1. Executive Summary

Project Name: DigitalBoneBox
System Type: Locally Deployed Educational Web Application (Node.js Middleware + GitHub Data Source)
Brief Overview of this Application: This is an open-source educational tool designed to provide students with interactive, high-fidelity anatomical resources. Architecturally, it functions as a locally run middleware service. The Node.js backend acts as a proxy, retrieving raw JSON and image data from a specific data branch in a public GitHub repository, caching it in-memory, and serving it to a local frontend interface.

Key Security Risks As a locally deployed, unauthenticated application that relies on an external service for data storage, our primary security risks are Availability which is about ensuring the app runs and Integrity that is, ensuring the medical data is correct.

- **Availability & Rate Limiting (Critical):** The application relies on unauthenticated requests to the GitHub Raw API to fetch data. GitHub imposes strict rate limits on unauthenticated traffic (60 requests/hour per IP). If the application's caching logic fails or if a student aggressively uses the search feature, they will hit this limit, causing the application to stop working immediately i.e. DoS.
- **Content Integrity & Unprotected Data Source (Critical):** The application treats the GitHub repository's data branch as its production database. Currently, there are no branch protection rules on the data branch. This allows direct, unmerged pushes to the data source without a Pull Request or code review. A compromised account or malicious contributor could immediately corrupt bone names, images, or descriptions. Since the application automatically fetches this data, these corruptions propagate instantly to users without any audit trail or approval process.
- **Input Validation & Injection (Medium):** The application accepts user input via the search bar and, critically, via URL parameters (e.g., /api/annotations/:boneId). While basic regex validation exists, these inputs are used to construct downstream URLs for data fetching. Vulnerabilities here could lead to unpredictable behavior or Client-Side Cross-Site Scripting (XSS) if error messages or reflected inputs are not strictly sanitized.

Security Priorities

- **Immediate Branch Protection:** Implement strict Branch Protection rules on the data branch to disable direct pushes and require Pull Request reviews for all data changes.
- **Resilient Data Fetching:** Optimize the caching strategy to minimize calls to GitHub and prevent rate-limit exhaustion.
- **Input Sanitization:** Ensure all user inputs (search queries and URL parameters) are strictly validated and sanitized before being processed or reflected back to the UI.

2. Product Context

2.1. Just as stated above in the project overview, in terms of design/ system architecture, the system functions as a locally deployed middleware service rather than a traditional cloud-hosted web app.

- **The Backend:** A Node.js/Express application that runs on the user's local machine. It acts as a proxy, retrieving raw JSON files and images from a specific branch called the “data” branch, of the project's public GitHub repository via the GitHub Raw API. It caches this data in-memory to serve the frontend efficiently.
- **The Frontend:** A client-side interface (HTML/CSS/JavaScript) that consumes the backend API to display bone images, overlay annotations, and conduct interactive quizzes.
- **The Data Source:** The "database" is a collection of static JSON files hosted on GitHub, organized by bone set (e.g., DataPelvis).

2.2. User Roles & Actors Since the application operates without an authentication system, roles are defined by their relationship to the code repository and the development lifecycle.

- **Tech Lead (Maintainer & Administrator):**
    - **Access Level:** Highest Privilege. The only actor authorized to merge Pull Requests (PRs) into the main branch.
    - **Responsibilities:** Acts as the central gatekeeper for security and quality. They review all code and data contributions, define the development roadmap, and manage repository settings.
    - **Security Context:** This is a critical "Trusted Insider" role. A compromised Tech Lead account allows an attacker to bypass all checks and merge malicious code directly into the stable codebase.
- **Developers & External Contributors:**
    - **Access Level:** Write Access to the repository.
    - **Behavior:** They implement features and submit data updates. They are expected to use Pull Requests for changes.
    - **Security Context:** Untrusted input source. However, due to current repository settings, they effectively have direct write access to the data branch (see Section 3).
- **Students (End Users):**
    - **Access Level:** Read-Only (Anonymous/Local).
    - **Behavior:** Run the application locally to browse bone sets, perform searches, and take quizzes.

- **Security Context:** They are the protected asset. We must ensure their local execution environment is not compromised by the application.

2.3. Critical Assets (What We Protect) Our protection goals focus on Integrity and Availability rather than Confidentiality.

- **Educational Integrity (Data Integrity):** The medical accuracy of bone names, descriptions, and annotation coordinates. If a malicious actor modifies the JSON to mislabel anatomical features, the tool fails its core educational mission.
- **Service Availability:** The ability of the application to fetch data without error. Reliance on the unauthenticated GitHub API creates a risk where rate-limiting renders the local application unusable for the student.
- **User Browser Security:** The safety of the student's local browser environment. We must prevent the application from executing malicious scripts (XSS) reflected from user input or corrupted data sources.

3. Attack Surface Analysis

This section maps out the specific "doors and windows" an attacker could try to open.

3.1. Entry Points (Where untrusted data enters)

- **Public Web Interface (Frontend Inputs):**
  - **Search Bar:** Accepts free-text input. This is a primary vector for Client-Side Cross-Site Scripting (XSS) if the application reflects this input back to the user.
  - **Quiz Interface:** User selections (and potentially text input in future versions) are processed by client-side logic.
- **API Endpoints (Backend Inputs):**
  - **URL Parameters (req.query & req.params):** The application reads the boneId parameter from the URL (e.g., /api/annotations/:boneId) and uses it to dynamically construct file paths for data fetching.
- **The Supply Chain (Data Source):**
  - **GitHub Repository (data branch):** Because the application automatically fetches whatever is in the data branch, the repository itself is an entry point. If an attacker can push a file named ilium.json that contains invalid JSON or malicious script tags, the application will ingest it.

3.2. Data Flow & Trust Boundaries

- **Boundary 1: Browser and Node.js Server:** The student's browser sends requests to localhost:8000. Data crossing this boundary is technically local, but still subject to browser-based attacks (XSS).
- **Boundary 2: Node.js Server and GitHub Raw API (The Internet):** The server sends HTTP GET requests to raw.githubusercontent.com. This crosses from the local machine to the public internet. We trust GitHub availability, but we cannot implicitly trust the integrity of the data content.
- **Boundary 3: Contributor and Repository (The Broken Boundary):** A contributor pushes code or data to GitHub.
    - **Code (main branch):** Protected. Must pass through the Tech Lead (Trust Boundary enforced).
    - **Data (data branch): Unprotected.** Contributors can push directly, bypassing the Tech Lead. This represents a collapse of the trust boundary.

4. Threat Catalog (STRIDE Analysis)

This section catalogs potential threats identified from the attack surface analysis, categorized using the STRIDE model.

S - Spoofing

- **Threat ID: S-1**
    - **Description:** Malicious Contributor Impersonation. A bad actor creates a GitHub account that mimics a trusted contributor to submit malicious data or code.
    - **Impact:** Medium. While they can submit PRs, the Tech Lead must still review them for the main branch. However, they could exploit the unprotected data branch to bypass review.
    - **Mitigation:** Enforce strict code review and require signed commits (future state).

T - Tampering (CRITICAL)

- **Threat ID: T-1**
    - **Description:** Direct Push to Data Branch (Data Corruption). Currently, the data branch lacks protection rules. Any contributor with write access can check out the data branch, corrupt JSON files (e.g., changing "Femur" to "Tibia"), or inject malicious scripts into description fields, and push directly to origin.
    - **Impact:** Critical. This bypasses the Pull Request and Code Review process entirely. The application will automatically fetch and display this corrupted data to all students immediately upon their next cache refresh.

    - **Mitigation:** Immediate: Enable "Branch Protection" for the data branch on GitHub to block direct pushes and require PR reviews.

- **Threat ID: T-2**
    - **Description:** Supply Chain Tampering (Dependencies). An attacker compromises an NPM package used in package.json.
    - **Impact:** High. Could lead to arbitrary code execution on the student's local machine when they run npm install / npm start.
    - **Mitigation:** Regularly run npm audit, lock file versions, and minimize external dependencies.

R - Repudiation

- **Threat ID: R-1**
    - **Description:** Unmerged Data Changes. If a user pushes directly to the data branch (Threat T-1), the change is recorded in Git history, but lacks the accountability of a Pull Request approval record. It becomes difficult to determine if a change was malicious or accidental, or who authorized it.
    - **Impact:** Low/Medium. Hinders auditability.
    - **Mitigation:** Branch protection ensures every change is associated with a Pull Request and a Tech Lead approval.

I - Information Disclosure

- **Threat ID: I-1**
    - **Description:** Error Stack Traces. The backend code currently logs full error details to the console and potentially returns stack traces to the client during 500 errors.
    - **Impact:** Low. Might reveal local file paths or logic to an attacker, aiding in crafting more complex attacks.
    - **Mitigation:** Ensure production error responses are generic ("Internal Server Error") and do not leak stack traces.

D - Denial of Service (CRITICAL)

- **Threat ID: D-1**
    - **Description:** GitHub API Rate Limit Exhaustion. The application makes unauthenticated requests to the GitHub Raw API. Unauthenticated requests are capped at 60 per hour per IP. If the application's caching logic fails, or if a student navigates aggressively, they will be banned by GitHub.

- **Impact:** Critical. The application stops working entirely for that user.
- **Mitigation:** Implement a robust Server-Side "Warm Cache" strategy that fetches all necessary data once at startup and serves all subsequent requests from memory.

E - Elevation of Privilege

- **Threat ID: E-1**
  - **Description:** XSS via Search or URL Parameters. An attacker sends a malicious link to a student: http://localhost:3000/boneset.html?boneId=<script>.... If the application reflects this parameter into the HTML without sanitization, the script executes in the student's browser.
  - **Impact:** Medium/High. Could execute arbitrary JavaScript in the context of the application.
  - **Mitigation:** Strict output encoding (escaping HTML entities) for all user-supplied data before rendering it in the DOM.

5. Risk Prioritization

In order to determine the order of remediation, ranking of identified threats is needed and done in this section. Priorities are assigned based on the likelihood of occurrence and the Severity of impact on the application's educational mission.

Priority 1: Critical (Immediate Action Required) These threats represent existential risks to the application's functionality or integrity. They must be addressed immediately.

- **T-1: Direct Push to Data Branch (Data Corruption)**
  - **Likelihood:** Certain. The vulnerability currently exists and requires no special skills to exploit. Any contributor can accidentally or maliciously push bad data.
  - **Impact:** Critical. It corrupts the "source of truth." Incorrect anatomical labels or descriptions directly undermine the educational value of the tool.
  - **Reasoning:** This is an open door. Until branch protection is enabled, the integrity of the entire project is unsecured.
- **D-1: GitHub API Rate Limit Exhaustion**
  - **Likelihood:** High. Unauthenticated requests are strictly capped by GitHub (60/hour). A single student testing the search feature aggressively or a bug in the caching logic will trigger this immediately.
  - **Impact:** Critical. The application becomes unusable (Denial of Service) for the user.

    - **Reasoning:** This is the most common failure mode for this architecture. Without a fix, the app is not reliable for study.

Priority 2: High (Address in Next Sprint) These threats allow for potential compromise of the user's machine or the development pipeline.

- **E-1: XSS via Search or URL Parameters**
    - **Likelihood:** Medium. Requires an attacker to craft a malicious link and trick a student into clicking it.
    - **Impact:** High. Could execute arbitrary scripts in the student's browser.
    - **Reasoning:** Input sanitization is a fundamental security requirement for any web application.
- **T-2: Supply Chain Tampering (Dependencies)**
    - **Likelihood:** Low/Medium. NPM ecosystem attacks are becoming more common.
    - **Impact:** High. Could lead to code execution on developer/student machines.
    - **Reasoning:** As the project grows, the dependency tree grows. Regular audits are necessary maintenance.

Priority 3: Medium/Low (Long-Term Planning) These threats are lower risk due to the specific context (local deployment) or difficulty of execution.

- **I-1: Information Disclosure (Error Stack Traces)**
    - **Likelihood:** High. Errors happen frequently during development.
    - **Impact:** Low. Since the app runs locally, the user "hacking" the app is hacking their own machine. However, leaking internal logic is bad practice.
    - **Reasoning:** Good to fix for professionalism and hardening, but not a blocker.
- **S-1: Contributor Impersonation**
    - **Likelihood:** Low. GitHub's own authentication controls make this difficult without account compromise.
    - **Impact:** Medium. Code review serves as a secondary defense layer.
    - **Reasoning:** We rely on GitHub's platform security for this; additional measures (like GPG signing) are "nice to have" for a student project.

6. Mitigation Strategy

6.1. Addressing T-1: Data Integrity & Direct Pushes

- **Threat:** Contributors pushing corrupted data directly to the data branch, bypassing review.
- **Status:** Planned (Immediate Priority)
- **Mitigation Plan:** Implement GitHub Branch Protection Rules for the data branch.
    - **Require Pull Requests:** No direct pushes allowed. All changes must come via PRs.
    - **Require Review:** At least one approval from Tech Lead as code owner is required before merging.
    - **Block Force Pushes:** Prevent history rewriting which could hide malicious changes.

6.2. Addressing D-1: Service Availability (Rate Limiting)

- **Threat:** Hitting GitHub's 60 req/hour rate limit due to inefficient fetching.
- **Status:** In Progress / Architecture Refactor
- **Mitigation Plan:** Transition from "Fetch-on-Request" to "Warm Cache" Architecture.
    - **Server Startup Fetch:** Instead of fetching data when a user clicks a bone, the server will fetch *all* necessary JSON structure files once upon startup (in initializeSearchCache).
    - **In-Memory Serving:** All subsequent search queries and navigation requests will be resolved against this local memory cache. This reduces GitHub API calls from *N* (per user action) to *1* (per server start), virtually eliminating the risk of rate-limiting during standard use.

6.3. Addressing E-1: Cross-Site Scripting (XSS)

- **Threat:** Injection of malicious scripts via Search or URL parameters.
- **Status:** Partially Implemented
- **Implemented Controls:**
    - **Input Validation:** The backend currently implements isValidBoneId() using regex to strictly limit URL parameters to alphanumeric characters. This effectively neutralizes path traversal and complex script injection in the API routes.
    - **Output Encoding:** The escapeHtml() helper function is implemented to sanitize search results before returning HTML fragments to the client.

- **Planned Enhancements:**
    - **Audit:** Review all frontend code to ensure innerHTML is not used with raw data.
    - **Content Security Policy (CSP):** Configure HTTP headers to restrict the sources from which scripts can load (e.g., only allow scripts from our own domain), providing a safety net even if XSS is attempted.

6.4. Addressing T-2: Supply Chain Risks

- **Threat:** Compromised NPM dependencies.
- **Status:** Ongoing Maintenance
- **Mitigation Plan:**
    - **Regular Audits:** Run npm audit before every major sprint release to identify known vulnerabilities.
    - **Lock Files:** Commit package-lock.json to the repository to ensure all developers are using the exact same version of dependencies, preventing "drift" where a new, malicious version of a library is accidentally installed.

7. Open Questions

1. **How will the security model change if we deploy to a public web server?** Currently, the application runs on localhost, where each student uses their own IP address to fetch data. If we deploy to a central server (e.g., Heroku or Vercel), all traffic to GitHub will originate from a single server IP. This will virtually guarantee we hit GitHub's API rate limits immediately, even with caching. A centralized deployment might require a complete architectural shift (e.g., storing data in a dedicated database like MongoDB instead of fetching from GitHub) or authentication with a GitHub App Key to increase rate limits.
2. **How do we handle user identity for future "Quiz Tracking" features?** The roadmap includes a "Tutor Mode" where students can track their quiz scores over time. Implementing user accounts introduces the risk of Credential Theft and Session Hijacking. We currently have zero PII (Personally Identifiable Information) risk, but storing student grades or progress changes our legal and security liability significantly. A decision is needed on whether to avoid server-side accounts entirely and use localStorage on the student's browser to track progress, maintaining the "No PII" security posture.

The Link to this document on google Doc:
https://docs.google.com/document/d/1TiqtFhr31pwNh9Q_fpWBPVuuodEcDir3Wtb5TdDh-6A/edit?usp=sharing