Node.js Developer Assignment: File Management System API

Project Overview:

You are tasked with developing a **Node.js API** for a file management system. The API will allow users to manage folders and documents in a hierarchical structure. The focus is on backend development, but **frontend integration (optional)** will be considered a plus point.
Use **microservice architecture** in which there are 3 services – Users, hierarchy, versions

API Requirements

Folder Endpoints:

1. **GET /viewstore**
    a. **Description:** Get initial folders in the project for the authenticated user.
    b. **Response:** List of root-level folders (folders without a parentFolder).
2. **GET /viewstore/:folderId**
    a. **Description:** Get the content of a folder, including subfolders and documents.

3. **POST /folders**
    a. **Description:** Create a new folder.
    b. **Request Body:**
    ```
    {
      "name": "Folder Name",
      "parentFolder": "parentFolderId" // Optional (null for root-level folders)
    }
    ```
    c. **Response:** Created folder details.
4. **PUT /folders/:id**
    a. **Description:** Update folder details (e.g., rename folder).
    b. **Request Body:**
    ```
    {
      "name": "Updated Folder Name"
    }
    ```
    c. **Response:** Updated folder details.
5. **DELETE /folders/:id**
    a. **Description:** Delete a folder.
    b. **Response:** Success message.

Document Endpoints:

1. **GET /documents/:id**
    a. **Description:** Get document details.
    Response:

```
{
  "id": "documentId",
  "title": "Document Title",
  "folder": "folderId",
  "createdAt": "timestamp",
  "versions": [
    {
      "version": "1.0",
      "fileUrl": "https://storage.example.com/file1.pdf",
      "uploadedAt": "timestamp"
    }
  ]
}
```

    b.

2. **POST /documents**
    a. **Description:** Create a new document.
    b. **Request Body:**

```
{
  "title": "Document Title",
  "content": "Document Content",
  "folder": "folderId"
  “file”: binary file data
}
```

    c. **Response:** Created document details.
3. **POST /documents/:id/version**
    a. **Description:** create version in the document
    b. **Request Body:**

```
{
  “versionNumber”: “1.1”
}
```

    c. Response:

```
{
  "id": "documentId",
  "version": "1.0",
  "fileUrl": "https://storage.example.com/file1.pdf",
  "uploadedAt": "timestamp"
}
```

    d. Rules:
        i. When a document is created, it **does not contain an actual file**, just a placeholder.
        ii. When a file is uploaded, **version 1.0** is created.
        iii. If the user uploads a new version, it is stored as **1.1, 2.0, etc.** based on versioning rules.

iv. Previous versions must be **retained** for reference.

4. **GET /documents/:id/versions**
    a. **Description:** Retrieve all versions of a document.
    b. Response:

```
[
  { "version": "1.0", "fileUrl": "https://storage.example.com/file1.pdf", "uploadedAt": "timestamp"
},
  { "version": "1.1", "fileUrl": "https://storage.example.com/file1_v1.1.pdf", "uploadedAt":
"timestamp" }
]
```

5. **PUT /documents/:id**
    a. **Description:** Update document details (e.g., rename or update content).
    b. **Request Body:**

```
{
  "title": "Updated Document Title",
  "content": "Updated Document Content"
}
```

    c. **Response:** Updated document details.
6. **DELETE /documents/:id**
    a. **Description:** Delete a document. All version included
    b. **Response:** Success message.

Filter Endpoint:

1. **GET /filter**
    a. **Description:** Return documents along with the path of the folder in which the document is stored.
    b. **Query Parameters:**
        i. search: Search term to filter documents by title or content.
    c. **Response:**

```
[
  {
    "id": "documentId",
    "title": "Document Title",
    "folderPath": "Root/Folder/Subfolder" // Full path of the folder
  }
]
```

Total Document Count Endpoint:

1. **GET /total-documents**

a. **Description:** Return the count of documents for the authenticated user.
b. **Response:**

```
{
  "totalDocuments": 25
}
```

Technical Requirements

Backend:

- **Framework:** Node.js with Express.js.
- **Database:** MongoDB with Mongoose.
- **Authentication:** JWT-based authentication.
- **Error Handling:** Proper HTTP status codes and meaningful error messages.
- **Validation:** Use **Joi** or **express-validator** for input validation.
- **Logging:** Implement a logging system for API requests and errors.

Frontend (Optional- Bonus):

- **Template Engine:** Use **EJS** for rendering views (optional).
- **ReactJS Integration (Plus Point):** If you are comfortable with ReactJS, you can create a simple frontend to interact with the API.

Submission Guidelines

- **GitHub Repo:** Include complete source code, Postman/Swagger documentation, and a README file with:
  - Architecture decisions.
  - Trade-off analysis.
  - Setup instructions.

Evaluation Criteria

- **Code Quality:** Clean, modular, maintainable.
- **Functionality:** All requirements met.
- **Database Integration:** Efficient schema design and indexing.
- **Error Handling:** Graceful error handling and user feedback.
- **Advanced Features:** Effective implementation of chosen features.

- **Frontend Integration (Bonus):** If implemented, ensure it works seamlessly with the API.

Evaluation Focus

1. Tree structure implementation quality.
2. Permission handling in nested resources.
3. MongoDB schema design for hierarchies.
4. Race condition prevention.
5. Proper transaction usage.
6. Memory management for large documents.
7. API response structure for hierarchical data.

This assignment tests advanced data modeling skills and complex relationship handling while maintaining security boundaries. The folder/document paradigm allows assessment of recursive logic implementation and performance optimization awareness.