University of Science and Technology of Southern Philippines

College of Information Technology and
Computing Department of Computer Science


# E-Flash

A PDF Flashcard Generator System Documentation

By:

Cagampang, Stephen Carl T.

Dimarucut, Jon Paolo L.

Tocle, Gabriel Luke M.

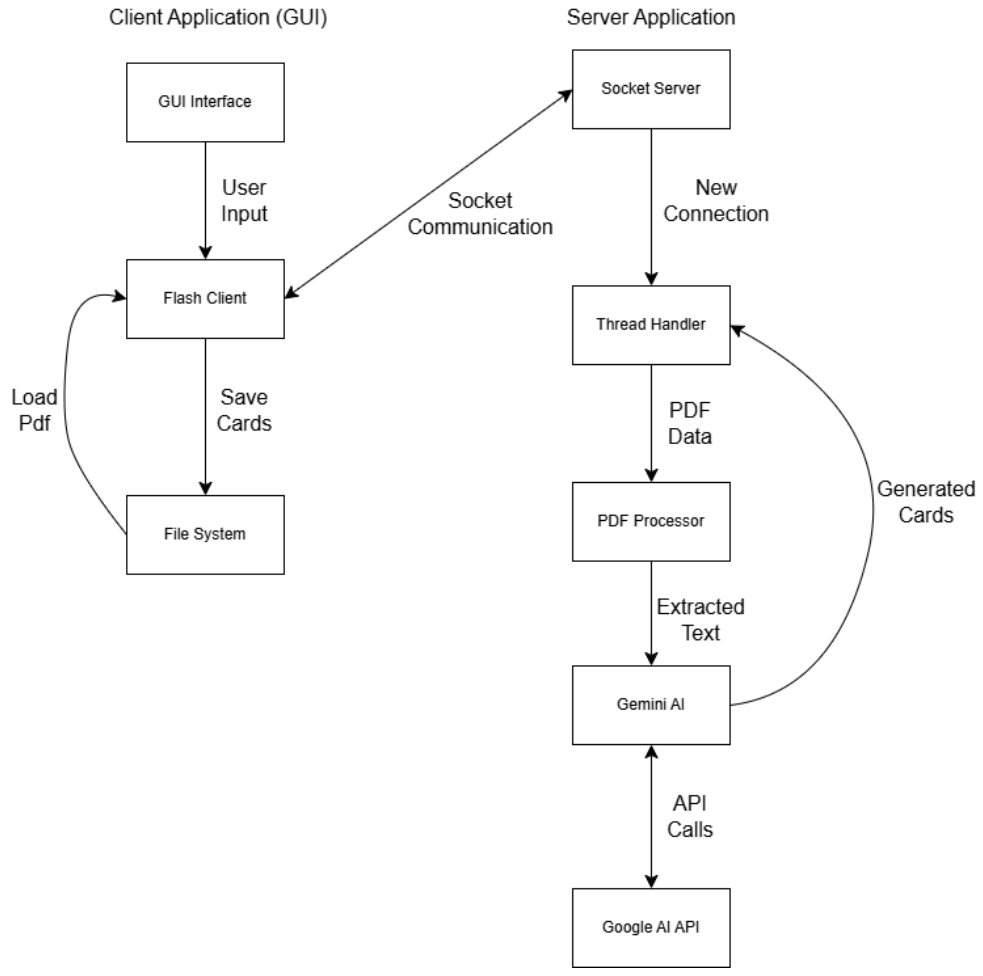
December 21, 2024

1. Project Overview & Objectives

The E-Flash a PDF Flashcard generator documentation and Client-Server System is designed to create an efficient client-server application that facilitates the uploading and processing of PDF files. Utilizing TCP socket communication, the system enables users to send files to a server, which extracts text from the documents, processes the content, and generates AI-powered flashcards. This project demonstrates the integration of networking principles, file handling, and AI capabilities to deliver an innovative educational tool.

The primary objective of this project is to develop a functional client-server application that allows reliable file transfers over a network. The server processes the PDF content using Python libraries and AI algorithms to generate question-and-answer flashcards, which can be displayed on the client interface for easy navigation. Additionally, the system incorporates error-handling mechanisms to address potential issues in file transfer or connection, ensuring a robust and reliable experience for users.

2. System Architecture (Client-Server System Architecture)



- Client Side:
    - GUI Interface: Handles user interactions and display
    - Flashcard Client: Manages communication with server
    - File System: Flashcard saving

- Server Side:
    - Socket Server: Manages client connections
    - Thread Handler: Enables multiple client support
    - PDF Processor: Extracts text from PDFs
    - AI Integration: Interfaces with Gemini AI

- Network Layer Components:
    - Socket Connection Management
    - Error Detection and Handling
    - Protocol Implementation

- Technology Stack Details:
    - Python as primary language
    - Core libraries (socket, os, PyPDF2, google.generativeai)
    - Custom protocol specifications
    - Threading for concurrent connections

- Error Handling in Network Layer:
    - Connection failures
    - Data transfer errors

- Protocol Specifications:
    - File chunking (4096 bytes)
    - Header information

3. Code Structure

The code structure for the E-Flash project consists of two main components: client.py and server.py. The client explicitly interacts with the user through a graphical user interface (GUI) made using Tkinter and provides features for uploading PDF files, rendering the resulting flashcards, and saving the contents of the flashcards to a local directory. It also handles network responsibilities like, joining to the server and sending file data through the TCP sockets. The server.py receives client requests by pulling text out of PDFs, generating flashcards via AI, and responding to the client. Both of them are modular so networking, file handling and AI integration is implemented into the program clearly and can be expanded if needed.

E-Flash/

client.py

```
├── FlashcardClient          # Handles networking
│   ├── test_connection      # Tests connection to the server
│   ├── send_file            # Sends a PDF to the server and receives flashcards
│
├── FlashcardGUI             # Manages the GUI
│   ├── create_server_config_frame  # Server configuration section (port setup)
│   ├── create_upload_frame         # Upload button for selecting PDF files
│   ├── create_flashcard_viewer     # Flashcard navigation and display
│   ├── create_status_bar           # Status display at the bottom
│   ├── upload_file          # Uploads PDF to server and processes response
│   ├── save_flashcards             # Saves flashcards locally
│   ├── update_card_display         # Updates question/answer display
│   ├── flip_card            # Flips between question and answer
│   ├── next_card            # Moves to the next flashcard
│   ├── previous_card               # Moves to the previous flashcard
│   └── test_server_connection      # Tests the server's availability
│
└── main                     # Initializes the GUI
```

server.py

```
├── FlashcardServer                  # Core server functionality
│   ├── initialize_socket            # Sets up server socket and port binding
│   ├── extract_text_from_pdf        # Extracts text content from uploaded PDFs
│   ├── divide_text                  # Splits large text into smaller sections
│   ├── generate_flashcards_with_ai  # Uses AI to create flashcards from text
│   ├── handle_client                # Manages client connections and requests
│   ├── cleanup                 # Closes server resources
│   └── start                   # Starts the server and listens for clients
│
└── main                        # Initializes the server
Testing/                        # Contains all unit tests for the project
│   ├── clienttest.py           # Tests client-side functionality (e.g., file transmission, server connectivity)
│   ├── fileoperationtest.py # Tests PDF processing and large file handling on the server
│   └── servertest.py           # Tests server-side networking and AI integration
├── README.md                   # Documentation for setup, usage, and project overview
├── requirements.txt            # Lists all dependencies required to run the project
```

4. Client-Side Implementation

Key Client Functions:

1. Connect to the server

**Purpose**: Test the connection to the server to ensure it is available and ready to process requests.

```
14      def test_connection(self):
15          """Test if the server is available on the specified port"""
16          try:
17              with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as test_socket:
18                  test_socket.connect((self.host, self.port))
19                  return True
20          except:
21              return False
```

2. Upload a PDF File

**Purpose**: Allow the user to upload a PDF file, which is sent to the server for processing.

```
140     def upload_file(self):
141         file_path = filedialog.askopenfilename(
142             filetypes=[("PDF files", "*.pdf")]
143         )
144         if file_path:
145             try:
146                 self.status_var.set("Uploading file and generating flashcards...")
147                 self.root.update()
148
149                 response = self.client.send_file(file_path)
150
151                 if "error" in response:
152                     messagebox.showerror("Error", response["error"])
153                     self.status_var.set("Error generating flashcards")
154                 else:
155                     self.flashcards = response
156                     self.current_card_index = 0
157                     self.showing_question = True
158                     self.update_card_display()
159                     self.status_var.set(f"Generated {len(self.flashcards)} flashcards")
160             except Exception as e:
161                 messagebox.showerror("Error", str(e))
162                 self.status_var.set("Error processing file")
```

3. Display Flashcards

**Purpose**: Show the flashcards generated by the server, with navigation and flipping between question and answer views.

```
184     def update_card_display(self):
185         if not self.flashcards:
186             self.card_text.delete(1.0, tk.END)
187             self.card_text.insert(tk.END, "No flashcards available")
188             return
189 
190         card = self.flashcards[self.current_card_index]
191         self.card_text.delete(1.0, tk.END)
192 
193         if self.showing_question:
194             self.card_text.insert(tk.END, f"Question ({self.current_card_index + 1}/{len(self.flashcards)}):\n\n")
195             self.card_text.insert(tk.END, card["question"])
196         else:
197             self.card_text.insert(tk.END, f"Answer ({self.current_card_index + 1}/{len(self.flashcards)}):\n\n")
198             self.card_text.insert(tk.END, card["answer"])
199             self.prev_btn["state"] = "normal" if self.current_card_index > 0 else "disabled"
200             self.next_btn["state"] = "normal" if self.current_card_index < len(self.flashcards) - 1 else "disabled"
```

4. Save Flashcards

**Purpose**: Save the flashcards locally as a text file for offline use.

```
164     def save_flashcards(self):
165         if not self.flashcards:
166             messagebox.showwarning("Warning", "No flashcards to save!")
167             return
168 
169         file_path = filedialog.asksaveasfilename(
170             defaultextension=".txt",
171             filetypes=[("Text files", "*.txt")],
172             initialfile="flashcards.txt"
173         )
174 
175         if file_path:
176             try:
177                 with open(file_path, 'w', encoding='utf-8') as f:
178                     for card in self.flashcards:
179                         f.write(f"Q: {card['question']}\nA: {card['answer']}\n\n")
180                 messagebox.showinfo("Success", "Flashcards saved successfully!")
181             except Exception as e:
182                 messagebox.showerror("Error", f"Error saving flashcards: {str(e)}")
```

5. Send File to a Server

**Purpose**: Handle the file upload process, send the PDF to the server, and receive the flashcards or error responses.

```
 9  class FlashcardClient:
23      def send_file(self, file_path):
24          with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
25              try:
26                  client_socket.connect((self.host, self.port))
27
28                  #Determines the file type
29                  file_type = file_path.split('.')[-1].lower()
30                  if file_type != 'pdf':
31                      raise ValueError("Unsupported file type. Only PDF files are supported.")
32
33                  #Sends the the file type
34                  client_socket.send(file_type.ljust(10).encode())
35
36                  #Read & Send file content
37                  with open(file_path, 'rb') as file:
38                      file_content = file.read()
39                      client_socket.send(str(len(file_content)).zfill(10).encode())
40                      client_socket.send(file_content)
41
42                  #Receive response size
43                  response_size = int(client_socket.recv(10).decode())
44
45                  #Receives the flashcards
46                  response = b""
47                  while len(response) < response_size:
48                      chunk = client_socket.recv(min(4096, response_size - len(response)))
49                      if not chunk:
50                          break
51                      response += chunk
52
53                  return json.loads(response.decode())
54
55              except ConnectionRefusedError:
56                  return {"error": "Could not connect to server. Please check if the server is running."}
57              except Exception as e:
58                  return {"error": str(e)}
```

5. Server-Side Implementation

1. Initialize the Server

**Purpose**: Set up the server to bind to a port and listen for client connections.

```python
    def initialize_socket(self):
        """Initialize the server socket with port finding capability"""
        max_attempts = 10
        current_port = self.port

        for attempt in range(max_attempts):
            try:
                self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                self.server_socket.bind((self.host, current_port))
                self.port = current_port
                self.server_socket.listen(5)
                print(f"Server successfully bound to port {self.port}")
                return
            except OSError:
                print(f"Port {current_port} is in use, trying next port...")
                if self.server_socket:
                    self.server_socket.close()
                current_port += 1

        raise Exception(f"Could not find an available port after {max_attempts} attempts")
```

2. Handle Client Connections

**Purpose**: Manage incoming client connections, receive files, and send back responses.

```python
    def handle_client(self, client_socket, addr):
        print(f"Connected to client: {addr}")

        try:
            # Receive file type
            file_type = client_socket.recv(10).decode().strip()

            if file_type != 'pdf':
                raise ValueError("Unsupported file type. Only PDF files are supported.")

            # Receive file size
            file_size = int(client_socket.recv(10).decode().strip())

            # Receive file content
            file_content = b""
            while len(file_content) < file_size:
                chunk = client_socket.recv(min(4096, file_size - len(file_content)))
                if not chunk:
                    break
                file_content += chunk

            # Extract text from PDF
            text = self.extract_text_from_pdf(file_content)

            # Generate flashcards using AI
            flashcards = self.generate_flashcards_with_ai(text)

            # Send flashcards back to client
            response = json.dumps(flashcards).encode()
            client_socket.send(str(len(response)).zfill(10).encode())
            client_socket.send(response)

        except Exception as e:
            error_msg = json.dumps({"error": str(e)}).encode()
            client_socket.send(str(len(error_msg)).zfill(10).encode())
            client_socket.send(error_msg)

        finally:
            client_socket.close()
```

3. Extract Text from PDFs

**Purpose**: Process the received PDF file and extract text for flashcard generation.

```python
    def extract_text_from_pdf(self, pdf_bytes):
        try:
            pdf_file = io.BytesIO(pdf_bytes)
            pdf_reader = PyPDF2.PdfReader(pdf_file)
            text = " ".join([page.extract_text() for page in pdf_reader.pages])
            return text
        except Exception as e:
            raise Exception(f"Error reading PDF file: {str(e)}")
```

4. Generate Flashcards

**Purpose**: Use an AI model to generate flashcards based on the extracted text.

```python
    def generate_flashcards_with_ai(self, text):
        try:
            divided_sections = self.divide_text(text)
            generated_flashcards = []

            for section in divided_sections[:1]:  # Limiting to first section for demo
                prompt = f"""Create flashcards from the following text.
                Return them in JSON format as an array of objects with 'question' and 'answer' fields.
                For example:
                [
                    {{"question": "What is...", "answer": "It is..."}},
                    {{"question": "How does...", "answer": "It does..."}}
                ]

                Text to process: {section}"""

                # Generate response using Gemini
                response = self.model.generate_content(prompt)

                try:
                    # Try to parse the response directly
                    content = response.text
                    # Find the JSON part in the response
                    start_idx = content.find('[')
                    end_idx = content.rfind(']') + 1

                    if start_idx != -1 and end_idx != -1:
                        json_str = content[start_idx:end_idx]
                        cards = json.loads(json_str)
                        generated_flashcards.extend(cards)
                    else:
                        print("Could not find JSON format in response")

                except json.JSONDecodeError as e:
                    print(f"Error parsing JSON: {e}")
                    print(f"Response content: {content}")
                except Exception as e:
                    print(f"Error processing response: {e}")

            return generated_flashcards

        except Exception as e:
            raise Exception(f"Error generating flashcards: {str(e)}")
```

5. Clean up Resources

**Purpose**: Close the server socket and free up resources during shutdown.

```
44      def cleanup(self):
45          """Cleanup server resources"""
46          if self.server_socket:
47              self.server_socket.close()
48              self.server_socket = None
49              print("Server socket closed")
```

6. Start the Server

**Purpose**: Start the server and listen for incoming client connections.

```
155     def start(self):
156         print(f"Server listening on {self.host}:{self.port}")
157         try:
158             while True:
159                 client_socket, addr = self.server_socket.accept()
160                 client_thread = threading.Thread(target=self.handle_client, args=(client_socket, addr))
161                 client_thread.start()
162         except KeyboardInterrupt:
163             print("\nShutting down server...")
164         finally:
165             self.cleanup()
```

6. Protocol Design

The custom protocol for E-Flash facilitates reliable communication between the client and server for PDF file transfers and flashcard generation. It ensures structured message exchange, error handling, and proper session management.

Protocol Outline:

1. **Handshake**

Purpose: Establish initial communication and validate the client-server connection.

Steps:

- Client sends a message specifying the file type (e.g., `"pdf "` padded to 10 bytes).
- Server validates the file type and responds with `"OK"` if the type is supported, or an error message if unsupported.

2. **File Transfer**

Purpose: Transmit the PDF file content from the client to the server.

Steps:

- Client sends the file size as a 10-byte header (e.g., `"000001024"` for a 1024-byte file).
- Client transmits the file content in a single operation (or chunked if necessary).
- Server receives the file and confirms successful reception.

3. **Processing and Response**

Purpose: Generate flashcards and send them back to the client.

Steps:

- Server processes the PDF file, extracts text, and generates flashcards using AI.
- Server serializes the flashcards into a JSON string.
- Server sends the response size as a 10-byte header, followed by the serialized JSON response.
- Client receives and parses the response.

4. **Error Handling**

Purpose: Handle invalid requests, file issues, or processing errors gracefully.

Mechanisms:

- If the file type is unsupported, the server sends an error message: `{"error": "Unsupported file type"}`.
- For any unexpected errors during processing, the server sends a generic error message in JSON format: `{"error": "Description of the issue"}`.

Protocol Flow:

**Step-by-Step Interaction**

1. **Handshake**:
   - Client sends: `"pdf "` (file type, padded to 10 bytes).
   - Server responds: `"OK"` if the file type is supported.
2. **File Transfer**:
   - Client sends: `"0000123456"` (file size, padded to 10 bytes).
   - Client sends: `<PDF file content>`.
3. **Processing and Response**:
   - Server processes the file and generates flashcards.
   - Server sends: `"0000000120"` (response size, padded to 10 bytes).
   - Server sends: `[{ "question": "What is AI?", "answer": "Artificial Intelligence" }, ...]` (JSON response).
4. **Error Handling** (if any issue occurs):
   - Server sends: `{"error": "Description of the issue"}`.

7. Error Handling

The E-Flash project includes several layers of error handling to ensure reliable communication and processing between the client and server. Below is an analysis of error handling strategies based on the provided code:

- **Client-Side Error Handling**
  a. Network Communication Errors
     - Handles server connection failures (e.g., server unavailable or incorrect port).

```
 9  class FlashcardClient:
23      def send_file(self, file_path):
55              except ConnectionRefusedError:
56                  return {"error": "Could not connect to server. Please check if the server is running."}
```

  b. General Exception Handling
     - Catches unexpected errors and provides meaningful error messages to the user.

```
 9  class FlashcardClient:
23      def send_file(self, file_path):
57              except Exception as e:
58                  return {"error": str(e)}
```

  c. User Feedback

- Displays error messages through the GUI to keep the user informed.

```
60   class FlashcardGUI:
140      def upload_file(self):
151                  if "error" in response:
152                      messagebox.showerror("Error", response["error"])
153                      self.status_var.set("Error generating flashcards")
```

- **Server-Side Error Handling**
  a. PDF Processing Errors
     - Catches errors during file reading or text extraction.

```
10   class FlashcardServer:
51       def extract_text_from_pdf(self, pdf_bytes):
57           except Exception as e:
58               raise Exception(f"Error reading PDF file: {str(e)}")
```

  b. AI Processing Errors
     - Handles issues with AI response parsing or invalid output formats.

```
10   class FlashcardServer:
71       def generate_flashcards_with_ai(self, text):
112          except Exception as e:
113              raise Exception(f"Error generating flashcards: {str(e)}")
```

  c. General Exception Handling for Client Requests
     - Ensures the server sends error messages back to the client in JSON format for any unexpected issues.

```
10   class FlashcardServer:
115      def handle_client(self, client_socket, addr):
147          except Exception as e:
148              error_msg = json.dumps({"error": str(e)}).encode()
149              client_socket.send(str(len(error_msg)).zfill(10).encode())
150              client_socket.send(error_msg)
```

8. Documentation

README.MD

```
# PDF Flashcard Generator


## Overview


The PDF Flashcard Generator is a client-server application designed to
  generate flashcards from PDF files using AI. The server processes
  uploaded PDFs to extract text and uses a generative AI model to
  create flashcards. The client provides a graphical interface for
  users to upload PDFs, view generated flashcards, and save them
  locally.


## Requirements


- **Python Version:** 3.7+
- **Libraries:**
  - `socket`
  - `json`
  - `PyPDF2`
  - `threading`
  - `sys`
  - `io`
  - `google.generativeai`
  - `tkinter`
  - `nltk` (optional, for further processing)
```

## Running the Server

1. Navigate to the directory containing `server.py`.
2. Run the server:

   ```bash
   python server.py [optional_start_port]
   ```

   - Replace `[optional_start_port]` with a custom starting port (default is 5000).

## Running the Client

1. Navigate to the directory containing `client.py`.
2. Run the client:

   ```bash
   python client.py
   ```

   - The client GUI will open. Configure the server port in the GUI, then upload a PDF file to generate flashcards.

## Configuration Settings

### Environment Variables

- **Google Generative AI API Key:** The server requires a valid API key to interact with the AI model. Replace the placeholder in `server.py`:

  ```python
  genai.configure(api_key='YOUR_API_KEY')
  ```

### Default Ports

- The server starts on port 5000 by default. If unavailable, it increments to the next available port.
- The client connects to port 5000 by default but can be adjusted in the GUI.

## Testing the Application

### Testing Setup

There are three testing conducted `clienttest.py` for client, `servertest.py` for server, and `fileoperationtest.py` for file operation.

Move the three files within the `testing` folder to the same directory as `client.py` and `server.py`

### Running Unit Tests

```
1. Run the scripts `clienttest.py` `servertest.py` `fileoperationtest.py`
2. Verify all tests pass.

### Manual Testing
- Start the server and client applications.
- Test file upload, flashcard generation, navigation, and saving functionality in the client GUI.

## Troubleshooting

- **Error:** "Connection Refused"
  - Ensure the server is running and reachable.

- **Error:** "File Not Found"
  - Verify the file path on the client side.

- **Error:** "API Key Invalid"
  - Confirm the API key is correct and active.
```

9. Testing

The E-Flash application underwent three primary testing phases to ensure the reliability and functionality of its core components. The testing covered:

1. **Client-Side Functionalities:** To verify the functionality of the client application, including its ability to connect to the server, handle file transfers, and gracefully manage errors.

**Tests Performed:`clienttest.py` script**

- **Connection Tests: `clienttest.py` script**
  - **Test: Successful Connection**
    - Simulated a successful connection to the server using a mock socket.
    - Verified that the client correctly establishes a connection with the server at the specified host and port.
  - **Test: Connection Failure**
    - Simulated a connection refusal scenario.
    - Verified that the client correctly identifies a failed connection attempt and returns `False`.
- **File Transfer Tests:**
  - **Test: Successful File Transfer**
    - Created a dummy PDF file and simulated sending it to the server using a mock socket.
    - Verified that the client sends the correct file type, size, and content.
    - Ensured the client correctly interprets the server's response and returns flashcards in JSON format.
  - **Test: Invalid File Type**
    - Attempted to upload a file with an unsupported extension (e.g., `.txt`).
    - Verified that the client identifies the invalid file type and returns an error message indicating unsupported file types.
  - **Test: Connection Error During File Transfer**
    - Simulated a scenario where the server is unavailable during the file transfer.
    - Verified that the client gracefully handles the error and returns an appropriate error message.

**Edge Cases Covered:**

- Handling unsupported file types.
- Managing server unavailability during connection attempts or file transfers.
- Ensuring the correct data format and content are sent during file transfers.

2. **Server-Side Functionalities:** To verify the reliability and functionality of the server

application, including its ability to manage connections, handle PDF processing, and generate flashcards.

**Tests Performed: `servertest.py` script**

- **Port Binding Test:**
  - **Objective:**
    - Ensure the server binds to the specified port (`5555`) during initialization.
  - **Process:**
    - Verified the port used by the server matches the expected value.
  - **Outcome:**
    - Confirmed the server dynamically binds to the correct port and handles conflicts gracefully.
- **Client Handling Test:**
  - **Objective:**
    - Test the server's ability to receive and process client requests.
  - **Process:**
    - Mocked the server's `extract_text_from_pdf` and `generate_flashcards_with_ai` methods.
    - Simulated a client uploading a PDF file and verified the server's response.
  - **Outcome:**
    - Confirmed the server correctly processes PDF files and returns flashcards in the expected format.
- **Concurrent Connections Test:**
  - **Objective:**
    - Verify the server can handle multiple clients simultaneously.
  - **Process:**
    - Simulated five concurrent clients connecting to the server, sending mock PDF data, and disconnecting.
  - **Outcome:**
    - Confirmed the server efficiently handles multiple connections without errors or data loss.

**Edge Cases Covered:**

- Server's ability to bind to alternate ports if the default port is unavailable.
- Handling of concurrent requests from multiple clients.
- Simulated realistic client-server interactions, including proper handling of file size and content metadata.

3. **File Operation Functionalities:** To validate the application's ability to handle file-related tasks, including extracting text from PDF files, dividing text into

manageable sections, and processing large files.

**Tests Performed: `fileoperationtest.py` script**

- **PDF Text Extraction Test:**
  - **Objective:**
    - Ensure text can be accurately extracted from a PDF file.
  - **Process:**
    - Created a test PDF with sample text.
    - Verified the server's `extract_text_from_pdf` method successfully extracts non-empty text.
  - **Outcome:**
    - Confirmed text extraction works as expected, returning a valid string.
- **Text Division Test:**
  - **Objective:**
    - Ensure long text is divided into chunks of specified sizes.
  - **Process:**
    - Passed a string of 2000 characters to the server's `divide_text` method with a chunk size of 1000.
    - Verified the text is divided into two chunks, each 1000 characters long.
  - **Outcome:**
    - Confirmed the text is divided correctly and maintains integrity.
- **Large File Handling Test:**
  - **Objective:**
    - Assess the server's capability to process large PDF files.
  - **Process:**
    - Generated a 10-page PDF with repeated content.
    - Verified the `extract_text_from_pdf` method handles the larger file and extracts a substantial amount of text.
  - **Outcome:**
    - Confirmed the server handles large files efficiently without errors or performance degradation.

**Edge Cases Covered:**

- Handling small and large PDF files.
- Correct division of overly lengthy text for efficient processing.
- Ensuring non-empty, valid text is extracted even from complex files.