Universitatea Tehnică „Gheorghe Asachi” din Iași

Facultatea de Inginerie Electrică, Energetică și Informatică Aplicată

Proiect TAD

Specializarea: Sisteme informatice de monitorizare a mediului

Sistem HTTP folosind API RESTful integrat cu o bază de date

-Aplicație server-client-

Coordonator științific,
Conferențiar dr. Ing.
Damian Cătălin

Masterand,
Isachi Mihai

Semestrul II – 2024

Documentație : https://github.com/MEEEshu/TAD

Cuprins

Memoriu justificativ

În contextul creșterii nevoii de acces rapid și precis la date meteorologice pentru diverse aplicații, am identificat utilizarea unui API (Application Programming Interface) ca o soluție eficientă pentru colectarea și gestionarea acestor informații.

Acest memoriu justificativ explică motivele și beneficiile utilizării unui API pentru obținerea datelor meteorologice și detaliază pașii necesari pentru implementarea acestuia. Beneficiile dorite ale unui sistem de monitorizare a mediului cu acces în timp real sunt: furnizarea datelor meteorologice cât mai precis într-un timp cât mai scurt, informarea populației la cerere, acces la un istoric meteorologic și contribuție la cercetarea științifică.

I.Introducere

Accesul la date meteorologice precise și actualizate este esențial pentru numeroase industrii și aplicații, inclusiv agricultură, transport, energie, și planificarea evenimentelor. Utilizarea unui API pentru obținerea acestor date prezintă multiple avantaje:

1)Actualizare în timp real: API-urile permit accesul la date meteorologice actualizate constant, oferind informații precise despre condițiile meteorologice curente și previziunile pe termen scurt și lung.

2)Automatizare: API-urile facilitează automatizarea colectării și procesării datelor meteorologice, reducând necesitatea intervenției manuale și minimizând erorile umane.

3)Scalabilitate: API-urile sunt scalabile, permițând accesul la date pentru mai multe locații simultan și gestionarea volumelor mari de date fără probleme.

4)Integrare facilă: API-urile pot fi integrate cu ușurință în diverse aplicații și sisteme existente, oferind flexibilitate și compatibilitate cu diferite platforme și tehnologii.

5)Costuri reduse: Utilizarea unui API reduce costurile asociate cu achiziționarea, întreținerea și actualizarea echipamentelor meteo și a infrastructurii necesare pentru colectarea datelor.

Alegerea unui API care oferă date meteorologice precise și fiabile. Exemple de API-uri populare includ OpenWeatherMap, AirVisual, și Api Ninjas. Fiecare dintre acestea oferă date variate, de la condițiile actuale și previziuni, până la istoricul meteorologic. Acesta este favorizat într-o serie de domenii, inclusiv sistemele de poziționare în interior, automatizarea inteligentă a locuinței și agricultura inteligentă .

Dezvoltarea unui modul de integrare care să extragă datele meteorologice necesare din API-ul selectat, actualizarea requesturilor cu integrarea unei baze de date legate de specificații auto si să le proceseze conform cerințelor specifice ale aplicației noastre.

Implementarea unui API pentru obținerea datelor de orice tip va aduce numeroase beneficii proiectului nostru, inclusiv:

- Eficiență operațională: Automatizarea proceselor de colectare și analiză a datelor va reduce timpul și efortul necesar pentru obținerea informațiilor necesare clientului.
- Decizii informate: Accesul la date precise și actualizate va permite luarea unor decizii mai bune și mai rapide, bazate pe informațiile date de bazele de date apelate sau create.
- Flexibilitate și adaptabilitate: Soluția propusă va fi ușor de adaptat pentru a răspunde nevoilor viitoare și pentru a integra noi funcționalități și surse de date.

Pentru acest proiect, am creat o aplicație web folosind Flask care încorporează o bază de date SQLite pentru a gestiona datele mașinii. Aplicația are, de asemenea, capacitatea de a utiliza o varietate de API-uri externe pentru a prelua și afișa date despre populație și vreme. Voi trece în revistă părțile principale ale aplicației, designul bazei de date și caracteristicile API în secțiunile care urmează.

II.Descrierea, funcționalitatea și importanța interfeței cu bază de date relațională

Introducerea bazelor de date într-un API reprezintă un pas esențial în dezvoltarea aplicațiilor moderne, oferind o structură robustă și scalabilă pentru gestionarea datelor. Prin intermediul bazelor de date, API-urile pot stoca, actualiza, șterge și interoga eficient informațiile necesare pentru funcționarea aplicației. Acest proces implică integrarea unui sistem de gestionare a bazelor de date cu API-ul, configurarea conexiunilor și implementarea operațiilor CRUD (Create, Read, Update, Delete).[1]

II.1. API cu bază de date integrată

O colecție de date stocate și aranjate electronic se numește bază de date. De la simple fișiere text până la sisteme complexe care gestionează milioane de înregistrări și relațiile dintre ele, toate pot fi considerate baze de date. Bazele de date sunt utilizate în contextul unei API pentru a menține persistența datelor, garantând disponibilitatea și esențialitatea lor pe termen lung.

În funcție de cerințele aplicației, există diferite tipuri de baze de date, fiecare având avantaje și dezavantaje proprii:

1)Relațională (SQL): utilizează ca bază un model cu tabele și relații de date. PostgreSQL, MySQL și SQLite sunt câteva exemple. Aplicațiile care necesită o integritate puternică a datelor și suport pentru tranzacții complicate sunt cele mai potrivite pentru acestea.[2]

2)NoSQL: În aceste baze de date nu se folosesc modelele tradiționale de tip tabelar. Redis, Cassandra și MongoDB sunt câteva exemple. Acestea sunt potrivite pentru aplicațiile care trebuie să gestioneze o varietate de tipuri de date cu o scalabilitate și o flexibilitate excelente.[2]

II.1.1 Structura back-end a proiectului cu baza de date mySQL integrată

Pentru a controla interacțiunea cu baza de date și pentru a oferi funcții suplimentare referitoare la populație și la datele meteorologice, aplicația Flask oferă o serie de rute. Elementele cheie sunt descrise în detaliu mai jos:

II.1.1.1.Rute Flask

Ruta / servește ca punct de intrare principal al aplicației.

Ruta /<username> afișează o pagină personalizată pentru utilizatorul specificat.

Ruta /ProiectTAD servește fișierul HTML interfataPROIECT.html.

Ruta /Masinute servește fișierul HTML interfata_client.html.

Ruta /API servește fișierul HTML interfata_clientv2.html.

II.1.1.2.Funcționalități API pentru Gestionarea Mașinilor:

Ruta /api/cars (metoda GET) returnează toate înregistrările din tabela cars.

Ruta /api/cars/<int:car_id> (metoda GET) returnează detaliile unei mașini specificate prin index.

Ruta /api/cars (metoda POST) adaugă o nouă mașină în baza de date.

Ruta /api/cars/<int:car_id> (metoda POST) adaugă o nouă mașină specificând ID-ul acesteia.

Ruta /api/cars/<int:car_id> (metoda PATCH) permite actualizarea parțială a unei mașini.

Ruta /api/cars/<int:car_id> (metoda PUT) permite actualizarea completă a unei mașini.

Ruta /api/cars/<int:car_id> (metoda DELETE) șterge o mașină specificată prin ID.

Baza de date SQLite conține o singură tabelă principală cars, structurată astfel: id-ul sau indexul unic pentru fiecare mașină, brand-ul mașinii, modelul brand-ului și specificațiile mașinii.

Această aplicație arată cum se poate lega o bază de date SQLite cu o API Flask și cum se pot adăuga mai multe funcționalități prin obținerea de date relevante pentru utilizator prin apeluri la API-uri externe. Ar fi simplu să extindeți acest proiect pentru a include mai multe funcționalități și pentru a îmbunătăți experiența utilizatorului.

II.1.2 Structura front-end a proiectului cu baza de date mySQL integrată

Prin intermediul unei interfețe web, cadrul client HTML este configurat pentru a oferi o experiență interactivă utilizatorului. Aceasta cuprinde mai multe părți pentru afișarea, adăugarea, actualizarea și eliminarea mașinilor, în plus față de afișarea altor date cruciale, cum ar fi ora și data din acest moment și cantitatea de interogări API care au fost efectuate.

II.1.2.1. Structura <head> :

Conține elementele <title> și <style>.

<title>: Definește titlul paginii web afișat în bara de titlu a browser-ului.

<style>: Conține stilurile integrate pentru a stiliza pagina HTML.

II.1.2. 2. Structura <body> :

Titlul Principal (<h1>): Afișează titlul aplicației.

Secțiunea de Afișare a Tuturor Mașinilor:

<h2>: Subtitlul secțiunii.

<button>: Buton pentru a iniția cererea de afișare a tuturor mașinilor.

<pre id="carList">: Container pentru afișarea listei de mașini într-un format JSON.

Secțiunea de Adăugare a Mașinilor:

<h2>: Subtitlul secțiunii.

<label> și <input>: Etichete și câmpuri de intrare pentru marca (‘brand’), modelul(‘model’) și anul mașinii (‘year’.

<button>: Buton pentru a iniția adăugarea unei noi mașini.

Secțiunea de Actualizare a Mașinilor:

<h2>: Subtitlul secțiunii pentru actualizarea unei mașini existente.

<label> și <input>: Etichete și câmpuri de intrare pentru ID-ul mașinii (updateCarId), noua marcă (updateBrand), noul model (updateModel) și noul an (updateYear).

<button>: Buton pentru a iniția actualizarea sau adăugarea unei mașini.

Secțiunea de Ștergere a Mașinilor:

<h2>: Subtitlul secțiunii pentru ștergerea unei mașini.

<label> și <input>: Etichete și câmpuri de intrare pentru ID-ul mașinii (deleteCarId).

<button>: Buton pentru a iniția ștergerea unei mașini.

Afișează Data și Ora Curentă:

<h2>: Subtitlul secțiunii pentru afișarea datei și orei curente.

<p id="currentDateTime">: Paragraf pentru afișarea datei și orei curente, actualizat în timp real.

Afișează Numărul de Cereri:

<h2>: Subtitlul secțiunii pentru afișarea numărului de cereri.

<p id="requestCount">: Paragraf pentru afișarea numărului de cereri efectuate, inițial setat la 0.

<script>:

Conține funcții JavaScript pentru gestionarea interacțiunilor cu API-ul și actualizarea interfeței utilizatorului.

getCars(): Obține toate mașinile de la server și actualizează lista afișată.

addCar(): Adaugă o nouă mașină prin trimiterea datelor către server.

updateCar(): Actualizează o mașină existentă pe server.

deleteCar(): Șterge o mașină de pe server.

displayCurrentDateTime(): Afișează data și ora curentă, actualizată la fiecare secundă.

updateRequestCount(): Actualizează și afișează numărul de cereri efectuate.

II.2. Structura API cu cheie

Una dintre cele mai importante sarcini în dezvoltarea aplicațiilor actuale este introducerea de date într-un client de la o API cu cheie. Această procedură presupune achiziția și utilizarea de date dintr-o varietate de surse externe, inclusiv statistici demografice și meteorologice.

II.2.1. Backend-ul aplicației

Un cadru web Python numit Flask simplifică crearea de aplicații online, iar codul furnizat utilizează Flask pentru a construi un server web care va răspunde la solicitările din partea frontală și va servi API-uri.

Backend-ul face apeluri către alte API-uri, cum ar fi API-urile de informații despre vreme și despre oraș, utilizând datele de intrare pe care utilizatorii le furnizează prin intermediul cererilor HTTP. Pentru a trimite cereri HTTP către serverele API externe și a obține răspunsuri JSON, care sunt apoi procesate și trimise înapoi la frontend, aceste apeluri sunt efectuate cu ajutorul modulului requests.

Prin intermediul endpoint-ului specificat, backend-ul (Python cu Flask) primește aceste date, procesează cererea și returnează datele relevante. Pe baza rezultatelor, interfața cu utilizatorul este apoi actualizată folosind datele primite de la backend.[3]

II.2.1.1. Funcțiile apelate in HTML – partea de script din codul HTML

1)Funcția padZero(num): această funcție este responsabilă pentru adăugarea unui zero în fața numerelor mai mici decât 10, astfel încât să obținem o formatare uniformă a datelor de timp.

2)Funcția setInterval(() => {...}, 1000: această funcție este utilizată pentru a actualiza data și ora afișate în interfață o dată pe secundă.

3)Funcția getWeather(): această funcție este apelată atunci când utilizatorul accesează butonul "Obține informații". Ea este responsabilă pentru:

- Colectarea datelor de intrare (latitudine și longitudine) introduse de utilizator.
- Trimiterea acestor date către backend pentru a obține informații despre oraș, populație și date meteorologice.
- Actualizarea interfeței cu datele primite de la backend.
- Integrarea cu Backend-ul.

4)Funcția getWeather(): trimite datele introduse de utilizator către backend prin intermediul unei cereri fetch.

Aceste funcții din codul HTML reprezintă interfața de utilizator prin care utilizatorii interacționează cu aplicația și trimit cereri către backend pentru a obține și afișa datele corespunzătoare.

II.2.1.2. Structura backend a aplicației prin codul Python

Inițializarea aplicației Flask: La începutul codului Python, aplicația Flask este inițializată folosind constructorul Flask(__name__), unde constructorul __name__ îl gasim și la clase.

Definirea rutelor (endpoints): Cu ajutorul decoratorului @app.route(), sunt definite mai multe rute pentru a gestiona diferitele solicitări HTTP primite de la client. De exemplu, ruta / este asociată funcției home(), care returnează un mesaj de bun venit în cazul în care se accesează pagina principală a aplicației.

Target-ul unor fișiere statice: Folosind funcția send_from_directory(), backend-ul servește fișiere statice, precum fișierele HTML și CSS, către frontend pentru a fi afișate în browser.

II.2.2.1. Structura frontend a aplicației

Structura generală a acestui cod HTML implică mai multe elemente, inclusiv formulare, butoane și div-uri pentru afișarea datelor:

1)Tag-ul head: Conține metadatele documentului HTML, precum și referințe către fișierele CSS și scripturi JavaScript necesare pentru stilizare și funcționarea paginii.

2)Tag-ul body: Aici este definită întreaga interfață utilizatorului.

3)Titlul: Titlul paginii este definit folosind tag-ul <h1>, indicând utilizatorului că este pe o pagină pentru obținerea informațiilor meteo.

4)Containerul principal: Este reprezentat de un div cu clasa container. Acesta are un aspect vizual înglobat și este centrul paginii. Conține formulare pentru introducerea latitudinii și longitudinii, un buton pentru obținerea informațiilor, precum și un div pentru afișarea rezultatelor.

5)Formularul de introducere a datelor: Utilizatorul poate introduce latitudinea și longitudinea în aceste câmpuri de input.

6)Butonul de obținere a informațiilor: Atunci când acest buton este apăsat, este activată funcția JavaScript getWeather(), care trimite datele introduse către backend pentru prelucrare.

7)Div-ul de afișare a rezultatelor: Aici sunt afișate rezultatele primite de la backend, cum ar fi orașul, populația, temperatura și umiditatea.

Codul sursă

Python:

```python
from flask import Flask, jsonify, request, send_from_directory
import sqlite3
import requests



# Inițializarea aplicației Flask
app = Flask(__name__)
```

```python
# Endpoint pentru pagina principală a aplicației
@app.route('/')
def home():
    print('A fost accesată pagina lui Isachi Mihai /')
    return "<h3>Bine ai venit pe pagina mea!</h3> Sper să-ți placă. Wish you well </br>"

# Endpoint pentru pagina personalizată a utilizatorului
@app.route('/<username>')
def Username(username):
    print('primit:', username)
    return f"<h3>Bine ai venit pe pagina mea {username}!</h3> Sper să-ți placă. Wish you well </br>"


# Endpoint pentru a servi fișierul HTML pentru cele doua interfete
@app.route('/ProiectTAD')
def serve_ProiectTAD():
    return send_from_directory('.', 'interfataPROIECT.html')

# Endpoint pentru a servi fișierul HTML pentru interfața mașinilor
@app.route('/Masinute')
def serve_interface_masini():
    return send_from_directory('.', 'interfata_client.html')

# Endpoint pentru a servi fișierul HTML pentru interfața API-ului
@app.route('/API')
def serve_interface_api():
    return send_from_directory('.', 'interfata_clientv2.html')


# Restul codului dvs. rămâne neschimbat...

# Endpoint pentru obținerea tuturor mașinilor din baza de date
@app.route('/api/cars', methods=['GET'])
def get_cars():
    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM cars")
    cars = cursor.fetchall()
    conn.close()
    return jsonify(cars)

# Endpoint pentru obținerea unei mașini după ID din baza de date
@app.route('/api/cars/<int:car_id>', methods=['GET'])
def get_car(car_id):
    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM cars WHERE id=?", (car_id,))
```

```
    car = cursor.fetchone()
    conn.close()
    if car is None:
        return jsonify({'error': 'Mașina nu a fost găsită'}), 404
    return jsonify(car)

# Endpoint pentru adăugarea unei noi mașini în baza de date
@app.route('/api/cars', methods=['POST'])
def add_car():
    data = request.json
    if 'brand' not in data or 'model' not in data or 'an' not in data:
        return jsonify({'error': 'Toate câmpurile (brand, model, an) sunt
obligatorii'}), 400
    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO cars (brand, model, an) VALUES (?, ?, ?)",
(data['brand'], data['model'], data['an']))
    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': 'Mașina a fost adăugată cu succes!'}), 201

# Endpoint pentru adăugarea unei mașini cu un anumit ID în baza de date
@app.route('/api/cars/<int:car_id>', methods=['POST'])
def add_car_with_id(car_id):
    data = request.json
    if 'brand' not in data or 'model' not in data or 'an' not in data:
        return jsonify({'error': 'Toate câmpurile (brand, model, an) sunt
obligatorii'}), 400
    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO cars (id, brand, model, an) VALUES (?, ?, ?,
?)", (car_id, data['brand'], data['model'], data['an']))
    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': f'Mașina cu ID-ul {car_id} a fost adăugată cu
succes!'}), 201

# Endpoint pentru actualizarea parțială a unei mașini după ID în baza de date
@app.route('/api/cars/<int:car_id>', methods=['PATCH'])
def partial_update_car(car_id):
    data = request.json
    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("UPDATE cars SET brand=?, model=?, an=? WHERE id=?",
(data.get('brand'), data.get('model'), data.get('an'), car_id))
    conn.commit()
```

```
    cursor.close()
    conn.close()
    return jsonify({'message': 'Mașina a fost actualizată cu succes!'})

# Endpoint pentru actualizarea parțială a întregii colecții de mașini în baza
de date
@app.route('/api/cars', methods=['PATCH'])
def partial_update_all_cars():
    data = request.json
    if not data:
        return jsonify({'error': 'Nu s-au furnizat date pentru actualizare'}),
400

    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()

    for car_id, update_data in data.items():
        if 'brand' in update_data or 'model' in update_data or 'an' in
update_data:
            cursor.execute("UPDATE cars SET brand=?, model=?, an=? WHERE
id=?", (update_data.get('brand'), update_data.get('model'),
update_data.get('an'), car_id))

    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': 'Actualizare parțială a mașinilor efectuată cu
succes!'})

# Endpoint pentru actualizarea completă a unei mașini după ID în baza de date
@app.route('/api/cars/<int:car_id>', methods=['PUT'])
def update_car(car_id):
    data = request.json
    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("UPDATE cars SET brand=?, model=?, an=? WHERE id=?",
(data['brand'], data['model'], data['an'], car_id))
    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': 'Mașina a fost actualizată cu succes!'})

# Endpoint pentru actualizarea completă a întregii colecții de mașini în baza
de date
@app.route('/api/cars', methods=['PUT'])
def update_all_cars():
    data = request.json
    if not data:
```

```
        return jsonify({'error': 'Nu s-au furnizat date pentru actualizare'}),
400

    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()

    for car_id, update_data in data.items():
        cursor.execute("UPDATE cars SET brand=?, model=?, an=? WHERE id=?",
(update_data['brand'], update_data['model'], update_data['an'], car_id))

    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': 'Actualizare completă a mașinilor efectuată cu
succes!'})

# Endpoint pentru ștergerea unei mașini după ID din baza de date
@app.route('/api/cars/<int:car_id>', methods=['DELETE'])
def delete_car(car_id):
    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("DELETE FROM cars WHERE id=?", (car_id,))
    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': 'Mașina a fost ștearsă cu succes!'})

# Endpoint pentru ștergerea întregii colecții de mașini din baza de date
@app.route('/api/cars', methods=['DELETE'])
def delete_all_cars():
    confirmation = request.args.get('confirm')

    conn = sqlite3.connect('cardata.db')
    cursor = conn.cursor()
    cursor.execute("DELETE FROM cars")
    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': 'Toate mașinile au fost șterse cu succes!'})

# Endpoint pentru testarea metodei TRACE
@app.route('/api/cars', methods=['TRACE'])
def trace():
    return jsonify({'message': 'Acesta este un test de tip TRACE. cod :
200'}), 200

# Endpoint pentru testarea metodei HEAD
@app.route('/api/cars', methods=['HEAD'])
```

```
def head():
    return '', 200

# Rularea aplicației Flask
if __name__ == '__main__':
    app.run(debug=True, port=5555)
```

interfataPROIECT.html

```
<!DOCTYPE html>
<html lang="ro">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Proiect TAD</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f5f5f5;
            margin: 0;
            padding: 0;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
        }

        .container {
            text-align: center;
            background-color: #fff;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }

        h1 {
            color: #333;
        }

        button {
            background-color: #4CAF50;
            color: white;
            padding: 10px 20px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            margin: 10px;
```

```
            font-size: 16px;
        }

        button:hover {
            background-color: #45a049;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Proiect TAD</h1>
        <button onclick="location.href='http://127.0.0.1:5555/API'">Accesează
API</button>
        <button
onclick="location.href='http://127.0.0.1:5555/Masinute'">Accesează
Masinute</button>
    </div>
</body>
</html>
```

interfata_client.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Sistem de Gestionare a Mașinilor</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
            color: #333;
            margin: 0;
            padding: 20px;
        }
        h1 {
            color: #555;
        }
        button {
            background-color: #4caf50;
            border: none;
            color: white;
            padding: 10px 20px;
            text-align: center;
            text-decoration: none;
            display: inline-block;
            font-size: 16px;
            margin: 4px 2px;
```

```
            cursor: pointer;
            border-radius: 5px;
        }
        input[type="number"],
        input[type="text"] {
            padding: 8px;
            margin: 4px;
            border-radius: 5px;
            border: 1px solid #ccc;
            box-sizing: border-box;
        }
        pre {
            padding: 10px;
            background-color: #fff;
            border-radius: 5px;
            border: 1px solid #ccc;
            overflow: auto;
        }
    </style>
</head>
<body>
    <h1>Sistem de Gestionare a Mașinilor</h1><!-- Secțiunea de Afișare a
Tuturor Mașinilor -->
    <div>
        <h2>Arată toate mașinile</h2>
        <button onclick="getCars()">Arată toate mașinile</button>
        <pre id="carList"></pre>
    </div>

    <!-- Secțiunea de Adăugare a Mașinilor -->
    <div>
        <h2>Adaugă Mașină</h2>
        <label for="brand">Marcă:</label>
        <input type="text" id="brand" placeholder="Marcă">
        <br>
        <label for="model">Model:</label>
        <input type="text" id="model" placeholder="Model">
        <br>
        <label for="year">An:</label>
        <input type="number" id="year" placeholder="An">
        <br>
        <button onclick="addCar()">Adaugă Mașină</button>
    </div>

    <!-- Secțiunea de Actualizare a Mașinilor -->
    <div>
        <h2>Actualizează sau adaugă Mașină</h2>
        <label for="updateCarId">ID-ul Mașinii de Actualizat:</label>
```

```
        <input type="number" id="updateCarId" placeholder="ID-ul Mașinii">
        <br>
        <label for="updateBrand">Noua Marcă:</label>
        <input type="text" id="updateBrand" placeholder="Noua Marcă">
        <br>
        <label for="updateModel">Noul Model:</label>
        <input type="text" id="updateModel" placeholder="Noul Model">
        <br>
        <label for="updateYear">Noul An:</label>
        <input type="number" id="updateYear" placeholder="Noul An">
        <br>
        <button onclick="updateCar()">Actualizează sau adaugă Mașină</button>
    </div>

    <!-- Secțiunea de Ștergere a Mașinilor -->
    <div>
        <h2>Șterge Mașină</h2>
        <label for="deleteCarId">ID-ul Mașinii de Șters:</label>
        <input type="number" id="deleteCarId" placeholder="ID-ul Mașinii">
        <br>
        <button onclick="deleteCar()">Șterge Mașină</button>
    </div>

    <!-- Afișează Data și Ora Curentă -->
    <div>
        <h2>Data și Ora Curentă:</h2>
        <p id="currentDateTime"></p>
    </div>

    <!-- Afișează Numărul de Cereri -->
    <div>
        <h2>Numărul de Cereri:</h2>
        <p id="requestCount">0</p>
    </div>

    <script>
        function getCars() {
            fetch('http://127.0.0.1:5555/api/cars')
            .then(response => response.json())
            .then(data => {
                document.getElementById('carList').textContent =
JSON.stringify(data, null, 2);
                updateRequestCount(); // Actualizează numărul de cereri după o
cerere reușită
            })
            .catch(error => console.error('Eroare:', error));
        }
```

```
        function addCar() {
            const brand = document.getElementById('brand').value;
            const model = document.getElementById('model').value;
            const year = document.getElementById('year').value;

            fetch('http://127.0.0.1:5555/api/cars', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ brand: brand, model: model, an: year
}),
            })
            .then(response => response.json())
            .then(data => {
                console.log('Succes:', data);
                updateRequestCount(); // Actualizează numărul de cereri după o
cerere reușită
            })
            .catch(error => console.error('Eroare:', error));
        }

        function updateCar() {
            const id = document.getElementById('updateCarId').value;
            const brand = document.getElementById('updateBrand').value;
            const model = document.getElementById('updateModel').value;
            const year = document.getElementById('updateYear').value;

            fetch(`http://127.0.0.1:5555/api/cars/${id}`, {
                method: 'PUT',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ brand: brand, model: model, an: year
}),
            })
            .then(response => response.json())
            .then(data => {
                console.log('Succes:', data);
                updateRequestCount(); // Actualizează numărul de cereri după o
cerere reușită
            })
            .catch(error => console.error('Eroare:', error));
        }

        function deleteCar() {
            const carId = document.getElementById('deleteCarId').value;
```

```
            fetch(`http://127.0.0.1:5555/api/cars/${carId}`, {
                method: 'DELETE',
            })
            .then(response => response.json())
            .then(data => {
                console.log('Succes:', data);
                updateRequestCount(); // Actualizează numărul de cereri după o
cerere reușită
            })
            .catch(error => console.error('Eroare:', error));
        }

        // Afișează data și ora curentă
        function displayCurrentDateTime() {
            const currentDate = new Date();
            document.getElementById('currentDateTime').textContent =
currentDate.toLocaleString();
        }

        // Actualizează data și ora curentă la fiecare secundă
        setInterval(displayCurrentDateTime, 1000);

        // Actualizează numărul de cereri
        function updateRequestCount() {
            const requestCountElement =
document.getElementById('requestCount');
            let currentCount = parseInt(requestCountElement.textContent);
            currentCount++;
            requestCountElement.textContent = currentCount;
        }
    </script>
</body>
</html>
```

interfata_clientv2.html

```
<!DOCTYPE html>
<html lang="ro">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Informații Meteo</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f5f5f5;
            margin: 0;
            padding: 0;
```

```
        }    .container {
        max-width: 600px;
        margin: 20px auto;
        background-color: #fff;
        padding: 20px;
        border-radius: 8px;
        box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    }

    h1 {
        text-align: center;
        color: #333;
    }

    label {
        display: block;
        margin-bottom: 5px;
        font-weight: bold;
        color: #555;
    }

    input[type="text"] {
        width: 100%;
        padding: 8px;
        margin-bottom: 10px;
        border: 1px solid #ccc;
        border-radius: 4px;
        box-sizing: border-box;
    }

    button {
        background-color: #4CAF50;
        color: white;
        padding: 10px 20px;
        border: none;
        border-radius: 4px;
        cursor: pointer;
    }

    button:hover {
        background-color: #45a049;
    }

    .result {
        margin-top: 20px;
    }

    .result p {
```

```
        margin-bottom: 8px;
        color: #333;
    }

    #numar_requesturi {
        color: #888;
    }

    #data_ora {
        color: #888;
    }
</style>
</head>
<body>
    <h1>Informații Meteo</h1><div class="container">
        <label for="latitudine">Latitudine:</label>
        <input type="text" id="latitudine" placeholder="Latitudine">

        <label for="longitudine">Longitudine:</label>
        <input type="text" id="longitudine" placeholder="Longitudine">

        <button onclick="getWeather()">Obține informații</button>

        <div class="result">
            <p id="oras">Orașul: <span id="city"></span></p>
            <p id="populatie">Populație: <span id="population"></span></p>
            <p id="temperatura">Temperatura: <span
id="temperature"></span>°C</p>
            <p id="umiditate">Umiditate: <span id="humidity"></span>%</p>
            <p id="numar_requesturi">Număr de request-uri: <span
id="request_count">0</span></p>
            <p id="data_ora">Data și ora: <span id="datetime"></span></p>
        </div>
    </div>

    <script>
    // Funcție pentru a adăuga un zero în fața numerelor mai mici decât 10
    function padZero(num) {
        return num < 10 ? `0${num}` : num;
    }

    // Actualizare data și ora o dată pe secundă
    setInterval(() => {
        const now = new Date();
        const datetimeString = `${now.getFullYear()}-${padZero(now.getMonth()
+ 1)}-${padZero(now.getDate())}
${padZero(now.getHours())}:${padZero(now.getMinutes())}:${padZero(now.getSecon
ds())}`;
```

```
        document.getElementById('datetime').textContent = datetimeString;
    }, 1000);

    let requestCount = 0; // Inițializăm counter-ul de request-uri

    function getWeather() {
        const latitudine = document.getElementById('latitudine').value;
        const longitudine = document.getElementById('longitudine').value;
        const api_key_openweathermap = '5d3293c93dbdd7f72c412b81cc5a5259';
        const api_key_airvisual = '41ed48c0-9c02-4d7a-8e77-1cbfcfff1713';
        const api_key_api_ninjas = 'A5oYGIf8UgK9vC0ZL0e12g==jZ8fdWw8jy7Guyrd';

        // Incrementăm counter-ul de request-uri la fiecare apel al funcției
        requestCount++;
        document.getElementById('request_count').textContent = requestCount;

        // Apelul către API pentru obținerea orașului
        const city_api_call =
`http://api.airvisual.com/v2/nearest_city?lat=${latitudine}&lon=${longitudine}
&key=${api_key_airvisual}`;

        fetch(city_api_call)
            .then(response => {
                if (!response.ok) {
                    throw new Error('A apărut o problemă la obținerea datelor
despre oraș.');
                }
                return response.json();
            })
            .then(data => {
                // Afișare oraș
                document.getElementById('city').textContent = data.data.city;

                // Apelul către API pentru obținerea populației
                const city_name = encodeURIComponent(data.data.city);
                const population_api_call = `https://api.api-
ninjas.com/v1/city?name=${city_name}&X-Api-Key=${api_key_api_ninjas}`;

                fetch(population_api_call)
                    .then(response => {
                        if (!response.ok) {
                            throw new Error('A apărut o problemă la obținerea
datelor despre populație.');
                        }
                        return response.json();
                    })
                    .then(data => {
                        // Afișare populație
```

```
                        document.getElementById('population').textContent =
data[0].population;
                    })
                    .catch(error => {
                        console.error('Eroare:', error.message);
                        document.getElementById('population').textContent =
'n/a';
                    });
            })
            .catch(error => {
                console.error('Eroare:', error.message);
                document.getElementById('city').textContent = 'n/a';
            });

        // Apelul către API pentru obținerea informațiilor meteorologice
        const weather_api_call =
`https://api.openweathermap.org/data/3.0/onecall/timemachine?lat=${latitudine}
&lon=${longitudine}&dt=1643803200&appid=${api_key_openweathermap}`;

        fetch(weather_api_call)
            .then(response => {
                if (!response.ok) {
                    throw new Error('A apărut o problemă la obținerea datelor
meteorologice.');
                }
                return response.json();
            })
            .then(data => {
                // Afișare temperatură (convertită din Kelvin în Celsius)
                const temperature = (data.data[0].temp - 273).toFixed(2);
                document.getElementById('temperature').textContent =
temperature;

                // Afișare umiditate
                const humidity = data.data[0].humidity;
                document.getElementById('humidity').textContent = humidity;
            })
            .catch(error => {
                console.error('Eroare:', error.message);
                document.getElementById('temperature').textContent = 'n/a';
                document.getElementById('humidity').textContent = 'n/a';
            });
    }
    </script>
</body>
</html>
```

Bibliografie

1) Ciprian-Octavian Truica, Alexandru Boicea, Ionut Trifan ”CRUD Operations in MongoDB ” Proceedings of the 2013 International Conference on Advanced Computer Science and Electronics Information (ICACSEI 2013)
2) Walter Kriha NoSQL Databases-Selected Topics on Software-Technology Ultra-Large Scale Sites - Computer Science and Media (CSM) University Hochschule der Medien, Stuttgart (Stuttgart Media University)
3) Ghimire, Devndra (2020) - Comparative study on Python web frameworks: Flask and Django