Relazione Progetto Tecnologie Internet

Maggio 2024

Contents

1 Dependencies

In questa sezione si trovano tutti quei pacchetti che vanno installati per il corretto funzionamento dell'applicazione:

1.1 Tecnologie da installare

Essendo il progetto svolto in React e Flask dobbiamo andare ad installare rispettivamente node e python:

- install node
- install python

1.2 Pacchetti da installare lato client

Ora che abbiamo i linguaggi di programmazione installati dobbiamo spostarci nella cartella "frontend" tramite terminale e andare ad installare i seguenti pacchetti:

- npm install
- npm i react-router-dom
- npm axios

1.3 Pacchetti da installare lato server

Passando invece alla cartella "backend" andiamo ad installare tramite terminale i seguenti pacchetti:

- pip3 install flask
- pip3 install Flask-Cors

2 Frontend

3 Backend

3.1 Import e setup

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
from flask import g
import os
import sqlite3
from werkzeug.security import generate_password_hash
from werkzeug.security import check_password_hash


app = Flask(__name__)
cors = CORS (app, origins="*")

#the secret key is generated through this piece of code:
# import secrets;
# print(secrets.token_hex())
#
#as suggested from the flask documentation
app.secret_key = 'a421c210278fd00c726cf138acbe3780410109e3857df5b7475d847cd4813a31'
```

3.1.1 Import

Il nostro file inizia con l'import di tutte le librerie che andremo ad usare: lato server, per la crittografia e per il database.

In particolare andiamo ad importare il framework Flask, scelto per la sua semplicità e minimalismo, inoltre andiamo ad importare tutte quelle librerie che ci servono per interagire con il client ricevendo ed inviando rispettivamente con "request" e "jsonify".

La seconda sezione di import riguarda tutto ciò di cui abbiamo bisogno per il database: g è un oggetto che useremo per salvare, temporaneamente, e condividere i dati in parti differenti della nostra applicazione.

L'import del modulo os serve per gestire la posizione dei file, in particolare del database in questo caso.

Come ultimo import in questa sezione abbiamo il modulo sqlite3, che ci consente di creare, inizializzare e utilizzare il nostro database.

Le utile due righe nella sezione di import servono per i moduli necessari rispettivamente alla creazione e al controllo dell'hash associati alla crittografia della password.

3.1.2 Setup di base

Le prime due righe di codice servono ad inizializzare la nostra applicazione in flask e a configurare CORS (Cross-Origin Resource Sharing), quest'ultimo ci

serve per consentire richieste da fonti esterne. L'asterisco associato al parametro origins specifica che le richieste possano avvenire da qualunque origine (dominio o indirizzo IP).

L'ultima riga di questa sezione è quella che riguarda l'inizializzazione della chiave segreta che useremo per la crittografia della password.

3.2 Setup Database

Questa sezione di codice, invece, è dedicata alla creazione, inizializzazione e chiusura del database sqlite3, già presente in Flask e ottimo per gestire carichi di lavoro moderati come in questo caso:

```
#as suggested from the flask documentation
app.secret_key = 'a421c210278fd00c726cf138acbe3780410109e3857df5b7475d847cd4813a31'

app.config.from_object(__name__)

app.config.update(dict(
    DATABASE=os.path.join(app.root_path, 'database.db'),
    SCHEMA=os.path.join(app.root_path, 'schema.sql')
))
```

Questa porzione di codice serve ad impostare quelle che sono le variabili di configurazione per l'inizializzazione e utilizzo del database.

3.2.1 Connessione con il database

```
#connects to the specific sqlite3 database
def connect_db():
    rv = sqlite3.connect(app.config['DATABASE'])
    rv.row_factory = sqlite3.Row
    return rv
```

Questa funzione serve per stabilire una connessione con uno specifico database sqlite3, rv è l'oggetto di connessione al database, che potrà poi essere usato per interagire con lo stesso.

3.2.2 Aprire una nuova connessione con il database

```python
#opens a new connection for this context g
def get_db():
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db
```

Funzione che ci permette, nel caso non fosse già presente, di instaurare una nuova connessione e memorizzarne il contenuto nell'oggetto g, permettendoci così di utilizzarla in diverse parti dell'applicazione.

3.2.3 Chiusura della connessione

```python
#this is executed every time the context is teardown, it closes the connection
@app.teardown_appcontext
def close_db(error):
    if hasattr(g, 'sqlite_db'):
        g.sqlite_db.close()
```

Questa funzione invece serve per concludere la connessione con il database allorché l'applicazione Flask termini di utilizzarlo, in modo da evitare problemi di risorse non rilasciate.

3.2.4 Inizializzazione del database

```python
#this is the function called in the flask script defined below, it uses the schema to initialize the database
#IT WILL DROP EVERY TABLE PRESENT IF CALLED
def init_db():
    db = get_db()
    with app.open_resource(app.config['SCHEMA'], mode='r') as f:
        db.cursor().executescript(f.read())
    db.commit()
```

Questa funzione serve per inizializzare il database all'interno dell'applicazione Flask: prima si invoca la funzione get_db in modo da ottenere la connessione con il database, poi andiamo ad aprire il file dello schema, in modalità lettura. Dopo aver ottenuto un oggetto cursos, andremo ad utilizzare tale oggetto per eseguire lo script che abbiamo letto nel file schema.sql; utilizziamo db.commit per confermare le modifiche appena fatte.

3.2.5 Inizializzazione da linea di comando

```python
#definition of the flask script, now you can write "flask initdb" in the terminal to initialize the database
@app.cli.command('initdb')
def initdb_command():
    """Initializes the database."""
    init_db()
    print('Initialized the database.')
```

Quest'ultima porzione di codice, per quanto riguarda la sezione dedicata al database, ci permette di inizializzare il database tramite linea di comando "flask initdb". Una volta dato questo comando il nostro database sarà inizializzato e pronto per essere utilizzato.

Da tenere in considerazione che tutte le volte che questo comando viene lanciato il database sarà azzerato, perdendo tutti i dati al proprio interno.

3.3 API

Ora entriamo nella sezione dedicata alle API, funzioni che ci permettono di comunicare con il nostro client e nelle quali andremo ad utilizzare effettivamente il database che abbiamo creato nella porzione precedente di codice.

3.3.1 Invio della classifica

```python
#sending the scoreboards data
@app.route("/api/score/<int:number>", methods=['GET'])
def send_score(number):
    #if the number in the url is equal to 1 we need snake's data
    if number==1:
        try:
            db = get_db()
            cur = db.cursor()
            query = 'SELECT distinct username, score, game_id FROM Matches WHERE game_id=1'
            cur.execute(query)
            results = cur.fetchall()
            db.commit()

            scores = []
            for row in results:
                score = {
                    "player": row[0],
                    "score": row[1],
                    "id": row[2]
                }
                scores.append(score)

            return jsonify(scores)

        except Exception as e:
            return jsonify({'message': 'Error: {}'.format(str(e))})
```

```
 #if number is 2, flappy bird's data is sent
elif number==2:
    try:
        db = get_db()
        cur = db.cursor()
        query = 'SELECT distinct username, score, game_id FROM Matches WHERE game_id=2'
        cur.execute(query)
        results = cur.fetchall()
        db.commit()

        scores = []
        for row in results:
            score = {
                "player": row[0],
                "score": row[1],
                "id": row[2]
            }
            scores.append(score)

        return jsonify(scores)

    except Exception as e:
        return jsonify({'message': 'Error: {}'.format(str(e))})
else:
    #otherwise something happened
    return "invalid number provided"
```

Questa funzione di tipo GET, che quindi invia dati dal server al client, serve per inviare i punteggio, id del gioco, e nome utente, legati tra loro, tenuti nel database, al client in modo da poter mostrare le classifiche divise per gioco.

La richiesta del client avviene ad un url specifico che termina con un valore numero intero, tale valore numerico assume un valore differente per specificare di quale gioco si vuole ricevere i punteggi.

Una volta che abbiamo identificato il gioco, tramite un semplice if statement, dobbiamo andare ad utilizzare le funzioni per instaurare la connessione con il database per poter effettuare la query ed estrarre le informazioni di cui abbiamo bisogno dal database.

Salviamo tutte le tuple ottenute in una variavbile, tramite il metodo fetchll, che che ci consente di tenere tutti i risultati e non solo uno, e poi andiamo a creare una lista di json con i risultati ottenuti; lista che poi inviamo al client tramite apposito metodo.

Il tutto viene fatto in regime di try catch per intercettare un qualunque errore che potrebbe avvenire e notificare di conseguenza il client.

3.3.2 Salvare il punteggio delle partite

```python
#receivin the match data and inserting it into the database
@app.route("/api/game", methods=['POST'])
def receive_game_score():
    data = request.get_json()
    playername = data['playername']
    score = data['score']
    gameId = data['gameId']

    try:
        db = get_db()
        cur = db.cursor()
        cur.execute('INSERT INTO Matches (game_id, score, username) VALUES (?, ?, ?)', (gameId, score, playername))
        db.commit()
        return jsonify({'message': 'Done'})

    except Exception as e:
        return jsonify({'message': 'Error: {}'.format(str(e))})
```

Questa funzione di tipo POST, che quindi invia dati dal client al server, serve per ricevere e i dati ottenuti da una partita giocata; i dati passati dal client in formato json vengono poi messi nel database tramite la funzione INSERT.

Queste operazioni vengono tutte eseguite in regime di try catch in modo da intercettare un qualunque errore e notificare il client sia in caso di operazione performata con successo, sia in caso in cui sia avvenuto un errore.

3.3.3 Controllo accesso di un utente

```python
#login control
@app.route("/api/login", methods =['POST'])
def get_data():
    #client sends username and password
    data = request.get_json()
    username = data['username']
    password = data['password']


    db = get_db()
    cur = db.cursor()
    #extract the password hash from the database if present
    cur.execute('SELECT username, user_password FROM User WHERE username = ?', (username,))

    user = cur.fetchone()

    if user is None:
        #the query result is empty, the credentials are worng!
        return jsonify({'message': 'Incorrect'})
    elif not check_password_hash(user["user_password"], password):
        #check_password_hash() returned false! the passwords do not match
        return jsonify({'message': 'Incorrect'})
    else :
        #everything is good, the user can login
        return jsonify({'message': 'Correct'})
```

Questa funzione serve per permettere l'accesso ad utente già registrato. La funzione riceve dal client il nome utente e la password inseriti: per prima cosa controlla che lo username esista, se esiste allora passiamo a controllare che anche la password, dopo aver usato l'apposita funzione per decriptarla, corrisponda. Il nome utente è univoco essendo chiave primaria nello schema sql, motivo per cui si usa la fatchone e non la fatchall.

Al termine notifichiamo il client con il risultato che ottenuto dalla verifica.

3.3.4 Iscrizione nuovo utente

```python
#sign in a new user
@app.route("/api/signin", methods=['POST'])
def save_data():

    #get username and password from client
    data = request.get_json()
    username = data['username']
    password = data['password']


    #since username is a primary key for the User table no duplicates are allowed, so no check is needed, if the username is invalid
    #the exception will be executed
    try:
        db = get_db()
        cur = db.cursor()
        cur.execute('INSERT INTO User (username, user_password) VALUES (?, ?)', (username, generate_password_hash(password)))
        db.commit()
        return jsonify({'message': 'Done'})

    except Exception as e:
        # the username was already used
        return jsonify({'message': 'Error: {}'.format(str(e))})
```

L'ultima funzione che abbiamo è quella per permette ad un nuovo utente di iscriversi: riceviamo ancora una volta il nome utente e la password dal client, sempre in formato json, dopo di che andiamo ad aggiungere questi valori al database. L'unico errore che possiamo ottenere è che il nome utente sia già presente nel database, essendo chiave primaria non possono essere presenti due nomi utenti uguali, in tal caso andiamo a notificare il client che non siamo riusciti a performare l'operazione richiesta.

3.3.5 Sezione dedicare all'esecuzione dell'app (facoltativa)

```python
if __name__ == '__main__':
    app.run(debug=True)
```

Queste sono le due ultime righe del nostro backend e sono facoltative: ci servono solo per avviare la nostra applicazione e farlo con la modalità debug attiva.

3.4 Database

Il nostro database, fatto in sqlite3, ha una struttura relazionale e si basa su tre tabelle sql.

```sql
DROP TABLE IF EXISTS User;
DROP TABLE IF EXISTS Game;
DROP TABLE IF EXISTS Matches;

CREATE TABLE User (
  username VARCHAR(30) PRIMARY KEY,
  user_password VARCHAR(30) NOT NULL
);

CREATE TABLE Game (
  game_id INTEGER PRIMARY KEY,
  game_name VARCHAR(30) NOT NULL
);

CREATE TABLE Matches (
  match_id INTEGER PRIMARY KEY AUTOINCREMENT,
  score INTEGER NOT NULL,
  game_id INTEGER NOT NULL,
  username VARCHAR(30) NOT NULL,
  FOREIGN KEY (game_id) REFERENCES Game(game_id),
  FOREIGN KEY (username) REFERENCES User(username)
);

INSERT INTO Game (game_id, game_name)
VALUES
    ('1', 'snake'),
    ('2', 'flappy');
```

Andiamo a creare 3 tabelle: User, Game e Matches.

La tabella User è quella che andrà a contenere il nome utente e la password di ogni utente; essendo username chiave primaria questo evita che ci possano essere due utenti con lo stesso nome.

Game invece contiene l'id identificativo associato a ciascun gioco (1 per snake e 2 per flappy) e viene inizializzato direttamente con questi valori al suo interno quando viene creato il database.

Matches è invece la tabella che si occupa di mettere in relazione le altre due: infatti contiene sia lo username che il game_id, ma in aggiunta troviamo il match_id, valore che identifica in modo univoco ogni partita giocata, e il punteggio ottenuto in tale partita. Infine troviamo anche esplicitati i vincoli di integrità referenziale.