CSE1500: Web - Lecture 6 - Node.js & PostgreSQL

Lecture Summary

The WDT Team

Ujwal Gadiraju, Sagar Chethan Kumar, Ciprian Ionescu

Contents

⚠️ These summaries are not extensive overviews of lecture content and **DO NOT** cover all possible exam material. Rather, these serve as a refresher and capture **main points** for a given lecture.

In addition to a general warning of the lecture summaries, we would like to emphasize that for this lecture we are **NOT** asking you to memorize the `node-postgres` syntax or methods but rather expect you to be able to interpret relevant code and explain theory related concepts.

Learning Goals

You should be able to do the following after following the **lecture** and **assignment**:

- **Develop server-side code for authorization, cookies, sessions, and database access. (LO5)**

1 Node.js & PostgreSQL

1.1 `node-postgres` & Dynamic Queries

`node-postgres` also referred to as `pg` is a non-blocking PostgreSQL client for Node.js. In simple terms, `node-postgres` can be described as a collection of Node.js modules that are used to interface with a PostgreSQL database!

For the sake of this summary, we will roughly follow the `cricket_db` tutorial interjecting where necessary to provide further theory and explanations. The goal of this is to explain through an example how a Node.js app can be constructed with an Express server to perform CRUD operations on a database (while making use of PostgreSQL).

As a refresher from the previous lectures and tutorials, your Express server and back-end setup are logically defined with a bodyParser and a set of routes/endpoints. See below for the example used in the tutorial:

index.js

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const db = require('./queries')
const port = 3000

app.use(bodyParser.json())
app.use(bodyParser.urlencoded({extended: true,}))

app.get('/', (request, response) => {
    response.json({ info: 'Node.js, Express, and Postgres' })
  })

app.get('/players', db.getPlayers)
app.get('/players/:player_id', db.getPlayerByID)
app.post('/players', db.createPlayer)
app.put('/players/:player_id', db.updatePlayer)
app.delete('/players/:player_id', db.deletePlayer)
```

As covered in the previous lectures, we can define various routes to listen for various HTTP requests, including `GET`, `PUT`, `POST`, and `DELETE` - our example includes all of these. When starting the server we can make use of our browser, Postman[1], or Thunder Client to interface with our server and make requests. Currently, the Express server is running and sending static JSON responses.

Now to be able to enable frequent **dynamic queries** within our application we can make use of a connection pool[2] with `node-postgres`! A pool is typically defined like this:

[1] Postman is a very popular API platform that can be used to test or develop APIs. You can find out more about it here.

[2] You can find out more about connection pooling here

queries.js

```
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'username',
  host: 'host',
  database: 'dbname',
  password: 'password',
  port: PORT,
})
```

By making use of a `pool` we can query with our database. Naturally, we could just write many `pool.query(_)` statements, however, we can make use of Express's routes to properly define our CRUD operations under separate endpoints with corresponding SQL queries.

queries.js

```
// We assume the pool is defined as shown above.
const getPlayers = (req, res) => {
    pool.query('SELECT * FROM players ORDER BY player_id ASC', (error, results) => {
      if (error) {
        throw error
      }
      res.status(200).json(results.rows)
    })
  }

// Omitted other routes, for readability, refer to the tutorial for the whole file.

// *** Exporting the CRUD functions in a REST API ***
module.exports = {
    getPlayers,
    getPlayerByID,
    createPlayer,
    updatePlayer,
    deletePlayer,
  }
```

As a reminder, we have to also retrieve all the exported functions from `queries.js` which we did with `const db = require('./queries')` in `index.js`. Having, done so we are able to define our `app._` routes in `index.js` with their respective HTTP request methods.

And that is it! Thanks to `node-postgres` our server, database, and API are all set up allowing us to run frequent dynamic queries. Be sure to make use of Thunder Client or Postman to actually test your HTTP requests and endpoints.

2 Sample Question Types

(1) Explain why the use of packages like `node-postgres` helps to interface with a PostgreSQL database. What steps are needed to facilitate frequent dynamic queries?

(2) What is the missing code (denoted by the XXXX) that is required to make this `node-postgres` query work as intended?

```
const getPlayerByID = (req, res) => {
    const player_id = parseInt(XXXX)

    pool.query('SELECT * FROM players WHERE player_id = $1', [player_id], (error,
    ↪  results) => {
      if (error) {
        throw error
      }
      res.status(200).json(results.rows)
    })
}
```

a. `res.params.player_id`

b. `req.params.player_id`

c. `player_id`

d. `res.rows`

3 Extra Exercises

If you are feeling confident on the topic of **Lecture 6** try out the following tasks [3]:

- Expand on the `cricket_db` database we defined in Lecture 6's tutorial! Add new entities, define their relationships, and model their endpoints.
- Alternatively, model another sport/activity you enjoy and refer to the tutorial we provided. Remember, while interfacing with the database, adhere to RESTful principles, as mentioned in Lecture 4!

4 References and Further Reading

Here are some useful resources for learning more about **Lecture 6**:

- `node-postgres` documentation
- PostgreSQL and Node.js Tutorial
- PostgreSQL and Node.js Explained

[3] Note: None of these exercises are mandatory nor provide a bonus.