SOUTH PARK EPISODES DATABASE

Aplicatie Web Full-Stack pentru Gestionarea Episoadelor South Park

---

Universitatea: Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

Profesor Coordonator: Bogdan Florea

Disciplina: Tehnologii Web

Studenti: Eduard-Alexandru Kasaj & Vlad Stoica

Grupa: 445B

CUPRINS

---

1. DESCRIEREA GENERALA A PROIECTULUI

1.1 Introducere si Context

South Park Episodes Database este o aplicatie web full-stack dezvoltata in cadrul cursului de Tehnologii Web de la Universitatea Politehnica Bucuresti. Aplicatia ofera o platforma completa pentru vizualizarea, cautarea si gestionarea episoadelor din celebrul serial animat South Park. Proiectul implementeaza concepte avansate de web development, incluzand operatiuni CRUD complete, sistem de autentificare JWT, si functionalitati moderne de search si filtrare.

1.2 Arhitectura Generala

Aplicatia utilizeaza o arhitectura Client-Server cu separare completa intre frontend si backend. Aceasta alegere arhitecturala ofera multiple avantaje, inclusiv scalabilitate independenta a celor doua componente, flexibilitate in deployment, si posibilitatea de a dezvolta frontend si backend in paralel. Comunicarea intre cele doua componente se realizeaza prin intermediul unui API RESTful, folosind formatul JSON pentru transferul datelor.

Backend-ul este construit pe platforma Node.js folosind framework-ul Express.js, oferind un set de endpoint-uri REST pentru gestionarea episoadelor si autentificarea utilizatorilor. Datele sunt stocate intr-o baza de date MySQL, folosind o coloana de tip JSON pentru flexibilitate maxima in structurarea informatiilor. Autentificarea este implementata folosind JSON Web Tokens care sunt generate la login si verificate la fiecare cerere catre endpoint-urile protejate.

Frontend-ul este dezvoltat cu React versiunea 19.1.1, cea mai recenta versiune disponibila, folosind Vite ca build tool pentru o experienta de dezvoltare rapida. Interfata utilizatorului este construita cu Tailwind CSS, un framework utility-first care permite crearea rapida de design-uri moderne si responsive. Rutarea este gestionata de React Router versiunea 7.9.5, care permite navigarea intre pagini fara reincarcarea completa a aplicatiei.

1.3 Functionalitatile Aplicatiei

Zona Publica

Aplicatia ofera o serie de functionalitati accesibile tuturor vizitatorilor, fara necesitatea autentificarii. Homepage-ul prezinta o interfata atractiva cu gradient colorat, afisand statistici in timp real despre numarul total de episoade, numarul de sezoane disponibile, si cel mai recent sezon adaugat. Aceste statistici sunt calculate dinamic prin interogarea bazei de date la fiecare incarcare a paginii, asigurand ca informatiile afisate sunt intotdeauna actualizate.

Pagina principala pentru explorarea episoadelor este accesibila la calea `/episodes` si ofera o experienta completa de cautare si filtrare. Episoadele sunt afisate intr-un grid responsive care se adapteaza automat la dimensiunea ecranului, aratand o singura coloana pe dispozitive mobile, doua coloane pe tablete, si trei coloane pe ecrane desktop. Fiecare card

de episod contine imaginea episodului, titlul, numarul sezonului si episodului, data de difuzare, si o descriere scurta truncata la trei linii pentru a mentine un aspect uniform.

Functionalitatea de cautare este implementata client-side si ofera rezultate instantanee pe masura ce utilizatorul introduce text in bara de cautare. Cautarea se realizeaza case-insensitive prin campul name al episoadelor, permitand gasirea rapida a episoadelor dorite. Complementar cautarii, aplicatia ofera doua dropdownuri de filtrare: unul pentru selectarea sezonului si unul pentru selectarea anului de difuzare. Aceste filtre pot fi folosite individual sau in combinatie pentru a rafina rezultatele.

Paginarea este implementata client-side cu optiuni configurabile pentru numarul de elemente pe pagina. Utilizatorii pot alege sa vizualizeze 10, 25 sau 50 de episoade per pagina, iar interfata ofera butoane Previous si Next impreuna cu numerotarea paginilor pentru navigare usoara. Un contor de rezultate afiseaza cate episoade sunt vizualizate din totalul rezultatelor filtrate, oferind feedback clar utilizatorului despre pozitia sa in lista.

Fiecare episod poate fi vizualizat in detaliu accesand pagina dedicata la calea `/episodes/:id`. Aceasta pagina afiseaza imaginea episodului la dimensiune mare, titlul complet, metadata organizata in carduri separate pentru sezon, numar episod si data de difuzare, si descrierea completa a episodului. Un link catre pagina South Park Wiki este furnizat pentru utilizatorii care doresc informatii suplimentare, deschis intr-un tab nou pentru a pastra aplicatia activa. Butoane de navigare permit revenirea rapida la homepage sau la lista de episoade.

O functionalitate suplimentara este butonul Random Episode de pe homepage, care selecteaza aleator un episod din baza de date si redirectioneaza utilizatorul catre pagina de detalii a acestuia. Aceasta functionalitate este implementata prin generarea unui index aleator si foloseste functia navigate din React Router pentru redirectionare.

Zona Administrativa

Zona administrativa este protejata prin autentificare JWT si ofera functionalitati complete de gestiune a continutului. Accesul se face prin pagina de login la calea `/login`, unde utilizatorul introduce username si parola. La submiterea formularului, credentialele sunt trimise catre backend care le verifica cu valorile stocate in fisierul de environment variables. Daca autentificarea este reusita, server-ul genereaza un token JWT cu o valabilitate de 24 de ore si il returneaza impreuna cu informatiile utilizatorului.

Token-ul primit este salvat in localStorage pentru persistenta intre sesiuni, permitand utilizatorului sa ramana autentificat chiar daca inchide tab-ul browserului. Context-ul de autentificare, implementat folosind React Context API, stocheaza informatiile utilizatorului si token-ul in state-ul global al aplicatiei, facandu-le accesibile din orice componenta fara a fi nevoie de prop drilling.

Dashboard-ul administrativ, disponibil la calea `/admin`, afiseaza un mesaj de bun venit cu username-ul utilizatorului autentificat si un buton de logout vizibil colorat in rosu pentru claritate vizuala. Un buton verde prominent permite adaugarea unui episod nou, redirectionand catre formularul de creare. Tabelul principal afiseaza toate episoadele din

baza de date cu coloane pentru ID, nume, sezon, numar episod, data de difuzare, si actiuni disponibile. Fiecare rand contine doua butoane: unul albastru pentru editare si unul rosu pentru stergere.

Formularul de adaugare episod, accesibil la `/admin/episodes/new`, implementeaza toate campurile necesare pentru crearea unui episod complet. Campurile name, season si episode sunt marcate ca obligatorii si sunt validate atat pe client cat si pe server. Campurile optionale includ air_date cu un date picker pentru selectie usoara, description ca textarea pentru text mai lung, wiki_url pentru linkul catre pagina Wiki, si image pentru imaginea episodului.

Upload-ul imaginii este implementat folosind FileReader API din JavaScript pentru conversia fisierului selectat in format Base64. Procesul incepe cu validarea tipului de fisier, acceptand doar imagini, urmat de verificarea dimensiunii pentru a nu depasi limita de 5MB. Daca validarile trec, fisierul este citit ca Data URI si rezultatul este salvat in state-ul formularului. Un preview al imaginii este afisat imediat pentru feedback vizual, permitand utilizatorului sa verifice imaginea inainte de submiterea formularului.

La submiterea formularului de adaugare, toate datele sunt trimise ca JSON catre endpoint-ul POST `/api/episodes`, incluzand token-ul JWT in headerul Authorization cu formatul Bearer. Backend-ul verifica token-ul, valideaza datele, si insereaza noul episod in baza de date folosind JSON.stringify pentru conversia obiectului JavaScript in format JSON storage. Dupa crearea cu succes, utilizatorul este redirectionat automat catre dashboard-ul administrativ unde poate vedea episodul nou adaugat in tabel.

Formularul de editare, disponibil la `/admin/episodes/:id/edit`, este similar cu formularul de adaugare dar cu functionalitatea suplimentara de pre-populare a campurilor cu datele existente. La incarcarea paginii, un useEffect hook face un fetch al episodului specific folosind ID-ul din URL si populeaza state-ul formularului cu valorile primite. Imaginea existenta este afisata ca preview, si utilizatorul poate alege sa o pastreze sau sa incarce una noua. Submit-ul formularului trimite un request PUT catre backend cu datele actualizate.

Stergerea unui episod este implementata cu un dialog de confirmare pentru a preveni stergeri accidentale. Cand utilizatorul da click pe butonul Delete din tabel, un window.confirm afiseaza mesajul “Are you sure you want to delete” urmat de numele episodului. Daca utilizatorul confirma, un request DELETE este trimis catre backend cu token-ul JWT. Dupa stergerea cu succes, lista de episoade este reincarcata pentru a reflecta modificarea.

Protectia Rutelor

Toate rutele administrative sunt protejate folosind un wrapper component ProtectedRoute care verifica daca utilizatorul este autentificat inainte de a permite accesul la continut. Implementarea foloseste hook-ul useAuth pentru a accesa functia isAuthenticated din context. Daca utilizatorul nu este autentificat, componenta redirectioneaza automat catre pagina de login folosind Navigate din React Router, salvand locatia curenta in state pentru a putea redirectiona utilizatorul inapoi dupa login.

Hook-ul useLocation permite salvarea paginii de unde a venit utilizatorul, astfel incat dupa autentificare sa fie redirectionat inapoi la pagina initiala. Acest pattern este implementat in pagina de login prin accesarea location.state?.from?.pathname si folosirea acestei valori ca destinatie pentru navigate dupa login reusit, cu fallback la '/admin' daca nu exista o pagina anterioara.

1.4 Flow-ul de Date

Comunicarea dintre frontend si backend se realizeaza exclusiv prin intermediul API-ului RESTful implementat pe server. Frontend-ul face cereri HTTP folosind libraria Axios, care este configurata cu un timeout de 10 secunde si un base URL care poate fi customizat prin environment variables. Toate cererile si raspunsurile folosesc formatul JSON pentru transferul datelor.

Pentru operatiuni de citire, frontend-ul face cereri GET catre `/api/episodes` pentru lista completa sau `/api/episodes/:id` pentru un episod specific. Backend-ul interogheaza baza de date MySQL folosind libraria mysql2 cu promise wrapper pentru async/await. Raspunsul contine datele episodului cu ID-ul si toate campurile din coloana JSON parsate automat de mysql2.

Operatiunile de scriere necesita autentificare si includ token-ul JWT in headerul Authorization cu prefixul Bearer. Middleware-ul authenticateToken din backend extrage token-ul din header, il verifica folosind functia jwt.verify cu secret-ul din environment variables, si salveaza informatiile decodate in req.user. Daca token-ul este invalid sau expirat, middleware-ul returneaza un raspuns 403 Forbidden si opreste procesarea cererii.

Pentru crearea unui episod nou, frontend-ul trimite un POST cu toate datele episodului incluzand imaginea in format Base64. Backend-ul valideaza ca campurile obligatorii sunt prezente si insereaza datele in baza de date folosind JSON.stringify explicit pentru conversia obiectului JavaScript in format JSON storage. MySQL genereaza automat un ID folosind AUTO_INCREMENT si il returneaza in raspuns.

Actualizarea unui episod existent se face prin PUT la `/api/episodes/:id` cu toate datele actualizate. Backend-ul verifica mai intai daca episodul exista prin SELECT, apoi face UPDATE cu datele noi folosind tot JSON.stringify. Stergerea se realizeaza prin DELETE la acelasi path, cu verificare similara a existentei si returnarea unui mesaj de eroare 404 daca episodul nu este gasit.

---

2. TEHNOLOGIILE UTILIZATE

2.1 Stack Tehnologic Backend

Backend-ul aplicatiei este construit pe platforma Node.js folosind Express.js ca framework web principal. Node.js versiunea 16 sau mai recenta ofera runtime-ul JavaScript necesar pentru executarea codului server-side, permitand folosirea aceluiasi limbaj de programare

pe frontend si backend. Aceasta unificare simplifica dezvoltarea si permite share-uirea de cod intre cele doua componente acolo unde este posibil.

Express.js versiunea 5.1.0 este framework-ul web ales pentru implementarea server-ului HTTP si definirea rutelor API. Express ofera un sistem de middleware puternic care permite procesarea secventiala a cererilor prin functii middleware care pot modifica request-ul, response-ul, sau pot opri procesarea. In aplicatia noastra, Express este configurat sa ruleze pe portul 5000 definit in environment variables cu fallback la aceasta valoare daca variabila nu este setata.

Prima linie de middleware configurata este cors din pachetul cu acelasi nume versiunea 2.8.5. Acest middleware rezolva problema Cross-Origin Resource Sharing care apare cand frontend-ul rulat pe localhost:5173 incearca sa faca cereri catre backend-ul de pe localhost:5000. Fara cors, browser-ul ar bloca aceste cereri pentru securitate. Middleware-ul este adaugat cu app.use(cors()) fara configurare suplimentara, permitand cereri de pe orice origin pentru simplitate in development.

Al doilea middleware important este body-parser versiunea 2.2.0 care parsheaza body-ul cererilor HTTP. Sunt configurate doua parsere: unul pentru JSON cu app.use(bodyParser.json({ limit: '50mb' })) si unul pentru URL-encoded data cu app.use(bodyParser.urlencoded({ extended: true, limit: '50mb' })). Limita de 50MB este necesara pentru a permite upload-ul imaginilor in format Base64 care sunt semnificativ mai mari decat formatul binar.

Baza de date folosita este MySQL versiunea 8.0 care ofera suport nativ pentru tipul de date JSON incepand cu versiunea 5.7. Acest suport permite stocarea unor structuri de date flexibile fara a fi nevoie de definirea unui schema rigid cu coloane separate pentru fiecare camp. In fisierul db.js este creata o conexiune pool folosind mysql2 versiunea 3.15.3, o versiune imbunatatita a driver-ului mysql care ofera suport nativ pentru Promises si async/await.

Codul de configurare a pool-ului este urmatorul:

```
const pool = mysql.createPool({
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USER || 'root',
  password: process.env.DB_PASSWORD || '',
  database: process.env.DB_NAME || 'south_park_db',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

const promisePool = pool.promise();
module.exports = promisePool;
```

Connection pool-ul mentine un set de maxim 10 conexiuni active catre MySQL care sunt reutilizate pentru multiple query-uri. Acest pattern imbunatateste semnificativ performanta comparativ cu crearea unei conexiuni noi pentru fiecare cerere. Promise

wrapper-ul permite folosirea async/await in loc de callbacks, rezultand cod mai curat si mai usor de citit.

Autentificarea este implementata folosind JSON Web Tokens prin libraria jsonwebtoken versiunea 9.0.2. JWT este un standard pentru crearea de tokene de acces care contin informatii despre utilizator si sunt semnate criptografic. Structura unui token JWT consta din trei parti separate prin punct: header, payload si signature. Header-ul contine tipul token-ului si algoritmul de semnare, payload-ul contine datele despre utilizator si metadata precum issued at si expiration, iar signature-ul este calculat folosind un secret key pentru a preveni falsificarea.

In fisierul server.js, endpoint-ul de login este implementat astfel:

```
app.post('/api/auth/login', (req, res) => {
  const { username, password } = req.body;

  if (username === process.env.ADMIN_USERNAME && password === process.env.ADM
IN_PASSWORD) {
    const token = jwt.sign(
      { username: username, role: 'admin' },
      process.env.JWT_SECRET,
      { expiresIn: '24h' }
    );
    return res.json({
      message: 'Login successful',
      token: token,
      user: { username, role: 'admin' }
    });
  }

  return res.status(401).json({ error: 'Invalid credentials' });
});
```

Credentialele admin sunt stocate in fisierul .env si nu intr-o baza de date pentru simplitate, avand in vedere ca aplicatia este un proiect educational cu un singur utilizator. In productie, acest approach ar trebui inlocuit cu o tabela de utilizatori si parole hash-uite folosind bcrypt sau argon2.

Middleware-ul de autentificare este implementat intr-un fisier separat authMiddleware.js si arata astfel:

```
const authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) {
    return res.status(401).json({ error: 'Access denied. No token provided.'
});
  }
```

```
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(403).json({ error: 'Invalid or expired token.' });
  }
};
```

Middleware-ul extrage token-ul din headerul Authorization care trebuie sa aiba formatul "Bearer TOKEN". Daca token-ul lipseste, raspunde cu status 401 Unauthorized. Daca token-ul este prezent dar invalid sau expirat, jwt.verify arunca o exceptie care este prinsa si rezulta intr-un raspuns 403 Forbidden. Daca verificarea reuseste, payload-ul decodat este salvat in req.user si executia continua cu next().

Pachetul dotenv versiunea 17.2.3 este folosit pentru a incarca variabilele de mediu din fisierul .env la pornirea aplicatiei. Prima linie din server.js este require('dotenv').config() care citeste fisierul si face variabilele disponibile in process.env. Fisierul .env.example contine un template cu toate variabilele necesare:

```
ADMIN_USERNAME=admin
ADMIN_PASSWORD=admin123
JWT_SECRET=south_park_secret_key_2025_super_secure
PORT=5000
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=
DB_NAME=south_park_db
```

2.2 Stack Tehnologic Frontend

Frontend-ul este dezvoltat cu React versiunea 19.1.1, cea mai recenta versiune a librariei JavaScript pentru construirea interfetelor utilizator. React a fost ales pentru popularitatea sa in industrie, ecosistemul vast de librarii si tooling, si suportul pentru component-based architecture care permite reutilizarea codului si mentinerea mai usoara.

Vite versiunea 7.1.7 este build tool-ul ales in loc de Create React App traditional. Vite ofera o experienta de dezvoltare semnificativ mai rapida datorita folosirii ES modules native in browser si Hot Module Replacement aproape instantaneu. Dev server-ul porneste in cateva sute de milisecunde comparativ cu zeci de secunde in CRA, iar modificarile in cod sunt reflectate in browser in sub 50 de milisecunde. Configurarea Vite este minimala in fisierul vite.config.js:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
})
```

Styling-ul aplicatiei este implementat folosind Tailwind CSS versiunea 3.4.18, un framework utility-first care ofera clase pre-definite pentru aproape orice proprietate CSS. Spre deosebire de frameworks ca Bootstrap care ofera componente pre-construite, Tailwind ofera building blocks pentru a construi propriul design. Configurarea Tailwind in tailwind.config.js specifica ce fisiere sa scaneze pentru clase:

```
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Property-ul content spune Tailwind sa scaneze toate fisierele JavaScript si JSX din directorul src pentru a detecta ce clase sunt folosite. La build pentru productie, Tailwind elimina automat toate clasele neutilizate rezultand un fisier CSS final foarte mic, de obicei sub 15KB.

React Router DOM versiunea 7.9.5 gestioneaza rutarea client-side permitand navigarea intre pagini fara reincarcarea completa a aplicatiei. Configurarea rutelor in App.jsx defineste toate path-urile disponibile:

```
<Router>
  <AuthProvider>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/episodes" element={<EpisodeList />} />
      <Route path="/episodes/:id" element={<EpisodeDetail />} />
      <Route path="/login" element={<Login />} />
      <Route path="/admin" element={<ProtectedRoute><AdminDashboard /></Prote
ctedRoute>} />
      <Route path="/admin/episodes/new" element={<ProtectedRoute><AddEpisode
/></ProtectedRoute>} />
      <Route path="/admin/episodes/:id/edit" element={<ProtectedRoute><EditEp
isode /></ProtectedRoute>} />
    </Routes>
  </AuthProvider>
</Router>
```

Rutele administrative sunt invelite in componenta ProtectedRoute care verifica autentificarea inainte de a permite accesul. AuthProvider este un wrapper al Context API care face state-ul de autentificare disponibil in toata aplicatia.

Axios versiunea 1.13.2 este libraria HTTP client folosita pentru toate cererile catre backend. In fisierul services/api.js este creata o instanta Axios cu configurare de baza:

```
const api = axios.create({
  baseURL: `${API_URL}/api`,
  timeout: 10000,
  headers: {
    'Content-Type': 'application/json'
  }
});

export const episodesAPI = {
  getAll: () => api.get('/episodes'),
  getById: (id) => api.get(`/episodes/${id}`),
  create: (episodeData) => api.post('/episodes', episodeData),
  update: (id, episodeData) => api.put(`/episodes/${id}`, episodeData),
  delete: (id) => api.delete(`/episodes/${id}`)
};
```

API_URL este definit in config.js si poate fi customizat prin environment variable VITE_API_URL, cu fallback la http://localhost:5000 pentru development. Timeout-ul de 10 secunde previne blocarea aplicatiei in cazul in care server-ul nu raspunde.

State management-ul global pentru autentificare este implementat folosind React Context API in fisierul AuthContext.jsx. Context-ul stocheaza informatiile utilizatorului, token-ul JWT, si ofera functii pentru login si logout:

```
const [user, setUser] = useState(null);
const [token, setToken] = useState(null);
const [loading, setLoading] = useState(true);

useEffect(() => {
  const savedToken = localStorage.getItem('token');
  const savedUser = localStorage.getItem('user');

  if (savedToken && savedUser) {
    setToken(savedToken);
    setUser(JSON.parse(savedUser));
  }
  setLoading(false);
}, []);

const login = async (username, password) => {
  try {
    const response = await axios.post(`${API_URL}/api/auth/login`, {
      username,
      password
    });

    const { token: newToken, user: newUser } = response.data;

    setToken(newToken);
    setUser(newUser);
```

```
    localStorage.setItem('token', newToken);
    localStorage.setItem('user', JSON.stringify(newUser));

    return { success: true };
  } catch (error) {
    return {
      success: false,
      error: error.response?.data?.error || 'Login failed. Please try again.'
    };
  }
};
```

La montarea aplicatiei, un useEffect hook verifica daca exista un token salvat in localStorage si il incarca in state. Aceasta permite utilizatorului sa ramana autentificat chiar daca inchide tab-ul. Functia login face cererea de autentificare si salveaza token-ul si user-ul atat in state cat si in localStorage.

Custom hook-ul useAuth permite accesarea usoara a context-ului din orice componenta:

```
export const useAuth = () => {
  const context = useContext(AuthContext);
  if (!context) {
    throw new Error('useAuth must be used within AuthProvider');
  }
  return context;
};
```

2.3 Dependinte Complete

Package.json-ul backend-ului listeaza urmatoarele dependinte:

```
{
  "type": "commonjs",
  "dependencies": {
    "body-parser": "^2.2.0",
    "cors": "^2.8.5",
    "dotenv": "^17.2.3",
    "express": "^5.1.0",
    "jsonwebtoken": "^9.0.2",
    "mysql2": "^3.15.3"
  }
}
```

Package.json-ul frontend-ului contine:

```
{
  "type": "module",
  "dependencies": {
    "axios": "^1.13.2",
    "react": "^19.1.1",
    "react-dom": "^19.1.1",
```

```
    "react-router-dom": "^7.9.5"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^5.0.4",
    "autoprefixer": "^10.4.21",
    "postcss": "^8.5.6",
    "tailwindcss": "^3.4.18",
    "vite": "^7.1.7"
  }
}
```

Diferenta dintre “dependencies” si “devDependencies” este ca devDependencies sunt necesare doar pentru development si build, dar nu sunt incluse in bundle-ul final de productie.

---

3. STRUCTURA DATELOR

3.1 Schema Bazei de Date

Baza de date MySQL foloseste o structura simpla cu un singur tabel numit data. Schema este definita in fisierul seed.sql astfel:

```
CREATE TABLE IF NOT EXISTS `data` (
  id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  data JSON
);
```

Structura este intentionat simpla pentru a maximiza flexibilitatea. In loc de a avea coloane separate pentru fiecare camp al unui episod, toate informatiile sunt stocate in coloana data de tip JSON. Aceasta abordare are avantaje si dezavantaje.

Avantajele includ flexibilitatea de a adauga campuri noi fara a fi nevoie de ALTER TABLE si migrari de schema. Daca dorim sa adaugam un camp nou precum rating sau views, putem pur si simplu sa il includem in obiectul JSON al episoadelor noi fara a afecta episoadele existente. Schema-less design-ul este ideal pentru aplicatii in stadiul initial de dezvoltare unde cerințele se pot schimba frecvent.

Dezavantajele includ performanta queries-urilor complexe care poate fi inferioara comparativ cu coloane indexate traditional. De asemenea, nu putem defini foreign keys pentru a crea relatii intre tabele, si validarea datelor trebuie facuta la nivel de aplicatie in loc sa se bazeze pe constrangeri de baza de date.

Coloana id este de tip INT UNSIGNED PRIMARY KEY AUTO_INCREMENT ceea ce inseamna ca MySQL genereaza automat un identificator unic crescator pentru fiecare rand inserat. Acest ID este folosit in URL-urile aplicatiei pentru a identifica episoade specific, de exemplu /episodes/1 sau /api/episodes/5.

3.2 Structura Obiectului JSON

Fiecare episod este stocat ca un obiect JSON cu urmatoarea structura:


```
{
  "name": "HUMANCENTiPAD",
  "season": 15,
  "episode": 1,
  "air_date": "2011-04-27",
  "description": "Kyle is intimately involved in the development of a revolut
ionary new product that is about to be launched by Apple. Meanwhile, Cartman
doesn't even have a regular iPad yet. He blames his mother.",
  "wiki_url": "https://southpark.fandom.com/wiki/HUMANCENTiPAD",
  "image": ""
}
```


Campul name este un string care contine titlul episodului si este obligatoriu. Titlurile pot contine litere, numere, spatii si caractere speciale. Validarea pe backend verifica ca acest camp nu este gol inainte de a permite crearea sau actualizarea unui episod.

Campul season este un numar care indica sezonul din care face parte episodul si este de asemenea obligatoriu. South Park are in prezent 27 de sezoane, astfel incat validarea ar trebui sa permita valori intre 1 si 27, desi aplicatia nu impune momentan aceasta constrangere specific.

Campul episode reprezinta numarul episodului in cadrul sezonului si este obligatoriu. Majoritatea sezoanelor au intre 10 si 14 episoade, cu cateva exceptii care au mai multe sau mai putine.

Campul air_date este un string in format ISO 8601 YYYY-MM-DD care reprezinta data de difuzare a episodului si este optional. In frontend, acest string este convertit la un obiect Date JavaScript si formatat folosind toLocaleDateString pentru afisare intr-un format mai citibil. De exemplu, “2011-04-27” devine “April 27, 2011”.

Campul description contine o descriere text a episodului si este optional. In lista de episoade, descrierea este truncata la 3 linii folosind clasa Tailwind line-clamp-3, in timp ce pe pagina de detalii este afisata integral.

Campul wiki_url contine un link catre pagina South Park Fandom Wiki a episodului si este optional. In interfata, acest link este afisat ca un buton care deschide pagina intr-un tab nou folosind target=“_blank” si rel=“noopener noreferrer” pentru securitate.

Campul image este un string care contine imaginea episodului encodata in format Base64 ca Data URI si este optional. Formatul complet este “” urmat de string-ul Base64 propriu-zis. Daca campul este gol sau lipseste, aplicatia afiseaza un placeholder sau pur si simplu nu afiseaza imagine.

3.3 Procesarea Imaginilor Base64

Upload-ul imaginilor este implementat in componenta AddEpisode.jsx folosind FileReader API din JavaScript. Procesul incepe cand utilizatorul selecteaza un fisier din input-ul de tip file. Functia handleImageChange este apelata cu event-ul care contine fisierul selectat:

```
const handleImageChange = (e) => {
  const file = e.target.files[0];
  if (!file) return;

  if (!file.type.startsWith('image/')) {
    setError('Please select a valid image file');
    return;
  }

  if (file.size > 5 * 1024 * 1024) {
    setError('Image size must be less than 5MB');
    return;
  }

  const reader = new FileReader();
  reader.onloadend = () => {
    setFormData(prev => ({
      ...prev,
      image: reader.result
    }));
  };
  reader.readAsDataURL(file);
};
```

Prima validare verifica ca fisierul selectat este efectiv o imagine verificand ca MIME type-ul incepe cu “image/”. Aceasta accepta JPEG, PNG, GIF, WebP si alte formate standard de imagine. Daca fisierul nu este o imagine, se afiseaza un mesaj de eroare si procesarea se opreste.

A doua validare verifica dimensiunea fisierului sa nu depaseasca 5 megabytes. Aceasta limita este necesara deoarece imaginile Base64 sunt cu aproximativ 33% mai mari decat echivalentul binar, si o imagine de 5MB devine aproximativ 6.6MB in Base64. Backend-ul are setat un limit de 50MB in body-parser pentru a acomoda aceste dimensiuni.

Daca ambele validari trec, este creat un obiect FileReader care citeste continutul fisierului. Metoda readAsDataURL converteste fisierul intr-un Data URI care poate fi folosit direct ca src pentru un element img. Cand citirea este completa, event handler-ul onloadend este apelat si rezultatul este salvat in state-ul formularului.

Rezultatul este un string care arata astfel pentru un JPEG: “…” unde partea dupa virgula este continutul fisierului encodat in Base64. Acest string este salvat direct in campul image al obiectului episod si trimis ca parte a payload-ului JSON la submiterea formularului.

Pe backend, string-ul Base64 este stocat exact asa cum este primit in coloana JSON din MySQL. Nu este nevoie de procesare suplimentara deoarece MySQL stocheaza JSON-ul ca text. La citire, mysql2 parsheaza automat JSON-ul inapoi in obiect JavaScript, iar frontend-ul poate folosi string-ul direct ca src pentru img tags.

Avantajul acestei abordari este simplitatea. Toate datele episodului inclusiv imaginea sunt in baza de date, facand backup-ul si restore-ul triviale. Dezavantajul este ca payload-urile HTTP sunt mari, impactand performanta pentru conexiuni lente, si MySQL are limit-uri pe dimensiunea coloanelor JSON care variaza in functie de configuratie.

3.4 Operatiuni pe Baza de Date

In fisierul server.js, toate operatiunile de baza de date folosesc promise pool-ul creat in db.js. Query-urile sunt executate folosind async/await pentru cod mai curat comparativ cu callbacks.

Operatiunea SELECT pentru toate episoadele arata astfel:

```
app.get('/api/episodes', async (req, res) => {
  try {
    const [rows] = await db.query('SELECT * FROM data');
    const episodes = rows.map(row => ({
      id: row.id,
      ...row.data
    }));
    res.json(episodes);
  } catch (error) {
    console.error('Error fetching episodes:', error);
    res.status(500).json({ error: 'Failed to fetch episodes' });
  }
});
```

Metoda db.query returneaza un array cu doua elemente: rows si fields. Destructurarea [rows] extrage doar primul element care contine randurile returnate. Fiecare rand are proprietatea id care este un numar si proprietatea data care este un obiect JavaScript (mysql2 a parshat automat JSON-ul). Spread operator-ul ...row.data combina id-ul cu toate proprietatile din data intr-un singur obiect plat.

SELECT pentru un episod specific foloseste un prepared statement pentru protectie impotriva SQL injection:

```
app.get('/api/episodes/:id', async (req, res) => {
  try {
    const { id } = req.params;
    const [rows] = await db.query('SELECT * FROM data WHERE id = ?', [id]);

    if (rows.length === 0) {
      return res.status(404).json({ error: 'Episode not found' });
    }
```

```
    const episode = {
      id: rows[0].id,
      ...rows[0].data
    };
    res.json(episode);
  } catch (error) {
    console.error('Error fetching episode:', error);
    res.status(500).json({ error: 'Failed to fetch episode' });
  }
});
```

Placeholder-ul ? in query este inlocuit cu valoarea din array-ul [id], dar mysql2 escape-uieste automat valoarea pentru a preveni SQL injection. Daca query-ul nu returneaza randuri, inseamna ca ID-ul nu exista si se returneaza status 404.

Operatiunea INSERT pentru crearea unui episod nou necesita JSON.stringify explicit:

```
app.post('/api/episodes', authenticateToken, async (req, res) => {
  try {
    const episodeData = req.body;

    if (!episodeData.name || !episodeData.season || !episodeData.episode) {
      return res.status(400).json({
        error: 'Name, season, and episode are required'
      });
    }

    const [result] = await db.query(
      'INSERT INTO data (data) VALUES (?)',
      [JSON.stringify(episodeData)]
    );

    res.status(201).json({
      message: 'Episode created successfully',
      id: result.insertId
    });
  } catch (error) {
    console.error('Error creating episode:', error);
    res.status(500).json({ error: 'Failed to create episode' });
  }
});
```

Middleware-ul authenticateToken asigura ca doar utilizatorii autentificati pot crea episoade. Validarea verifica ca cele trei campuri obligatorii sunt prezente. JSON.stringify converteste obiectul JavaScript in string JSON pentru storage. Property-ul result.insertId contine ID-ul generat de AUTO_INCREMENT pentru noul rand.

UPDATE este similar cu INSERT dar verifica mai intai existenta episodului:

```
app.put('/api/episodes/:id', authenticateToken, async (req, res) => {
  try {
    const { id } = req.params;
    const episodeData = req.body;

    const [existing] = await db.query('SELECT * FROM data WHERE id = ?', [id]
);
    if (existing.length === 0) {
      return res.status(404).json({ error: 'Episode not found' });
    }

    await db.query(
      'UPDATE data SET data = ? WHERE id = ?',
      [JSON.stringify(episodeData), id]
    );

    res.json({ message: 'Episode updated successfully' });
  } catch (error) {
    console.error('Error updating episode:', error);
    res.status(500).json({ error: 'Failed to update episode' });
  }
});
```

DELETE opreste executia si returneaza 404 daca episodul nu exista:

```
app.delete('/api/episodes/:id', authenticateToken, async (req, res) => {
  try {
    const { id } = req.params;

    const [result] = await db.query('DELETE FROM data WHERE id = ?', [id]);

    if (result.affectedRows === 0) {
      return res.status(404).json({ error: 'Episode not found' });
    }

    res.json({ message: 'Episode deleted successfully' });
  } catch (error) {
    console.error('Error deleting episode:', error);
    res.status(500).json({ error: 'Failed to delete episode' });
  }
});
```

Property-ul result.affectedRows indica cate randuri au fost afectate de query. Daca este 0, inseamna ca ID-ul nu exista in baza de date.

---

4. UTILIZAREA INTELIGENTEI ARTIFICIALE

4.1 Instrumentul AI Utilizat

Pentru dezvoltarea acestui proiect am utilizat Claude Code de la Anthropic, un asistent AI specializat in programare si dezvoltare web. Claude Code este un tool CLI oficial care permite colaborarea directa cu modelele Claude in contextul dezvoltarii software. Am folosit predominant modelul Claude Sonnet versiunea 4.5 pentru majoritatea task-urilor de cod si Claude Opus 4.5 pentru task-uri mai complexe care necesitau reasoning mai profund.

4.2 Procesul de Dezvoltare cu AI

AI-ul a avut un rol de suport in doua arii principale ale dezvoltarii: consultanta pentru arhitectura proiectului si generarea scheletului initial de cod. La inceputul proiectului, am consultat AI-ul pentru a primi recomandari privind stack-ul tehnologic potrivit pentru cerinte. Claude a sugerat folosirea Node.js cu Express pentru backend si React cu Vite pentru frontend, impreuna cu MySQL pentru baza de date, oferind o perspectiva generala asupra modului in care aceste tehnologii se integreaza.

Pentru arhitectura generala, AI-ul a oferit sugestii privind separarea aplicatiei in frontend si backend, organizarea fisierelor in directoare logice, si folosirea JWT pentru sistemul de autentificare. Aceste recomandari initiale au servit ca punct de plecare pentru deciziile noastre arhitecturale.

Generarea codului initial a constat in crearea unor template-uri de baza pentru fisiere precum configuratia serverului Express, structura middleware-ului de autentificare, si scheletul componentelor React principale. AI-ul a oferit un starting point pe care l-am putut adapta si extinde conform nevoilor specifice ale proiectului.

4.3 Contributia echipei

Desi AI-ul a oferit un punct de plecare util, intreaga dezvoltare efectiva a aplicatiei a fost realizata de echipa noastra. Toate problemele tehnice care au aparut in timpul dezvoltarii au necesitat investigare manuala si rezolvare independenta. Debugging-ul erorilor, identificarea bug-urilor si gasirea solutiilor au fost in totalitate responsabilitatea noastra, folosind instrumente precum console.log, React DevTools si testare manuala extensiva.

Deciziile de design si experienta utilizatorului au fost facute in totalitate de echipa. Alegerea schemei de culori, layout-ul paginilor, structura navigatiei, alegerea Tailwind CSS pentru styling, si toate aspectele vizuale ale aplicatiei reprezinta munca noastra creativa. AI-ul nu poate lua decizii subiective despre estetica sau despre ce arata bine pentru utilizatori, acestea fiind contributii exclusiv umane.

Testing-ul complet al aplicatiei a fost realizat manual de echipa. Am testat fiecare functionalitate, fiecare pagina, fiecare formular si fiecare flow de utilizare pentru a ne asigura ca aplicatia functioneaza corect. Am verificat edge cases precum submiterea formularelor incomplete, accesarea paginilor protejate fara autentificare, comportamentul

aplicatiei la refresh, si multe alte scenarii. Aceste teste au identificat si corectat numeroase probleme care nu puteau fi detectate altfel.

Refactoring-ul si optimizarea codului au fost de asemenea contributii ale echipei. Codul initial a fost revizuit, restructurat si imbunatatit pentru a fi mai curat, mai eficient si mai usor de intretinut. Am extras logica comuna in functii reutilizabile, am simplificat conditional rendering-ul complex, si am imbunatatit denumirile variabilelor pentru claritate. Error handling-ul comprehensiv si feedback-ul vizual pentru utilizatori au fost adaugate de noi acolo unde era necesar.

Intelegerea completa a codului a fost o prioritate. Tot codul care a ajuns in aplicatie a fost citit, analizat si inteles de echipa inainte de a fi integrat.

4.4 Concluzii despre utilizarea AI

AI-ul a fost un tool util care ne-a oferit un punct de plecare pentru dezvoltare, dar nu a inlocuit munca echipei. Scheletul initial de cod si sugestiile arhitecturale au reprezentat doar fundamentul pe care am construit intreaga aplicatie. Gandirea critica, rezolvarea problemelor, creativitatea in design, testarea manuala, debugging-ul si toate deciziile importante au ramas in totalitate responsabilitatea echipei.

In concluzie, AI-ul a servit ca un consultant initial care ne-a ajutat sa pornim mai rapid, dar dezvoltarea, finalizarea si calitatea aplicatiei sunt rezultatul direct al muncii noastre.

---

5. CONCLUZII

5.1 Rezumatul Proiectului

South Park Episodes Database este o aplicatie web full-stack completa care demonstreaza implementarea conceptelor fundamentale si avansate de dezvoltare web. Proiectul implementeaza o arhitectura client-server moderna cu separare clara intre frontend dezvoltat in React si backend dezvoltat in Node.js cu Express. Baza de date MySQL cu suport JSON ofera flexibilitate in stocarea datelor mentinand in acelasi timp simplitatea unui sistem relational traditional.

Aplicatia ofera functionalitati complete atat pentru vizitatori publici cat si pentru administratori autentificati. Zona publica permite explorarea episoadelor prin search, filtrare si paginare implementate client-side pentru performanta maxima. Zona administrativa protejata prin JWT authentication permite gestionarea completa a continutului prin operatiuni CRUD comprehensive.

Tehnologiile alese reprezinta o selectie moderna si relevanta pentru industria de web development. React versiunea 19.1.1 cu Vite pentru build oferă o experienta de dezvoltare rapida si cod optimizat pentru productie. Express versiunea 5.1.0 pe Node.js ofera un backend simplu dar puternic capabil sa serveasca un API RESTful complet. MySQL

versiunea 8.0 cu suport JSON combina fiabilitatea unei baze de date relational cu flexibilitatea unui document store.

5.2 Competente Dobandite

Realizarea acestui proiect a permis consolidarea unor competente esentiale de dezvoltare web full-stack, atat pe partea de backend, cat si pe partea de frontend.

Pe backend, a fost aprofundata construirea unui API REST cu Express, lucrul cu o baza de date MySQL si gestionarea datelor in format JSON. A devenit clara diferenta dintre operatiile de citire si scriere atunci cand sunt utilizate coloane JSON, precum si importanta folosirii prepared statements si a autentificarii JWT pentru securizarea rutelor administrative.

Pe frontend, proiectul a oferit experienta practica in utilizarea React intr-o aplicatie completa, incluzand hooks, rutare client-side si formulare cu validare. Gestionarea autentificarii prin Context API si implementarea rutelor protejate au fost intelese si aplicate intr-un context real.

Integrarea dintre frontend si backend a contribuit la o mai buna intelegere a fluxului de date intr-o aplicatie web moderna si a modului in care componentele comunica printr-un API.

5.3 Dificultati Intampinate si Solutii

Prima dificultate majora a fost problema de JSON parsing in MySQL care a cauzat erori la prima rulare. Eroarea “Unexpected token J in JSON at position 0” nu indica clar ca problema era double parsing. Investigatia cu console.log ne-a aratat ca row.data era deja un obiect JavaScript, nu un string JSON. Solutia a fost sa eliminam JSON.parse() la SELECT si sa folosim doar spread operator pentru a combina id-ul cu data. Invers, la INSERT si UPDATE am descoperit ca trebuie folosit explicit JSON.stringify() pentru ca mysql2 nu converteste automat obiectele JavaScript la scriere.

A doua dificultate a fost payload too large errors cand incarcam imagini. Initial backend-ul crasha cu “PayloadTooLargeError: request entity too large” pentru imagini mai mari de 100kb. Problema era ca body-parser are o limita default foarte mica. Am crescut limita la 50MB prin { limit: ‘50mb’ } in configurare, ceea ce a rezolvat crashurile. Pentru a preveni abuse, am adaugat validare client-side de 5MB maximum inainte ca imaginea sa fie convertita la Base64. Aceasta combinatie de validare client-side si limit server-side asigura o experienta buna pentru utilizator pastrând securitatea.

O problema a fost 404 errors pe refresh in development cand accesam o ruta React Router direct. De exemplu, navigarea de la homepage la /episodes/1 functiona perfect, dar refresh-ul pe /episodes/1 returna 404 Not Found. Initial am crezut ca e o problema de configurare Vite, dar am descoperit ca Vite versiunea 7 deja face historyApiFallback automat. Problema reala era ca browser-ul meu avea cached un 404 response de la o incercare anterioara. Clear cache a rezolvat complet problema, invatandu-ne importanta de a verifica cache-ul inainte de a cauta solutii complexe.

CORS errors au fost inevitabile cand am separat frontend si backend pe porturi diferite. Eroarea “Access to XMLHttpRequest blocked by CORS policy” aparea la fiecare cerere API. Am invatat ca browser-ul blocheaza cross-origin requests by default pentru securitate. Solutia simpla a fost sa adaugam middleware-ul cors() pe backend care adauga headerele Access-Control-Allow-Origin necesare. Pentru development am permis toate origins, dar in productie ar trebui restrictionat la domeniul frontend-ului.

5.4 Directii Viitoare

Aplicatia poate fi extinsa in mai multe directii, in functie de complexitatea dorita.

Un pas logic ar fi implementarea unui sistem complet de utilizatori, cu stocarea acestora in baza de date, in locul credentialelor hardcodate. Acest lucru ar permite existenta mai multor conturi si roluri diferite.

O alta imbunatatire importanta ar fi optimizarea imaginilor, prin redimensionare sau compresie inainte de salvare, pentru a reduce dimensiunea payload-urilor si impactul asupra performantei.

Functionalitati precum comentarii, rating-uri sau marcarea episoadelor ca favorite ar putea creste nivelul de interactiune al utilizatorilor. De asemenea, extinderea bazei de date cu personaje ar permite filtrarea episoadelor in functie de personaje recurente.

La nivel de interfata, implementarea unui dark mode sau a suportului pentru mai multe limbi ar aduce aplicatia mai aproape de standardele moderne.

5.5 Lectii Personale

Acest proiect a evidentiat diferenta clara dintre invatarea teoretica si aplicarea practica a cunostintelor.

Un aspect esential a fost importanta debugging-ului. Intelegerea erorilor prin analizarea mesajelor din consola si a stack trace-urilor s-a dovedit mai eficienta decat cautarea imediata a solutiilor externe.

Planificarea initiala a structurii aplicatiei s-a dovedit importanta pentru reducerea refactorizarilor ulterioare. De asemenea, testarea manuala a fost esentiala pentru identificarea unor probleme care nu sunt evidente din cod.

Utilizarea inteligentei artificiale a fost utila ca punct de plecare, insa proiectul a aratat clar ca intelegerea codului si luarea deciziilor raman responsabilitatea echipei.

In final, proiectul a oferit o baza solida pentru dezvoltari ulterioare si o intelegere realista a modului in care se construieste o aplicatie web completa.