

# Relazione Progetto Tecnologie Internet

Maggio 2024

## Contents

<b>1</b>	<b>Dependencies</b>	<b>2</b>
1.1	Tecnologie da installare . . . . .	2
1.2	Pacchetti da installare lato client . . . . .	2
1.3	Pacchetti da installare lato server . . . . .	2
<b>2</b>	<b>Frontend</b>	<b>2</b>
2.1	Componenti . . . . .	2
2.1.1	Card . . . . .	3
2.1.2	Login . . . . .	4
2.1.3	Navbar . . . . .	5
2.1.4	Scoreboard . . . . .	5
2.2	Giochi . . . . .	5
2.2.1	Snake . . . . .	6
2.2.2	FlappyBird . . . . .	6
<b>3</b>	<b>Backend</b>	<b>6</b>
3.1	Import e setup . . . . .	6
3.1.1	Import . . . . .	6
3.1.2	Setup di base . . . . .	7
3.2	Setup Database . . . . .	7
3.2.1	Connessione con il database . . . . .	8
3.2.2	Aprire una nuova connessione con il database . . . . .	8
3.2.3	Chiusura della connessione . . . . .	8
3.2.4	Inizializzazione del database . . . . .	9
3.2.5	Inizializzazione da linea di comando . . . . .	9
3.3	API . . . . .	9
3.3.1	Invio della classifica . . . . .	10
3.3.2	Salvare il punteggio delle partite . . . . .	11
3.3.3	Controllo accesso di un utente . . . . .	12
3.3.4	Iscrizione nuovo utente . . . . .	12
3.3.5	Invio informazioni giochi . . . . .	13
3.3.6	Sezione dedicare all'esecuzione dell'app (facoltativa) . . . . .	13
3.4	Database . . . . .	13

# 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

## 2.1 Componenti

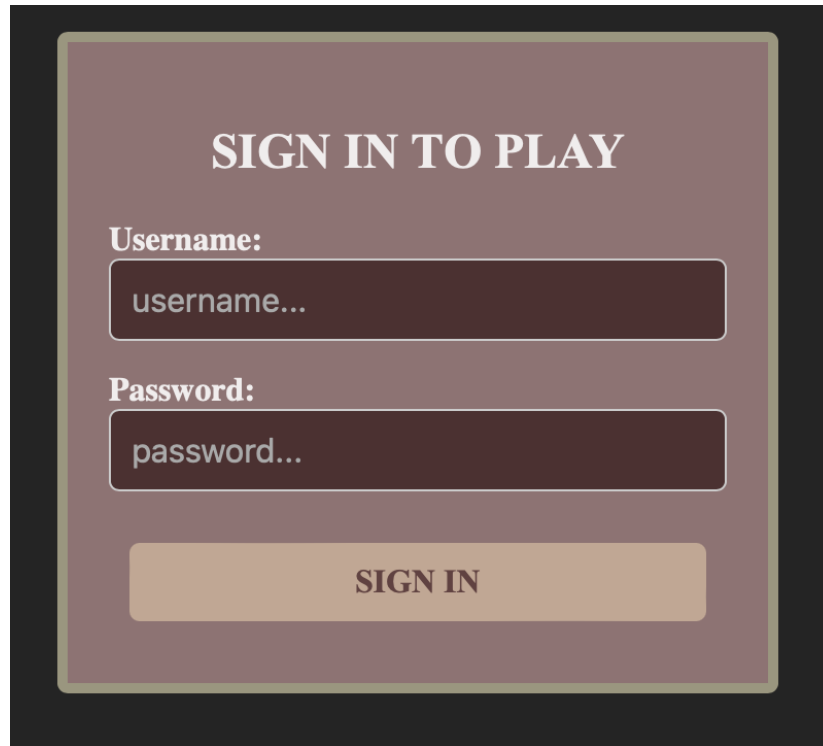
il progetto è stato realizzato con il framework di React in quanto è tra i più utilizzati al mondo attualmente. Questo ci permette di scrivere meno codice definendo delle Componenti che poi verranno riutilizzate nelle varie pagine del sito web senza dover scrivere ulteriore codice. Sotto vengono presentate tutte le componenti React del progetto.

### 2.1.1 Card



Questa componente è una semplice scheda utilizzata per indicare i giochi presenti nel sito, è composta da un semplice flexbox contenente degli elementi paragrafo per il nome e una semplice descrizione e un'immagine. successivamente questa componente verrà racchiusa in un componente link di React-Router-dom che sul click dell'utente reindirizza alla pagina del gioco rappresentato. Card è presente solo nella pagina principale del sito.

### 2.1.2 Login



The image shows a login form titled "SIGN IN TO PLAY". It has a dark background with a lighter, rounded rectangular area in the center. Inside this area, the title "SIGN IN TO PLAY" is at the top. Below it, there are two labels: "Username:" and "Password:". Each label is followed by a text input field. The "Username:" field contains the placeholder text "username...". The "Password:" field contains the placeholder text "password...". At the bottom of the form, there is a large, light-colored button with the text "SIGN IN" in dark letters.

Questa componente è la stessa sia per Login che per Sign in, è molto simile a Card in come si presenta, tuttavia ha una differenza sostanziale in termini di javascript. il bottone del componente è associato a una funzione chiamata 'handleLogin'

```
//verification of the login based on the backend response
const handleLogin = (event) => {
  event.preventDefault();
  const currUser = event.target.username.value;
  const currPassword = event.target.password.value;
  location.pathname.endsWith("/") ? sendDataLogin(currUser, currPassword) : sendDataSignIn(currUser, currPassword)
  localStorage.setItem("username", currUser);
  localStorage.setItem("password", currPassword);
}
```

questa funzione raccoglie i dati inseriti dall'utente nel form e in base alla route, con cui si decide quale delle due versioni del componente utilizzare, li invia al backend per verificare le credenziali o per creare un nuovo account tramite le due funzioni 'sendDataSignIn' e 'sendDataLogIn'. come ultima cosa salva i valori inseriti dall'utente nella memoria locale del browser così che siano reperibili in qualunque momento, questi valori sono poi eliminati dalla funzione di logout in una componente successiva.

Questa componente è montata solo nella pagina iniziale del sito

### 2.1.3 Navbar



Navbar è una componente molto semplice, è composta solo da dei bottoni associati a dei link e uno associato alla funzione di logout prima citata. Presenta due stati possibili, quando la componente è montata nella pagina principale del sito ha solo due bottoni che collegano le pagine di classifica, mentre quando è montata in una pagina di classifica, si rende accessibile anche un bottone per ritornare alla pagina principale. Anche la funzione di logout è molto semplice in quanto elimina solamente i dati dell'utente dalla memoria locale. Navbar è presente nella pagina principale e nelle pagine di classifica.

### 2.1.4 Scoreboard



Scoreboard è una componente che raccoglie i dati riguardanti un gioco specifico dal backend e li riordina in una lista da presentare all'utente illuminando i risultati che lo interessano. L'unico aspetto interessante della componente è come ricava i dati da presentare. Questo viene fatto tramite Axios, un Client HTTP per node.js, che permette di inoltrare richieste HTTP e ricevere i dati in formato JSON. Scoreboard è presente solo nelle pagine di classifica.

## 2.2 Giochi

La sezione è dedicata a una analisi più approfondita del funzionamento del codice dei giochi. Non sono il focus del progetto, in quanto non utilizzano tecnologie particolari se non React.js. La sezione è mirata a facilitare la lettura del codice, se necessario.

### 2.2.1 Snake

### 2.2.2 FlappyBird

Il gioco si basa essenzialmente sul generare degli ostacoli che devono essere evitati dal giocatore. Gli ostacoli sono inseriti in una variabile di stato di React e sono renderizzati offscreen ogni 2 secondi.

Innanzitutto si definiscono delle variabili di stato necessarie per il controllo del gioco.

```
//game variables and state
const [birdPosition, setBirdPosition] = useState({ x: 50, y: 200 });
const [pipes, setPipes] = useState([]);
const [gameOver, setGameOver] = useState(false);
const [score, setScore] = useState(0);
const [gameStarted, setGameStarted] = useState(false);
const username = localStorage.getItem('username');
const [dataSent, setDataSent] = useState(false);
```

## 3 Backend

### 3.1 Import e setup

```
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 file inizia con l'import di tutte le librerie verranno utilizzate in seguito: lato server, per la crittografia e per il database.

In particolare viene importato il framework Flask, scelto per la sua semplicità e minimalismo, inoltre si importano tutte le librerie che servono per interagire con il client ricevendo ed inviando rispettivamente con “request” e ”jsonify”.

La seconda sezione di import riguarda tutto il necessario per il database: `g` è un oggetto utilizzato per salvare, temporaneamente, e condividere i dati in parti differenti dell'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 si ha il modulo `sqlite3`, che consente di creare, inizializzare e utilizzare il database.

Le ultime 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 l'applicazione in flask e a configurare CORS (Cross-Origin Resource Sharing), quest'ultimo 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 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

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

### 3.2.3 Chiusura della connessione

```
#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

```
#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 si apre il file dello schema, in modalità lettura. Dopo aver ottenuto un oggetto `cursor`, questo si utilizza per eseguire lo script letto nel file `schema.sql`; `db.commit` confermare le modifiche appena fatte.

### 3.2.5 Inizializzazione da linea di comando

```
#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, permette di inizializzare il database tramite linea di comando “`flask initdb`”. Una volta dato questo comando il 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

Questa sezione è dedicata alle API, funzioni che permettono di comunicare con il client e nelle quali si utilizza effettivamente il database creato nella porzione precedente di codice.

### 3.3.1 Invio della classifica

```
#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 identificato il gioco, tramite un semplice if statement, si utilizzano le funzioni per instaurare la connessione con il database per poter effettuare la query ed estrarre le informazioni necessarie dal database.

Tutte le tuple ottenute sono salvate in una variabile, tramite il metodo fetchll, che consente di tenere tutti i risultati e non solo uno, poi una lista di json viene creata con i risultati ottenuti; lista che poi sarà 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

```
#receiving 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

```
#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 wrong!
        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 controlla 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 fetchone e non la fetchall.

Al termine il client é notificato con il risultato ottenuto dalla verifica.

### 3.3.4 Iscrizione nuovo utente

```
#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))})
```

Questa funzione è quella per permettere ad un nuovo utente di iscriversi: riceve ancora una volta il nome utente e la password dal client in formato json, dopo di che aggiunge questi valori al database. L'unico errore che può verificarsi è che il nome utente sia già presente nel database, essendo chiave primaria non possono essere presenti due nomi utenti uguali, in tal caso il client è notificato che l'operazione richiesta non è andata a buon fine.

### 3.3.5 Invio informazioni giochi

```
#send information about the games names and descriptions
@app.route("/api/info", methods=['GET'])
def send_info():
    try:
        db = get_db()
        cur = db.cursor()
        # extract game name and description from db
        cur.execute('SELECT game_id, game_name, game_description FROM Game')
        games = cur.fetchall()
        return jsonify([{"game_id": game[0], "game_name": game[1], "game_description": game[2]} for game in games])
    except Exception as e:
        return jsonify({'message': 'Error: {}'.format(str(e))})
```

Quest'ultima funzione serve per inviare le informazioni relative ai giochi. Esegue una query per estrarre tutte le informazioni a disposizione nel database e poi le invia al client in formato JSON.

### 3.3.6 Sezione dedicare all'esecuzione dell'app (facoltativa)

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

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

## 3.4 Database

Il database in sqlite3 ha una struttura relazionale e si basa su tre tabelle 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,
    game_description VARCHAR(100) 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, game_description)
VALUES
    ('1', 'snake', 'Move your snake and let him eat apples'),
    ('2', 'flappy', 'Make your bird fly avoiding obstacles');

```

sono create 3 tabelle: User, Game e Matches.

La tabella User è quella che contiene 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), il nome del gioco e la sua descrizione. 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 e in aggiunta il match\_id, valore che identifica in modo univoco ogni partita giocata, e il punteggio ottenuto in tale partita. Infine sono esplicitati i vincoli di integrità referenziale.