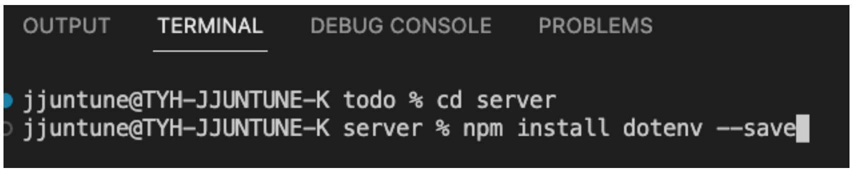
# Exercise. Todo (part 6) – Environment variables and managing database connection

At the moment, backend consists of only one file, which includes all software logic and other necessary code. In this exercise database manipulation logic will be implemented as a separate file/module. Also, use of environment variables is described.

Key learnings for this part are:

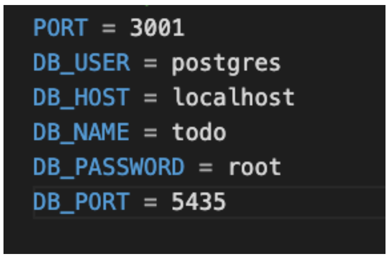
- Creating .env file for configuration
- Creating a separate module for managing database connection and executing query

Environment variables are used to create configuration file for the backend. Instead of hardcoding, for example, database connection details into code a separate configuration file can be created. Documentation for using env file is available on https://www.npmjs.com/package/dotenv. Start by installing required node module.



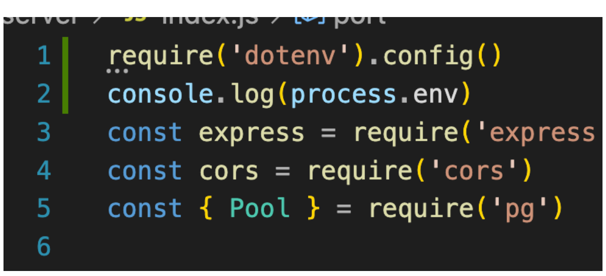
```
jjuntune@TYH-JJUNTUNE-K todo % cd server
jjuntune@TYH-JJUNTUNE-K server % npm install dotenv --save
```

Create .env file under server folder containing port and connection details for database. This example uses local database (or configuration might be different depending on the used development environment).



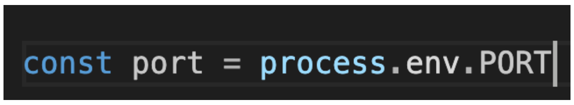
```
PORT = 3001
DB_USER = postgres
DB_HOST = localhost
DB_NAME = todo
DB_PASSWORD = root
DB_PORT = 5435
```

On index.js add require statement to use environment variables.



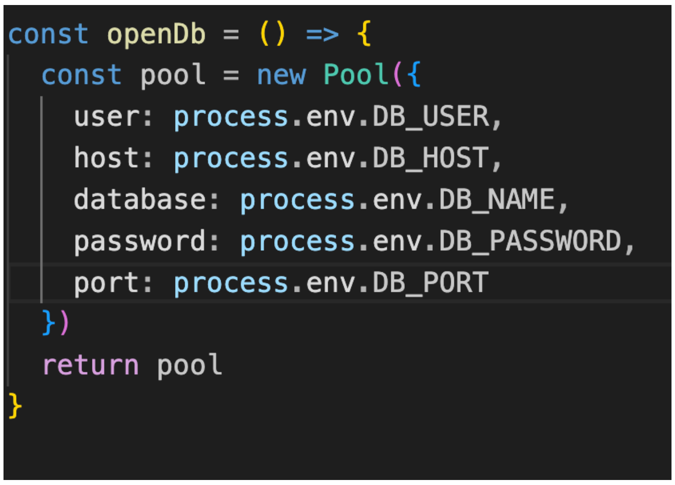
```
require('dotenv').config()
console.log(process.env)
const express = require('express
const cors = require('cors')
const { Pool } = require('pg')

```

Instead of using hardcoded value read port from .env file.



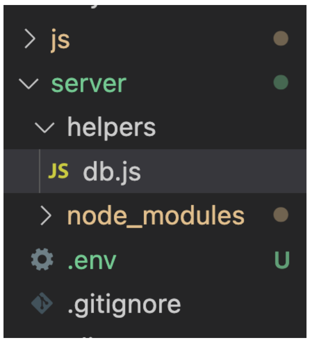
```
const port = process.env.PORT
```

Instead of using hardcoded values for database connection read data from .env file.

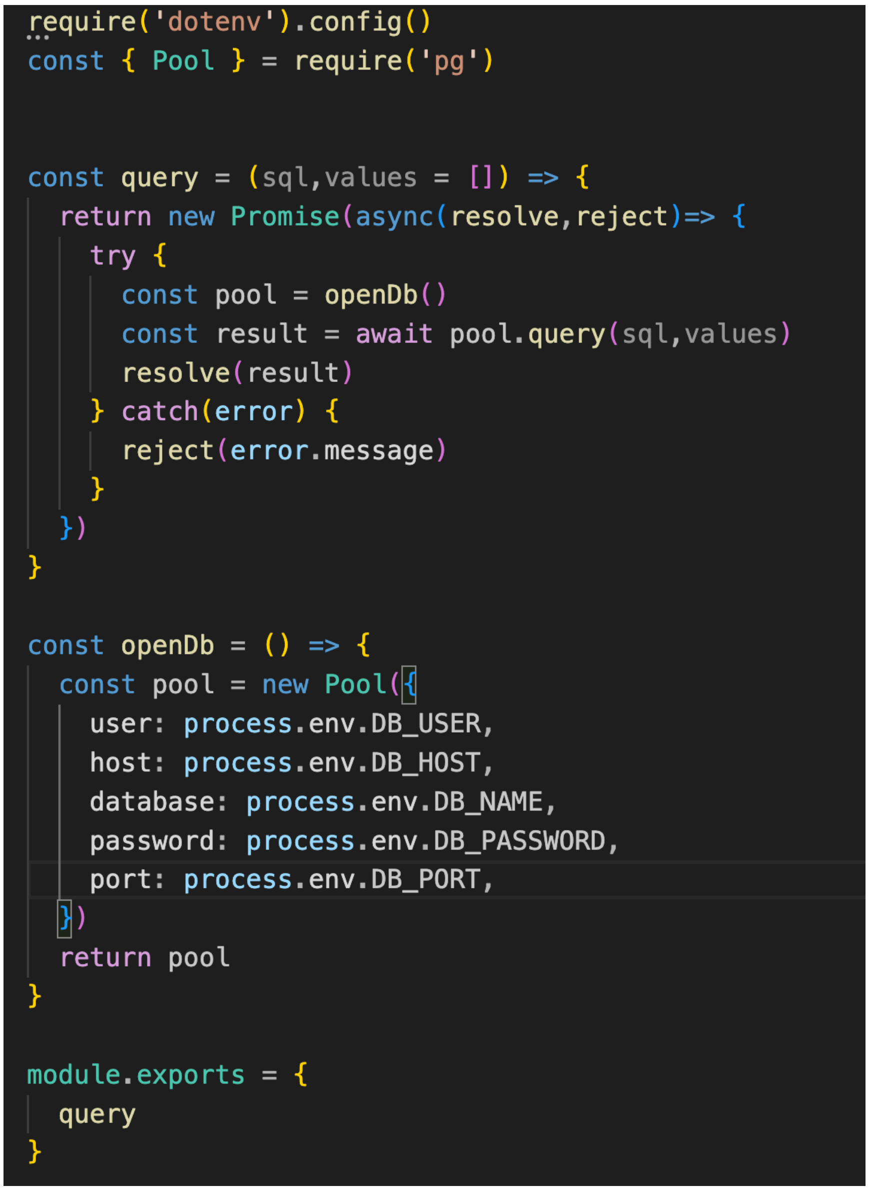


```
const openDb = () => {
  const pool = new Pool({
    user: process.env.DB_USER,
    host: process.env.DB_HOST,
    database: process.env.DB_NAME,
    password: process.env.DB_PASSWORD,
    port: process.env.DB_PORT
  })
  return pool
}
```

Database manipulation logic will be implemented as a separate file. Create helpers folder under server folder and file under name db.js.

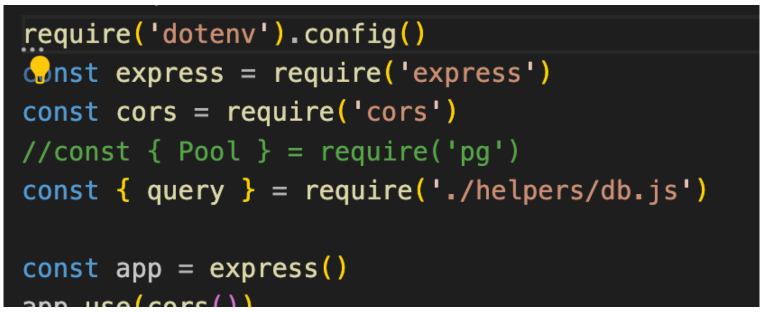


This file contains function for opening database and this function is not callable outside this file/module. There are also exported function that can execute sql statements (select, insert, delete, …) to the database. Function receives executed sql statement and possible values/parameters for sql. Values/parameters have default value, so value can be omitted (no values passed), if it is not necessary. Function returns a promise.



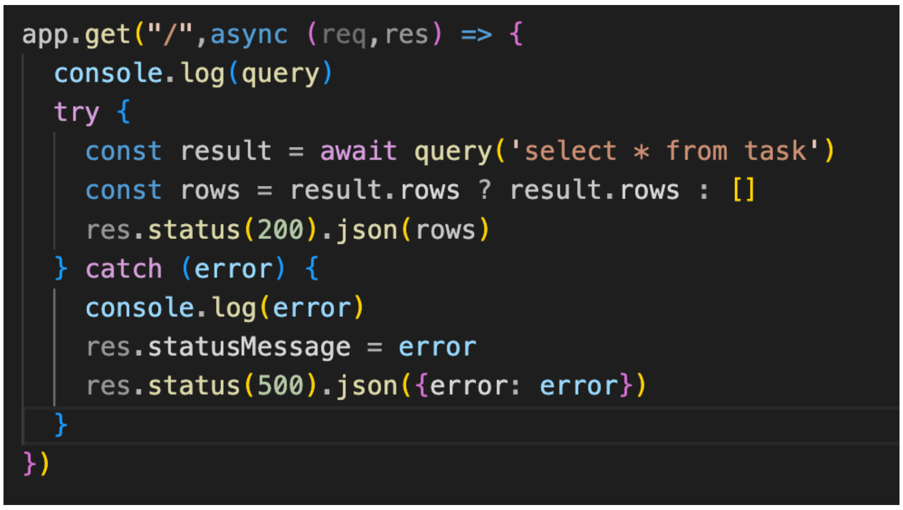
```
require('dotenv').config()
const { Pool } = require('pg')


const query = (sql,values = []) => {
  return new Promise(async(resolve,reject)=> {
    try {
      const pool = openDb()
      const result = await pool.query(sql,values)
      resolve(result)
    } catch(error) {
      reject(error.message)
    }
  })
}

const openDb = () => {
  const pool = new Pool({
    user: process.env.DB_USER,
    host: process.env.DB_HOST,
    database: process.env.DB_NAME,
    password: process.env.DB_PASSWORD,
    port: process.env.DB_PORT,
  })
  return pool
}

module.exports = {
  query
}
```

On index.Js remove unnecessary require statements. Index.js will be modified in a way that it does not handle database logic but it is done through query function. You may also remove openDb function from index.js (since it is now implemented on db.js).



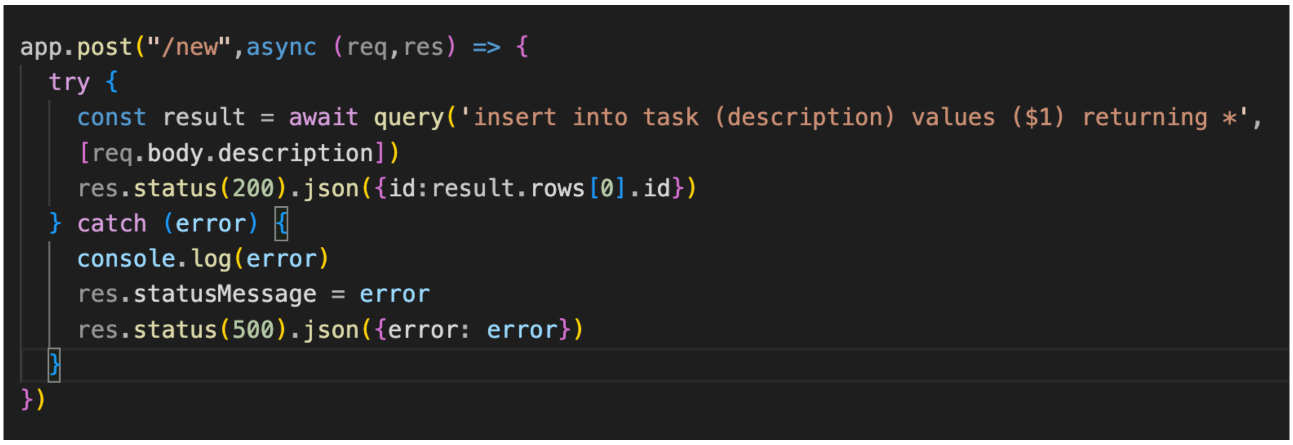
```
require('dotenv').config()
const express = require('express')
const cors = require('cors')
//const { Pool } = require('pg')
const { query } = require('./helpers/db.js')

const app = express()
app.use(cors())
```

App.get route returns tasks from backend. Select query without parameters should be executed and therefore query function is called by passing select statement and second parameter is not provided. Result is returned from query as promise and therefore await keyword is used to “wait” that asynchronous code returning promise is executed/resolved.  There is also a check that if database is empty there will be no rows and empty array is then returned as JSON. Since await is used inside function it must be declared to be asynchronous by using keywork async. Possible error is catched and returned to the client. There is also console.log for error which can be sometimes useful when developing the code (errors are printed to the console where NodeJS app is running).



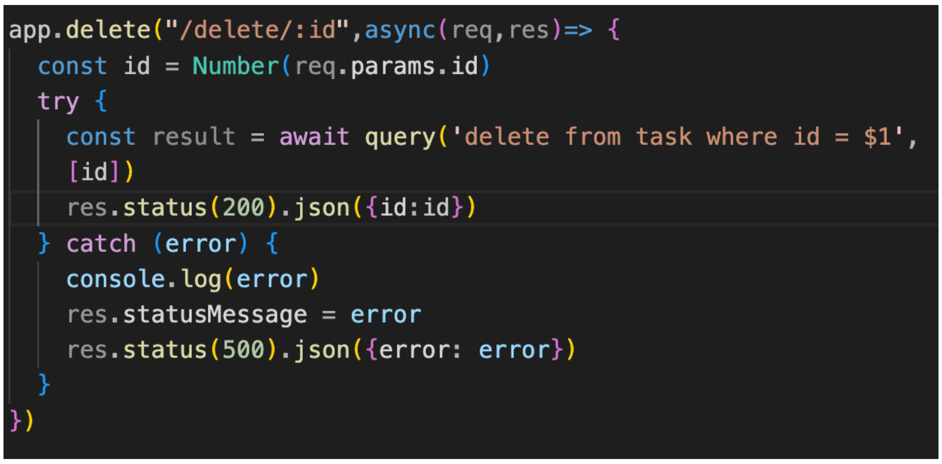
```
app.get("/",async (req,res) => {
  console.log(query)
  try {
    const result = await query('select * from task')
    const rows = result.rows ? result.rows : []
    res.status(200).json(rows)
  } catch (error) {
    console.log(error)
    res.statusMessage = error
    res.status(500).json({error: error})
  }
})
```

App.post is modified to use query the same way. In this case second parameter is also passed to the query function to contain data that is inserted into database.



```
app.post("/new",async (req,res) => {
  try {
    const result = await query('insert into task (description) values ($1) returning *',
    [req.body.description])
    res.status(200).json({id:result.rows[0].id})
  } catch (error) {
    console.log(error)
    res.statusMessage = error
    res.status(500).json({error: error})
  }
})
```

Also modify App.delete. Backend should be now working. Database logic is now moved into db.ts. There is no need to repeat opening database and executing sql multiple times on the backend code.



```
app.delete("/delete/:id",async(req,res)=> {
  const id = Number(req.params.id)
  try {
    const result = await query('delete from task where id = $1',
    [id])
    res.status(200).json({id:id})
  } catch (error) {
    console.log(error)
    res.statusMessage = error
    res.status(500).json({error: error})
  }
})
```