System Requirements Specification Index

For

Document Management System

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

TABLE OF CONTENTS

1 Project Abstract

A small legal office needs a comprehensive document management system to handle various file types including text documents, PDFs, JSON data files, binary files, and spreadsheets. The system will organize client information, track case documents, manage billing data, and provide secure document archiving capabilities. This application will enable efficient document search, controlled access, version tracking, and automated backups.

2 Business Requirements:

Screen Name	Console input screen
Problem Statement	1. Process and convert text files in various encodings (UTF-8, ASCII) 2. Read and write structured data in JSON format 3. Handle binary file operations for document archiving 4. Generate reports by combining data from multiple file sources 5. Implement secure file locking and access control mechanisms 6. Organize client information and case documents 7. Track document versions and modifications 8. Enable efficient document search and retrieval 9. Provide automated backup and recovery mechanisms

3 Constraints

3.1 File Format Requirements

1. Client Information File (`clients.json`):

- Format: JSON object with client records
- Example:
  ```json
  {
   "clients": [
    {
     "id": "CL001",
     "name": "Jane Smith",
     "contact": "jane.smith@email.com",
     "cases": ["CA001", "CA003"]
    }
   ]
  }
  ```
- Must validate client IDs follow pattern "CL" + 3 digits
- All clients must include a cases list, even if empty

2. Case Documents (`case_CAXXXX.txt`):
   - Format: Structured text with metadata header and content body
   - Example:

     ```
     TITLE: Smith v. Johnson
     DATE: 2023-03-15
     STATUS: Active
     ATTORNEY: Michael Davis
     ---
     Case notes and details follow here...
     Multiple lines of text with potentially
     special characters and formatting.
     ```
   - Must preserve original formatting
   - Must parse metadata header correctly
   - Must strip whitespace from both keys and values in metadata
3. Billing Records (`billing.json `):
   - Format: JSON array of billing entries
   - Example:

```json
{
 "billing": [
  {
   "case_id": "CA001",
   "date": "2023-03-15",
   "hours": 2.5,
   "rate": 150.00,
```

```
      "amount": 375.00,
      "description": "Initial consultation"
    }
  ]
}
```

- Must validate numeric values
- Must ensure dates are in YYYY-MM-DD format
- Must calculate and store amount field (hours × rate)

4. Document Archive (Binary files):
   - Format: Custom binary format with header and content sections
   - Fixed-length header with metadata (32 bytes)
   - Variable-length content section
   - Must include checksum for integrity verification

3.2 Data Validation Requirements

1. Client Data Validation:
   - Client IDs must follow required pattern (CL + 3 digits)
   - Email addresses must be valid format
   - Phone numbers must follow consistent format
   - All client objects must include a cases array
2. Case Document Validation:
   - Case IDs must follow pattern "CA" + 3 digits
   - Dates must be in valid YYYY-MM-DD format
   - Status must be one of predefined values
   - Document content must not exceed 10MB
3. Billing Validation:
   - Hours must be positive numbers with up to one decimal place
   - Rates must be positive numbers with up to two decimal places
   - Descriptions must not be empty
   - Total billing amount must be calculated accurately as hours × rate
   - Amount must be rounded to two decimal places

4. TEMPLATE CODE STRUCTURE:

1. Client Management Functions:

- `load_clients(file_path)` - reads client data from JSON file
  - Returns dictionary of client objects
  - Must handle file not found and invalid JSON errors
  - Example: `{"CL001": {"name": "Jane Smith", "contact": "jane.smith@email.com", "cases": ["CA001"]}}`
- `add_client(file_path, client_id, name, contact)` - adds new client
  - Validates client data
  - Updates JSON file with new client
  - Always initializes new clients with an empty cases list
  - Returns success/failure status
- `search_clients(file_path, search_term)` - searches client records
  - Case-insensitive search across client data
  - Returns list of matching client objects
  - Must handle no matches found

2. Document Processing Functions:

- `read_case_document(file_path)` - reads and parses case document
  - Separates metadata and content
  - Returns dictionary with parsed structure
  - Must strip whitespace from metadata keys and values
  - Must handle different text encodings
- `create_case_document(file_path, title, date, status, attorney, content)` - creates document
  - Formats metadata header
  - Writes structured document to file
  - Must validate date is in YYYY-MM-DD format
- `update_case_document(file_path, field, value)` - updates document metadata
  - Modifies specified field in document header
  - Preserves remaining content
  - Must validate field names and values

3. Billing Functions:

- `record_billing(file_path, case_id, date, hours, rate, description)` - adds billing entry
  - Validates case_id follows "CA" + 3 digits pattern
  - Validates hours and rate are positive numbers
  - Calculates and stores total amount = hours × rate (rounded to 2 decimal places)
  - Updates JSON billing file
- `generate_invoice(billing_file, client_file, case_id, output_file)` - creates invoice

    - Combines client and billing data
    - Formats detailed invoice as text file
    - Uses stored amount values to ensure calculation consistency
    - Calculates totals from all entries

4. Binary File Functions:

- `archive_document(source_path, archive_path)` - archives document in binary format
    - Reads source document
    - Creates binary archive with metadata header
    - Calculates and appends checksum
- `extract_document(archive_path, output_path)` - extracts from archive
    - Verifies checksum for integrity
    - Extracts original document content
    - Restores to original format

5. File System Operations:

- `create_case_directory(base_path, case_id)` - creates directory structure
    - Validates case_id follows "CA" + 3 digits pattern
    - Creates case-specific directory
    - Sets up subdirectories for documents, billing, etc.
    - Creates empty case info file with basic metadata
    - Returns path to created directory
- `list_case_files(case_path, file_type=None)` - lists files in case directory
    - Recursively walks through all subdirectories
    - Optional filtering by file type/extension
    - Returns list of file paths with relative paths and metadata
    - Sorts by modification date (newest first)

6. Backup Operations:

- `backup_files(source_dir, backup_dir)` - backs up important files
    - Creates timestamp-based backup copies
    - Preserves directory structure in backup
    - Handles both .json and .txt files
    - Returns count of files backed up

7. Main Function:

- `main()` - provides menu-driven interface
    - Displays interactive menu
    - Handles user input
    - Calls appropriate functions based on selections
    - Displays results or error messages

5. Execution Steps to follow:

1. Implement file handling functions for each file type:
   - JSON functions with proper validation
   - Text file processing with encoding support
   - Binary file operations with integrity checks
   - File system operations for organization

2. Create a structured directory organization:
   - Main directory for client data
   - Case-specific subdirectories
   - Archive directory for historical documents
   - Temporary directory for in-process files

3. Implement comprehensive error handling:
   - Validate input before writing to files
   - Handle file system errors gracefully
   - Create detailed error logs
   - Implement recovery mechanisms where possible

4. Build user interface components:
   - Menu system for navigation
   - Input forms for data entry
   - Search interface for document retrieval
   - Reporting interface for generating summaries

5. Test with various file scenarios:
   - Normal operation with valid data
   - Error conditions with invalid data
   - Recovery from interrupted operations
   - Performance with large files and directories