""
16          then respond 422 { error: "User name must not be empty" }
17          else do
18            liftAff (PG.withConnection pool $ flip P.insertUser u)
19            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:

```
1   $ http POST http://localhost:4000/v1/users name="abhinav"
2   HTTP/1.1 422 Unprocessable Entity
3   Connection: keep-alive
4   Content-Length: 97
5   Content-Type: application/json; charset=utf-8
6   Date: Mon, 11 Sep 2017 05:51:28 GMT
7   ETag: W/"61-BgsrMukZpImcdwAJEKCZ+70WBb8"
8   X-Powered-By: Express
9
10  {
11      "error": "Error at array index 0: (ErrorAtProperty \"id\" (TypeMismatch \"Int\" \"Undefined\"))"
12  }
```

```
1   $ http POST http://localhost:4000/v1/users id:=1 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: Mon, 11 Sep 2017 05:51:42 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:=1 name="abhinav"
2   HTTP/1.1 201 Created
3   Connection: keep-alive
4   Content-Length: 0
5   Date: Mon, 11 Sep 2017 05:52:23 GMT
6   X-Powered-By: Express
```

```
1   $ http GET http://localhost:4000/v1/user/1
2   HTTP/1.1 200 OK
3   Connection: keep-alive
4   Content-Length: 25
5   Content-Type: application/json; charset=utf-8
6   Date: Mon, 11 Sep 2017 05:52:30 GMT
7   ETag: W/"19-GC9FAtbd81t7CtrQgsNuc8HITXU"
8   X-Powered-By: Express
9
10  {
11      "id": 1,
12      "name": "abhinav"
13  }
```

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:

```
1   -- previous code
2   import Data.Foreign.NullOrUndefined (NullOrUndefined)
3   -- previous code
4
5   newtype UserPatch = UserPatch { name :: NullOrUndefined String }
6
7   derive instance genericUserPatch :: Generic UserPatch _
8
9   instance decodeUserPatch :: Decode UserPatch where
10    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:

```
1   -- previous code
2   import Data.Foreign.NullOrUndefined (unNullOrUndefined)
3   -- previous code
4
5   updateUser :: forall eff. PG.Pool -> Handler (postgreSQL :: PG.POSTGRESQL | eff)
6   updateUser pool = getRouteParam "id" >>= case _ of
7     Nothing -> respond 422 { error: "User ID is required" }
8     Just sUserId -> case fromString sUserId of
9       Nothing -> respond 422 { error: "User ID must be positive: " <> sUserId }
10      Just userId -> getBody >>= case _ of
11        Left errs -> respond 422 { error: intercalate ", " $ map renderForeignError errs}
12        Right (UserPatch userPatch) -> case unNullOrUndefined userPatch.name of
13          Nothing -> respondNoContent 204
14          Just userName -> if userName == ""
15            then respond 422 { error: "User name must not be empty" }
16            else do
17              savedUser <- liftAff $ PG.withConnection pool \conn -> PG.withTransaction
    conn do
18                P.findUser conn userId >>= case _ of
19                  Nothing -> pure Nothing
20                  Just (User user) -> do
21                    let user' = User (user { name = userName })
22                    P.updateUser conn user'
23                    pure $ Just user'
24              case savedUser of
25                Nothing -> respond 404 { error: "User not found with id: " <> sUserId }
26                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:

```purescript
-- previous code
import Node.Express.App (App, delete, get, http, listenHttp, post, useExternal)
import Node.Express.Types (EXPRESS, Method(..))
import SimpleService.Handler (createUser, deleteUser, getUser, 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
  where
    patch = http (CustomMethod "patch")
```

We can try it out now:

```
$ http GET http://localhost:4000/v1/user/1
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 26
Content-Type: application/json; charset=utf-8
Date: Fri, 11 Sep 2017 06:41:10 GMT
ETag: W/"1a-hoLBx55zeY8nZFWJh/kM05pXwSA"
X-Powered-By: Express

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

```
1   $ http PATCH http://localhost:4000/v1/user/1 name=abhinavsarkar
2   HTTP/1.1 200 OK
3   Connection: keep-alive
4   Content-Length: 31
5   Content-Type: application/json; charset=utf-8
6   Date: Fri, 11 Sep 2017 06:41:36 GMT
7   ETag: W/"1f-EG5i0hq/hYhF0BsuheD9hNXeBpI"
8   X-Powered-By: Express
9
10  {
11      "id": 1,
12      "name": "abhinavsarkar"
13  }
```

```
1   $ http GET http://localhost:4000/v1/user/1
2   HTTP/1.1 200 OK
3   Connection: keep-alive
4   Content-Length: 31
5   Content-Type: application/json; charset=utf-8
6   Date: Fri, 11 Sep 2017 06:41:40 GMT
7   ETag: W/"1f-EG5i0hq/hYhF0BsuheD9hNXeBpI"
8   X-Powered-By: Express
9
10  {
11      "id": 1,
12      "name": "abhinavsarkar"
13  }
```

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

```
1   $ http PATCH http://localhost:4000/v1/user/1 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: Fri, 11 Sep 2017 06:43:17 GMT
7   ETag: W/"27-JQsh12xu/rEFdWy8REF4NMtBUB4"
8   X-Powered-By: Express
9
10  {
11      "error": "User name must not be empty"
12  }
```

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:

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

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

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

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

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

```
1   $ http GET http://localhost:4000/v1/users
2   HTTP/1.1 200 OK
3   Connection: keep-alive
4   Content-Length: 65
5   Content-Type: application/json; charset=utf-8
6   Date: Fri, 11 Sep 2017 07:06:27 GMT
7   ETag: W/"41-btt9uNdG+9A1RO7SCLOsyMmIyFo"
8   X-Powered-By: Express
9
10  [
11      {
12          "id": 1,
13          "name": "abhinavsarkar"
14      },
15      {
16          "id": 2,
17          "name": "sarkarabhinav"
18      }
19  ]
```

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.