Patient Portal - Design Document

1. Tech Stack Choices

Q1. Frontend Framework

Choice: React

Reasoning:

- **Component-based architecture**: Allows for modular, reusable UI components (upload form, document list, etc.)
- **State management**: React hooks (useState, useEffect) provide simple yet powerful state management for this application
- **Rich ecosystem**: Extensive libraries available (lucide-react for icons)
- **Developer experience**: Fast development with hot reloading and excellent debugging tools
- **Industry standard**: Widely adopted, making it easier for team collaboration

Q2. Backend Framework

Choice: Node.js with Express

Reasoning:

- **JavaScript everywhere**: Same language as frontend, reducing context switching
- **Minimal boilerplate**: Express is lightweight and unopinionated, perfect for simple REST APIs
- **File handling**: Node.js has excellent built-in modules (fs, path) for file operations
- **Middleware ecosystem**: Easy integration with multer (file uploads), cors, etc.

- **Fast development**: Quick to set up and deploy for prototyping
- **Non-blocking I/O**: Efficient handling of file uploads/downloads

Alternatives considered:

- Flask/Django (Python): Would require context switching between languages
- FastAPI (Python): Great for async operations but overkill for this simple use case

Q3. Database

Choice: SQLite

Reasoning:

- **Zero configuration**: File-based database, no server setup required
- **Perfect for single-user scenario**: The requirement states "assume one user for simplicity"
- **Lightweight**: Minimal overhead, fast for small datasets
- **Easy deployment**: Database file can be committed or easily backed up
- **ACID compliant**: Ensures data integrity despite being file-based
- **Built-in with Node.js**: sqlite3 npm package is mature and well-maintained

Why not PostgreSQL:

- Overkill for a single-user local application
- Requires separate database server installation and configuration
- More complex setup would slow down development

When to use PostgreSQL:

- Multi-user production environment
- Need for advanced features (full-text search, JSON queries)
- Horizontal scaling requirements

Q4. Supporting 1,000 Users - Scalability Considerations

Infrastructure Changes:

1. **Database Migration:**
    - Move from SQLite to PostgreSQL or MySQL
    - Implement connection pooling
    - Add database indexing on frequently queried fields (created_at, user_id)
    - Consider read replicas for scaling reads
2. **File Storage:**
    - Move from local filesystem to cloud storage (AWS S3, Google Cloud Storage, Azure Blob)
    - Implement CDN for faster file downloads
    - Use presigned URLs for secure, direct downloads
    - Implement file size quotas per user
3. **Authentication & Authorization:**
    - Implement JWT-based authentication
    - Add user management system
    - Row-level security in database (user can only access their own documents)
    - Add API rate limiting per user
4. **Application Architecture:**

- Deploy behind a load balancer (nginx, AWS ALB)
- Containerize with Docker for consistent deployments
- Implement horizontal scaling with multiple backend instances
- Add Redis for session management and caching

5. **API Improvements:**
   - Add pagination for document listing
   - Implement search and filtering
   - Add API versioning (/api/v1/documents)
   - Implement request validation middleware

6. **Security Enhancements:**
   - HTTPS/TLS encryption
   - Input sanitization and validation
   - Virus scanning for uploaded files
   - Implement HIPAA compliance measures (encryption at rest, audit logs)
   - Add Content Security Policy headers

7. **Monitoring & Observability:**
   - Application logging (Winston, Bunyan)
   - Error tracking (Sentry, Rollbar)
   - Performance monitoring (New Relic, DataDog)
   - Database query performance monitoring

8. **Backup & Recovery:**
   - Automated database backups

- File storage versioning
- Disaster recovery plan

2. Architecture Overview

System Architecture Diagram

(PDF Storage)

Component Flow

Frontend (React):

- User Interface Components
- State Management (useState, useEffect)
- API Communication (fetch API)
- File Handling (FormData, Blob)

Backend (Express.js):

- Route Handlers
- Multer Middleware (file upload)
- CORS Middleware
- Error Handling
- File System Operations

Database (SQLite):

- Stores file metadata
- Provides querying capabilities

File Storage:

- Local `uploads/` directory
- Stores actual PDF files

Data Flow

1. **Upload Flow:** Frontend → FormData with PDF → Backend API → Multer processes → Save to uploads/ → Save metadata to DB → Return success
2. **List Flow:** Frontend → Request list → Backend API → Query DB → Return metadata array → Display in UI
3. **Download Flow:** Frontend → Request file by ID → Backend API → Query DB for filepath → Read file → Stream to client → Browser downloads
4. **Delete Flow:** Frontend → Request delete by ID → Backend API → Query DB → Delete file from filesystem → Delete DB record → Return success

3. API Specification

Base URL

```
http://localhost:3001/api
```

Endpoint 1: Upload PDF Document

URL: `/documents/upload`
Method: `POST`
Content-Type: `multipart/form-data`

Request:

```
POST /api/documents/upload
Content-Type: multipart/form-data

Form Data:
- document: [PDF File]
```

Sample Request (cURL):

```bash
curl -X POST http://localhost:3001/api/documents/upload \
  -F "document=@prescription.pdf"
```

Success Response (201 Created):

```json
{
  "id": 1,
  "filename": "1638234567890-9876543210-prescription.pdf",
  "filesize": 245678,
  "message": "File uploaded successfully"
}
```

Error Responses:

```json
```

```
// 400 Bad Request - No file
{
  "error": "No file uploaded"
}

// 400 Bad Request - Not a PDF
{
  "error": "Only PDF files are allowed"
}

// 400 Bad Request - File too large
{
  "error": "File size exceeds 10MB limit"
}

// 500 Internal Server Error
{
  "error": "Database error: [error message]"
}
```

Validation Rules:

- File must be PDF (application/pdf)
- Maximum file size: 10MB
- File is required

Endpoint 2: List All Documents

URL: `/documents`

Method: GET

Request:

```
GET /api/documents
```

Sample Request (cURL):

```bash
curl http://localhost:3001/api/documents
```

Success Response (200 OK):

```json
[
  {
    "id": 3,
    "filename": "1638234567890-9876543210-test-results.pdf",
    "filepath": "/path/to/uploads/1638234567890-9876543210-test-results.pdf",
    "filesize": 456789,
    "created_at": "2024-12-09 14:30:00"
  },
  {
    "id": 2,
    "filename": "1638234512345-1234567890-prescription.pdf",
    "filepath": "/path/to/uploads/1638234512345-1234567890-prescription.pdf",
    "filesize": 245678,
    "created_at": "2024-12-08 10:15:00"
  }
]
```

Error Response:

```
json

// 500 Internal Server Error
{
  "error": "Database error: [error message]"
}
```

Notes:

- Returns documents ordered by created_at DESC (newest first)
- Empty array if no documents exist

Endpoint 3: Download Specific Document

URL: `/documents/:id`
Method: `GET`
URL Parameters: `id` (integer) - Document ID

Request:

```
GET /api/documents/3
```

Sample Request (cURL):

```
bash
```

```
curl http://localhost:3001/api/documents/3 \
  -o downloaded-file.pdf
```

Success Response (200 OK):

- Returns the PDF file as binary data
- Content-Type: application/pdf
- Content-Disposition: attachment; filename="[original-filename].pdf"

Error Responses:

```
json

// 404 Not Found - Document doesn't exist
{
  "error": "Document not found"
}

// 404 Not Found - File missing from filesystem
{
  "error": "File not found on server"
}

// 500 Internal Server Error
{
  "error": "Database error: [error message]"
}
```

Endpoint 4: Delete Document

URL: `/documents/:id`
Method: `DELETE`
URL Parameters: `id` (integer) - Document ID

Request:

```
DELETE /api/documents/3
```

Sample Request (cURL):

```bash
curl -X DELETE http://localhost:3001/api/documents/3
```

Success Response (200 OK):

```json
{
  "message": "Document deleted successfully"
}
```

Error Responses:

```json
```

```
// 404 Not Found
{
  "error": "Document not found"
}

// 500 Internal Server Error
{
  "error": "Database error: [error message]"
}
```

Notes:

- Deletes both the file from filesystem and the database record
- If file doesn't exist in filesystem, still deletes DB record
- Operation is irreversible

---

4. Data Flow Description

Q5: File Upload Process (Step-by-Step)

Upload Flow:

1. **Frontend: User Selection**
    - User clicks upload area or input
    - Browser opens file picker
    - User selects a PDF file
2. **Frontend: Validation**

- Check file type (must be application/pdf)
- Check file size (must be ≤ 10MB)
- Display error if validation fails

3. **Frontend: Prepare Request**
   - Create FormData object
   - Append file with key "document"
   - Set uploading state to true
4. **Frontend: Send Request**
   - POST request to `/api/documents/upload`
   - Content-Type: multipart/form-data
5. **Backend: Receive Request**
   - Express receives request
   - CORS middleware validates origin
   - Multer middleware intercepts
6. **Backend: Multer Processing**
   - Validates file type (PDF only)
   - Validates file size (10MB max)
   - Generates unique filename (timestamp + random + original name)
   - Saves file to `uploads/` directory
   - Attaches file info to req.file
7. **Backend: Database Storage**
   - Extract filename, filepath, filesize from req.file

- Execute SQL INSERT query
- Store metadata in documents table
- Retrieve auto-generated ID

8. **Backend: Error Handling**
    - If DB insert fails, delete uploaded file
    - Return appropriate error response
9. **Backend: Success Response**
    - Return 201 Created status
    - Return document ID, filename, size in JSON
10. **Frontend: Handle Response**
    - Display success message
    - Refresh document list
    - Reset file input
    - Set uploading state to false

File Download Process (Step-by-Step)

Download Flow:

1. **Frontend: User Action**
    - User clicks download button for a specific document
    - Document ID is captured
2. **Frontend: Send Request**

- GET request to `/api/documents/:id`
- ID is passed as URL parameter

3. **Backend: Receive Request**
    - Express receives request with ID parameter
    - Parse ID from URL
4. **Backend: Database Query**
    - Execute SQL SELECT query: `SELECT * FROM documents WHERE id = ?`
    - Retrieve document metadata (filename, filepath)
5. **Backend: Validation**
    - Check if document exists in database (404 if not)
    - Check if file exists in filesystem (404 if not)
6. **Backend: File Reading**
    - Read file from filepath using fs module
    - Prepare file stream for transfer
7. **Backend: Send Response**
    - Set Content-Type: application/pdf
    - Set Content-Disposition: attachment; filename="[original-name]"
    - Stream file data to response
8. **Frontend: Receive Response**
    - Receive binary data (blob)
    - Create object URL from blob

9. **Frontend: Trigger Download**
    - Create temporary anchor element
    - Set href to object URL
    - Set download attribute to filename
    - Programmatically click anchor
    - Browser downloads file
10. **Frontend: Cleanup**
    - Revoke object URL to free memory
    - Remove anchor element
    - Display success message

---

5. Assumptions

Q6: Assumptions Made During Development

1. **Single User Environment**
    - No authentication or user management required
    - All documents belong to a single user
    - No user_id field in database
    - No session management needed
2. **File Type Restrictions**
    - Only PDF files are supported (application/pdf)
    - No support for images, Word docs, or other formats

- Assumption: Medical documents are typically PDFs

3. **File Size Limits**
    - Maximum file size: 10MB per document
    - Assumption: Most prescriptions and test results are under 10MB
    - Large imaging files (MRIs, CT scans) not supported
4. **Local Deployment**
    - Application runs on localhost
    - No HTTPS/SSL required
    - No cloud deployment considerations
    - Single server instance
5. **No Concurrent Upload Handling**
    - Users upload one file at a time
    - No queue system for multiple uploads
    - No progress tracking for large files
6. **File Naming**
    - Backend generates unique filenames using timestamp + random number
    - Original filename preserved in database for display
    - No duplicate filename handling needed (unique names guaranteed)
7. **Storage Capacity**
    - Unlimited storage assumed (local disk space)
    - No quota per user
    - No automatic cleanup of old files

8. **Network Reliability**
    - Stable network connection assumed
    - No retry logic for failed uploads/downloads
    - No resumable uploads
9. **Data Integrity**
    - File and database always in sync
    - No orphaned files (file exists but no DB record)
    - No orphaned records (DB record exists but no file)
    - If either operation fails, both are rolled back
10. **Security**
    - No HIPAA compliance requirements for prototype
    - No encryption at rest
    - No encryption in transit (HTTP not HTTPS)
    - No virus scanning of uploaded files
    - No file content validation (just MIME type)
    - Trust that uploaded files are legitimate PDFs
11. **Browser Compatibility**
    - Modern browsers (Chrome, Firefox, Safari, Edge)
    - JavaScript enabled
    - Fetch API supported
    - File API supported
12. **Error Handling**

- Basic error messages sufficient
- No detailed logging system
- Console logs for debugging only

13. **Performance**
    - Low traffic expected
    - No caching strategy needed
    - Database queries are fast enough (SQLite for small datasets)
    - No pagination needed for document list
14. **Data Retention**
    - Documents stored indefinitely
    - No automatic expiration
    - Manual deletion only
15. **Metadata**
    - Minimal metadata stored (filename, size, date)
    - No document categorization (prescription, test result, etc.)
    - No tags or search functionality
    - No document versioning

Future Enhancements (Out of Scope)

- Multi-user support with authentication
- Document categorization and tagging

- Full-text search within PDFs
- Document preview (PDF viewer)
- Sharing documents with healthcare providers
- Mobile app support
- Email notifications
- Audit logs
- Two-factor authentication
- Encrypted storage