Presentazione progetto Node.js

Di Ricci Matteo

App di Viaggi Orizon

Dopo aver installato le dependecies richieste, ho iniziato creando i file “script.js” e “db.js”.

All’interno del file script.js ho importato il framework Express.js, ho usato un Middleware per il parsing JSON e, una volta create le varie routes, le ho importate e associate all’interno del file.

```
1   const express = require('express');
2   const app = express();
3   const port = 3000;
4
5   //middleware per il parsing JSON
6   app.use(express.json());
7
8   // Importazione delle routes
9   const bigliettiRoutes = require('./routes/biglietti');
10  const utentiRoutes = require('./routes/utenti');
11  const acquistiRoutes = require('./routes/acquisti');
12
13  // Associazione delle routes
14  app.use('/api/biglietti', bigliettiRoutes);
15  app.use('/api/utenti', utentiRoutes);
16  app.use('/api/acquisti', acquistiRoutes);
17
18  app.listen(port, () => {
19      console.log(`Server in ascolto su http://localhost:${port}`);
20  });
```

Ho poi creato il file db.js per gestire la connessione al database. Per questo progetto, ho scelto di usare il database MySQL, che ho adoperato mediante MySQL Workbench.

Una volta finito il setup iniziale, ho iniziato a lavorare alle varie routes richieste.

```
const mysql = require('mysql2');

const db = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'root',
    database: 'orizon_viaggi'
});

//connessione al database
db.connect((err) => {
    if (err) {
        console.error('Errore di connessione:', err);
        return;
    }
    console.log('Connesso al database MySQL');
});

module.exports = db;
```

La route Biglietti

La prima richiesta era la creazione di una API RESTful JSON che consentisse l'inserimento, la modifica e la cancellazione di un prodotto venduto. Ho pensato che la scelta di un biglietto fosse una valida opzione.

Ho quindi creato la prima tabella biglietti con unico parametro il nome

```sql
-- Creazione del database
CREATE DATABASE IF NOT EXISTS orizon_viaggi;
USE orizon_viaggi;
-- Creazione della tabella 'biglietti'
CREATE TABLE IF NOT EXISTS biglietti (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(255) NOT NULL
);
```

Ho quindi creato il primo metodo POST per l'inserimento del biglietto. Ho utilizzato dei prepared statement per maggior sicurezza, e gestito le varie casistiche con gli status code di risposta.

```
const express = require('express');
const router = express.Router();
const db = require('../db');

//inserimento biglietto
router.post('/', (req, res) => {
    const { nome } = req.body;
    if (!nome) {
        return res.status(400).json({ errore: 'Nome richiesto' });
    }

    const sql = 'INSERT INTO biglietti (nome) VALUES (?)';
    db.execute(sql, [nome], (err, result) => {
        if (err) return res.status(500).json({ errore: err.message });
        res.status(201).json({ id: result.insertId, nome });
    });
});
```

Con la stessa logica ho gestito quindi il metodo PUT per modificare un biglietto già esistente e il metodo DELETE per cancellare un biglietto.

```
//modifica biglietto
router.put('/:id', (req, res) => {
    const { nome } = req.body;
    const { id } = req.params;

    if (!nome) {
        return res.status(400).json({ errore: 'Nome obbligatorio' });
    }

    const sql = 'UPDATE biglietti SET nome = ? WHERE id = ?';
    db.execute(sql, [nome, id], (err, result) => {
        if (err) return res.status(500).json({ errore: err.message });
        if (result.affectedRows === 0) {
            return res.status(404).json({ errore: 'Biglietto non trovato' });
        }
        res.json({ id, nome });
    });
});
```

```
//cancellazione biglietto
router.delete(':/id', (req, res) => {
    const { id } = req.params;

    const sql = 'DELETE FROM biglietti WHERE id = ?';
    db.execute(sql, [id], (err, result) => {
        if (err) return res.status(500).json({ errore: err.message });
        if (result.affectedRows === 0) {
            return res.status(404).json({ errore: 'Biglietto non trovato' });
        }
        res.json({ messaggio: 'Biglietto eliminato con successo' });
    });
});

module.exports = router;
```

La route Utenti

La seconda richiesta prevedeva un’API che permettesse l’inserimento, la modifica e la cancellazione di un utente avente come caratteristiche il nome, il cognome e l’email.

Come prima cosa ho quindi aggiunto la tabella utenti al database.

```sql
-- Creazione della tabella 'utenti'
CREATE TABLE IF NOT EXISTS utenti (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(255) NOT NULL,
    cognome VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE
);
```

Dopodiché ho creato il primo metodo POST per l'aggiunta di utenti alla tabella, il metodo PUT per la modifica…

```
const express = require('express');
const router = express.Router();
const db = require('../db');

// Inserimento utente
router.post('/', (req, res) => {
    const { nome, cognome, email } = req.body;
    if (!nome || !cognome || !email) {
        return res.status(400).json({ errore: 'Nome, cognome ed email sono richiesti' });
    }

    const sql = 'INSERT INTO utenti (nome, cognome, email) VALUES (?, ?, ?)';
    db.execute(sql, [nome, cognome, email], (err, result) => {
        if (err) {
            if (err.code === 'ER_DUP_ENTRY') {
                return res.status(409).json({ errore: 'Email già registrata' });
            }
            return res.status(500).json({ errore: err.message });
        }
        res.status(201).json({ id: result.insertId, nome, cognome, email });
    });
});
```

```
// Modifica utente
router.put('/:id', (req, res) => {
    const { nome, cognome, email } = req.body;
    const { id } = req.params;

    if (!nome || !cognome || !email) {
        return res.status(400).json({ errore: 'Nome, cognome ed email sono richiesti' });
    }

    const sql = 'UPDATE utenti SET nome = ?, cognome = ?, email = ? WHERE id = ?';
    db.execute(sql, [nome, cognome, email, id], (err, result) => {
        if (err) {
            if (err.code === 'ER_DUP_ENTRY') {
                return res.status(409).json({ errore: 'Email già registrata' });
            }
            return res.status(500).json({ errore: err.message });
        }
        if (result.affectedRows === 0) {
            return res.status(404).json({ errore: 'Utente non trovato' });
        }
        res.json({ id, nome, cognome, email });
    });
});
```

…e per finire, il metodo DELETE per la cancellazione degli utenti.

```
// Cancellazione utente
router.delete('/:id', (req, res) => {
    const { id } = req.params;
    const sql = 'DELETE FROM utenti WHERE id = ?';

    db.execute(sql, [id], (err, result) => {
        if (err) return res.status(500).json({ errore: err.message });
        if (result.affectedRows === 0) {
            return res.status(404).json({ errore: 'Utente non trovato' });
        }
        res.json({ messaggio: 'Utente eliminato con successo' });
    });
});

module.exports = router;
```

La route Acquisti

Per l'ultima richiesta, che richiedeva l'inserimento, la modifica e la cancellazione dell'ordine di vendita di un biglietto, ho usato la stessa logica delle prime 2 API.

Ho proceduto quindi a creare la tabella acquisti, questa volta però aggiungendo 2 FOREIGN KEY che fanno riferimento agli ID contenuti nelle tabelle "biglietti" e "utenti".

Questo perché l'ultima richiesta era anche di poter visualizzare e filtrare gli ordini in base alla data di inserimento e in base al biglietto.

```sql
-- Creazione della tabella 'acquisti'
CREATE TABLE IF NOT EXISTS acquisti (
    id INT AUTO_INCREMENT PRIMARY KEY,
    id_biglietto INT NOT NULL,
    id_utente INT NOT NULL,
    data_acquisto DATE DEFAULT (CURRENT_DATE),
    FOREIGN KEY (id_biglietto) REFERENCES biglietti(id) ON DELETE CASCADE,
    FOREIGN KEY (id_utente) REFERENCES utenti(id) ON DELETE CASCADE
);
```

Per fare ciò, ho creato una query SQL dinamica che seleziona da “acquisti” uniti a “utenti” e “biglietti” per mostrare ID dell’acquisto, data dell’acquisto, nome, cognome ed email dell’utente ed infine il nome del biglietto. Ho definito la costante params con array di valori da passare al prepared statement. Se data ed id_biglietto sono presenti nella query string, allora aggiunge anche i relativi filtri. Dopodiché i dati vengono restituiti dalla data più recente a quella più vecchia.

```
// Visualizzazione di tutti gli acquisti
router.get('/', (req, res) => {
    const { data, id_biglietto } = req.query;


    let sql = `
    SELECT
    a.id,
    DATE(a.data_acquisto) AS data_acquisto,
    u.nome AS nome_utente,
    u.cognome AS cognome_utente,
    u.email,
    b.nome AS nome_biglietto
    FROM acquisti a
    JOIN utenti u ON a.id_utente = u.id
    JOIN biglietti b ON a.id_biglietto = b.id
    WHERE 1=1
    `;
    const params = [];

    if (data) {
        sql += 'AND DATE(a.data_acquisto) = ?';
        params.push(data);
    }

    if (id_biglietto) {
        sql += ' AND a.id_biglietto = ?';
        params.push(id_biglietto);
    }

    sql += ' ORDER BY a.data_acquisto DESC';

    db.execute(sql, params, (err, result) => {
        if (err) return res.status(500).json({ errore: err.message });
        res.json(result);
    });
});


module.exports = router;
```

Questo conclude la mia presentazione! L’intero codice sorgente ed il file README.md sono visualizzabili al mio github tramite il seguente link:
- https://github.com/MrBrollo/AppViaggi.

Vi ringrazio anticipatamente per il vostro feedback!

Ricci Matteo