web-server\app.py

```python
from utils.Http import *
from process_requests import process_req
import json
import socket
import threading
from datetime import datetime
import file_cleanup

def main():
    ssocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ssocket.bind(("0.0.0.0", 8080))

    print("Server running with HTTP " + str(datetime.now()), end="\n\n")
    cleanup_thread = threading.Thread(target=file_cleanup.main, daemon=True)
    cleanup_thread.start()
    print("file cleanup started")

    while True:
        try:
            ssocket.listen()
            client_socket, client_address = ssocket.accept()
            print(str(datetime.now()) +  " - " + client_address[0] + " made a request")
            thread = threading.Thread(target=serve_client, args=(client_socket, client_address))
            thread.daemon = True
            thread.start()
        except Exception as e:
            pass


def serve_client(client_socket, client_address):

    def send(http_response: json) -> None:
        """
        Encodes and sends a JSON-formatted data packet through a client socket.

        This function prepares a JSON-encoded string (`data2`) for transmission:

        Args:
            http_response (dict): A dictionary containing data to be sent.

        Returns:
            None
        """
        http_response = d2h(http_response)
        client_socket.sendall(http_response)

    def get() -> json:
        """
        Receives HTTP data from the socket and returns it as json

        Returns:
```

```
52              dict: A dictionary containing parsed JSON data received from the client socket.
53          """
54          data = recvall(client_socket).decode()
55          data = h2d(data)
56          return json.loads(data)
57
58      try:
59          data = get()
60          response = process_req(data)
61          send(response)
62      except Exception:
63          print(f"Connection aborted {client_address} right now it means he sent a request i
    cant handle")
64      finally:
65          client_socket.close()
66
67  if __name__ == "__main__":
68      main()
69
```

web-server\process_requests.py

```python
import json
from utils.Core import Core
from utils.Files import Files
from utils.User import User

def process_req(http_request: json) -> bytes:
    """
    Processes an HTTP request and returns an HTTP response.


    Args:
        http_request (json): The HTTP request to process.

        It should contain the following keys:
            - "endpoint" (str): The endpoint being accessed.
            - "method" (str): The HTTP method (e.g., "GET", "POST").
            - "path" (str, optional): The specific path within the endpoint.
            - "body" (dict, optional): The body of the request, if applicable.
    Returns:
        bytes: The HTTP response as a JSON-encoded byte string.

        The response contains:
            - "response_code" (str): The HTTP response code.
            - "headers" (dict): The HTTP headers.
            - "body" (str): The HTML body of the response.
    """
    response = {
        "response_code": "400 Bad Request",
        "headers": {
            "Content-Type": "text/html"
        },
        "body": "<p>Bad Request</p>"
    }

    match http_request["endpoint"]:
        case "auth":
            match http_request["method"]:
                case "GET":
                    response = User.auth(http_request, response)
        case "pages":
            match http_request["method"]:
                case "GET":
                    response = Core.get_page(http_request, response)

        case "resources":
            match http_request["method"]:
                case "GET":
                    response = Core.get_resource(http_request, response)

        case "scripts":
            match http_request["method"]:
```

```
                case "GET":
                    response = Core.get_script(http_request, response)

        case "styles":
            match http_request["method"]:
                case "GET":
                    response = Core.get_style(http_request, response)

        case "users":
            match http_request["method"]:
                case "POST":
                    match http_request["path"]:
                        case "login":
                            response = User.login(http_request["body"], response)
                        case "signup":
                            response = User.signup(http_request["body"], response)
                case "PUT":
                    # might add some functions that have to do with the user's account and
data
                    pass
                case "GET":
                    match http_request["path"]:
                        case "info":
                            response = User.info(http_request, response)
                        case "admin":
                            response = User.admin_info(http_request, response)
        case "share":
            match http_request["method"]:
                case "GET":
                    Files.get_shared_file(http_request, response)
                case "POST":
                    Files.unlock_file(http_request, response)
            pass

        case "files":
            match http_request["method"]:
                case "POST":
                    match http_request["path"]:
                        case "upload/file":
                            response = Files.upload_file_info(http_request, response)

                        case "upload/chunk":
                            response = Files.upload_chunk(http_request, response)

                        case "create/folder":
                            response = Files.create_folder(http_request, response)
                case "GET":
                    match http_request["path"]:
                        case "download":
                            response = Files.get_file_content(http_request, response)
                        case "folder":
                            response = Files.folder_content(http_request, response)
                case "DELETE":
                    match http_request["path"]:
```

```
105                         case "delete/file":
106                             response = Files.delete_file(http_request, response)
107
108     return response
```

web-server\utils\Http.py

```
import json
from datetime import datetime
import pytz

UTC = pytz.utc

def h2d(raw_request):
    """
    Parse and serialize an HTTP request string into a JSON formatted string.
    This function takes a raw HTTP request and converts it into a structured JSON format,
    breaking down the request into its components including method, endpoint, path,
    query parameters, cookies, headers, and body.
    Args:
        raw_request (str): A string containing the raw HTTP request including request line,
                           headers, and body, separated by CRLF (\r\n).
    Returns:
        str: A JSON-formatted string containing the parsed request information with the following structure:
            {
                "method": str,          # HTTP method (GET, POST, etc.)
                "endpoint": str,        # First part of the path
                "path": str,           # Remaining path after endpoint
                "query_params": dict,   # Query parameters as key-value pairs
                "cookies": list,        # List of tuples containing (cookie_name, cookie_value)
                "headers": dict,        # HTTP headers as key-value pairs
                "body": str            # Request body
    Examples:
        >>> raw_request = "GET /pages/index.html?id=1 HTTP/1.1\\r\\nHost: example.com\\r\\n\\r\\n"
        >>> serialize_http(raw_request)
        {
            "method": "GET",
            "endpoint": "pages",
            "path": "index.html",
            "query_params": {"id": "1"},
            "cookies": [],
            "headers": {"Host": "example.com"},
            "body": ""
    """
    request_info = {
        "method": "",
        "endpoint": "",
        "path": "",
        "query_params": {},
        "cookies": [],
        "headers": {},
        "body": ""
    }

    try:
```

```
        lines = raw_request.split("\r\n")
        if not lines or not lines[0]:
            return json.dumps(request_info, indent=4)

        request_line = lines[0].split(" ")
        method = request_line[0]
        full_path = request_line[1]

        # Endpoint & path
        if full_path == "/":
            endpoint = "pages"
            path = "index.html"
        else:
            path_parts = full_path[1:].split("/")
            endpoint = path_parts[0]
            path = "/".join(path_parts[1:])

        # Headers
        headers = {}
        for line in lines[1:]:
            if line == "":
                break
            try:
                header, value = line.split(":", 1)
                headers[header.strip()] = value.strip()
            except ValueError:
                continue  # Skip malformed headers

        # Cookies
        cookies = []
        if "Cookie" in headers:
            cookie_string = headers["Cookie"]
            individual_cookies = cookie_string.split("; ")
            for cookie in individual_cookies:
                parts = cookie.split("=")
                if len(parts) == 2:
                    key, value = parts
                    cookies.append((key, value))

        # Body
        try:
            body_index = lines.index("") + 1
            body = "\r\n".join(lines[body_index:]) if body_index < len(lines) else ""
        except ValueError:
            body = ""

        # Query string
        query_string = ""
        if "?" in full_path:
            path, query_string = path.split("?", 1)

        query_params = {}
        if query_string:
            for param in query_string.split("&"):
```

```
                if "=" in param:
                    k, v = param.split("=", 1)
                    query_params[k] = v

        # Final assign
        request_info.update({
            "method": method,
            "endpoint": endpoint,
            "path": path,
            "query_params": query_params,
            "cookies": cookies,
            "headers": headers,
            "body": body
        })

    except Exception as e:
        print(f"Warning: Error serializing HTTP request: {e}")

    return json.dumps(request_info, indent=4)


def d2h(response: dict) -> bytes:
    """
    Unserializes an HTTP response from a dictionary.

    Args:
        response: A dictionary containing:
            * response_code: The HTTP status code (e.g., 200, 400).
            * body: The body of the HTTP response (text or binary).
            * headers (optional): Additional HTTP headers.
            * cookies (optional): cookies to set. list of tuples (key, value, exp)

    Returns:
        The HTTP response as bytes.

    Raises:
        ValueError: If the input dictionary is missing required keys.
    """

    http_headers = f"HTTP/1.1 {response['response_code']}\r\n"
    if "headers" in response.keys():
        for key, value in response["headers"].items():
            http_headers += f"{key}: {value}\r\n"

    if "cookies" in response.keys():
        for cookie in response["cookies"]:
            expiration_date = datetime.strptime(cookie[2], "%Y-%m-%d %H:%M:%S")
            formatted_expiration = expiration_date.strftime("%a, %d %b %Y %H:%M:%S GMT")
            http_headers += f"Set-Cookie: {cookie[0]}={cookie[1]};expires=
{formatted_expiration};path=/\r\n"

    http_headers += f"Date: {datetime.now(UTC).strftime('%Y:%m:%d %H:%M:%S %Z %z')}\r\n"
    http_headers += f"Content-Length: {len(response['body'])}\r\n\r\n"

```

```
    http_response = http_headers.encode('utf-8')

    if isinstance(response['body'], str):
        http_response += response['body'].encode('utf-8')
    else:
        http_response += response['body']

    return http_response


def recvall(sock):
    """
    Efficiently receives all HTTP data from a socket,
    handling both Content-Length and chunked encoding.

    Args:
        sock (socket.socket): The socket to read from.

    Returns:
        bytes: The complete HTTP response, including headers and body.
    """

    def recv_until_delimiter(delimiter):
        buffer = b""
        while delimiter not in buffer:
            chunk = sock.recv(4096)
            if not chunk:
                break
            buffer += chunk
        return buffer

    def parse_headers(header_bytes):
        header_text = header_bytes.decode('iso-8859-1')
        headers = {}
        lines = header_text.split("\r\n")
        for line in lines[1:]:
            if ": " in line:
                key, value = line.split(": ", 1)
                headers[key.lower()] = value
        return headers

    response = recv_until_delimiter(b"\r\n\r\n")
    header_end = response.find(b"\r\n\r\n") + 4
    headers_raw = response[:header_end]
    body = response[header_end:]

    headers = parse_headers(headers_raw)

    if 'content-length' in headers:
        content_length = int(headers['content-length'])
        while len(body) < content_length:
            chunk = sock.recv(4096)
            if not chunk:
                break
```

```
            body += chunk

    elif headers.get('transfer-encoding', '').lower() == 'chunked':
        body = b""
        while True:
            chunk_size_line = b""
            while not chunk_size_line.endswith(b"\r\n"):
                chunk_size_line += sock.recv(1)
            chunk_size = int(chunk_size_line.strip(), 16)
            if chunk_size == 0:
                sock.recv(2)
                break
            chunk_data = b""
            while len(chunk_data) < chunk_size + 2:
                chunk_data += sock.recv(chunk_size + 2 - len(chunk_data))
            body += chunk_data[:-2]

    return headers_raw + body
```

web-server\utils\Core.py

```python
import os
import json
import sass

class Core:
    @staticmethod
    def get_page(http_request: dict, response) -> dict:
        """
        Retrieves the contents of a requested webpage and prepares the HTTP response.

        Args:
            http_request (dict): Dictionary containing HTTP request information, including the 'path' key
                                 that specifies the requested page path
            response (dict): Dictionary containing the base HTTP response structure to be modified

        Returns:
            dict: Modified response dictionary containing:
                - headers: Dictionary with Content-Type and other HTTP headers
                - response_code: HTTP status code ("200 OK" or "404 Not Found")
                - body: String containing either the page contents or 404 error message

        Raises:
            None

        Note:
            The function assumes the website pages are stored in '<current_directory>/web-server/website/pages/'
        """
        try:
            file_path = f"{os.getcwd()}\\web-server\\website\\pages\\{http_request['path']}"
            if not os.path.exists(file_path):
                response["headers"]["Content-Type"] = "text/html"
                response["response_code"] = "404 Not Found"
                response["body"] = "<h1>404 Not Found</h1>"
            else:
                with open(file_path, "r") as file:
                    response["body"] = file.read()
                    response["response_code"] = "200 OK"
                    response["headers"]["Content-Type"] = "text/html"
            return response
        except Exception as e:
            response["response_code"] = "500 Internal Server Error"
            response["headers"]["Content-Type"] = "text/html"
            response["body"] = json.dumps({"failed": "couldn't fetch page", "message": "couldn't fetch page"})
            return response


    @staticmethod
```

```
    def get_resource(http_request: dict, response) -> dict:
        """
        Get and prepare a resource (file) for HTTP response.
        This function reads a file from the specified path in the HTTP request and prepares
        it for sending in an HTTP response, handling different file types appropriately.
        Args:
            http_request (dict): A dictionary containing HTTP request information.
                Must include 'path' key with the requested resource path.
            response (dict): The response dictionary to be modified.
                Must contain 'response_code', 'headers', and 'body' keys.
        Returns:
            dict: Modified response dictionary containing:
                - response_code: HTTP status code ("200 OK" or "404 Not Found")
                - headers: Dictionary with Content-Type header set based on file type
                - body: File content as bytes, or error message if file not found
        Raises:
            None: Exceptions are not explicitly raised but may occur during file operations.
        Examples:
            >>> response = {"headers": {}}
            >>> request = {"path": "image.png"}
            >>> get_resource(request, response)
            {'response_code': '200 OK', 'headers': {'Content-Type': 'image/png'}, 'body': b'...'}
        """
        try:
            file_type = http_request['path'].split(".")[-1]
            file_path = f"{os.getcwd()}\\web-server\\website\\resources\\{http_request['path']}"
            if not os.path.exists(file_path):
                response["response_code"] = "404 Not Found"
                response["body"] = "<h1>404 Not Found</h1>"
            else:
                with open(file_path, "rb") as file:
                    file_content = file.read()
                    response["response_code"] = "200 OK"

            match file_type:
                case "ico":
                    response["headers"]["Content-Type"] = "image/x-icon"
                    response["body"] = file_content
                case "png":
                    response["headers"]["Content-Type"] = "image/png"
                    response["body"] = file_content
                case "jpg" | "jpeg":
                    response["headers"]["Content-Type"] = "image/jpeg"
                    response["body"] = file_content
                case _:
                    response["headers"]["Content-Type"] = "application/octet-stream"
                    response["body"] = file_content
            return response
        except Exception as e:
            response["response_code"] = "500 Internal Server Error"
```

```
            response["headers"]["Content-Type"] = "text/html"
            response["body"] = json.dumps({"failed": "couldn't fetch resource", "message": "couldn't fetch resource"})
            return response
    @staticmethod
    def get_script(http_request: dict, response) -> dict:
        """
        Handles HTTP requests for script files and prepares the response.
        This function processes HTTP requests for script files, reads the requested file
        if it exists, and sets appropriate response headers based on the file type.
        Args:
            http_request (dict): A dictionary containing HTTP request information.
                Must include a 'path' key with the requested file path.
            response (dict): The response dictionary to be modified.
                Should contain 'response_code', 'headers', and 'body' keys.
        Returns:
            dict: The modified response dictionary containing:
                - response_code: HTTP status code (200 OK or 404 Not Found)
                - headers: Dictionary with Content-Type header
                - body: File content or error message
        Example:
            http_request = {'path': 'script.js'}
            response = {'headers': {}, 'response_code': '', 'body': ''}
            result = get_script(http_request, response)
        """
        try:
            file_type = http_request['path'].split(".")[-1]
            file_path = f"{os.getcwd()}\\web-server\\website\\scripts\\{http_request['path']}"

            if not os.path.exists(file_path):
                response["response_code"] = "404 Not Found"
                response["body"] = "<h1>404 Not Found</h1>"
            else:
                with open(file_path, "rb") as file:
                    file_content = file.read()
                    response["response_code"] = "200 OK"

            match file_type:
                case "js":
                    response["headers"]["Content-Type"] = "application/javascript"
                    response["body"] = file_content
                case _:
                    response["headers"]["Content-Type"] = "application/octet-stream"
                    response["body"] = file_content

            return response
        except Exception as e:
            response["response_code"] = "500 Internal Server Error"
            response["headers"]["Content-Type"] = "text/html"
            response["body"] = json.dumps({"failed": "", "message": "couldn't fetch scripts"})
            return response

```

```
149
150     @staticmethod
151     def get_style(http_request: dict, response) -> dict:
152         """Gets the style file content and sets the appropriate response headers.
153         This function retrieves the content of a style file (.css or .scss) and prepares
154         the HTTP response with the appropriate headers and content type.
155         Args:
156             http_request (dict): A dictionary containing the HTTP request information.
157                                  Must contain a 'path' key with the file path.
158             response (dict): A dictionary that will be populated with the response data.
159                              Should contain 'response_code', 'headers', and 'body' keys.
160         Returns:
161             dict: The response dictionary containing:
162                 - response_code: HTTP status code (200 OK or 404 Not Found)
163                 - headers: Dictionary with Content-Type header
164                 - body: The file content or error message
165         Raises:
166             None: Handles file not found cases by returning 404 response
167         Example:
168             http_request = {'path': 'styles/main.scss'}
169             response = {'headers': {}}
170             result = get_style(http_request, response)
171         """
172         try:
173             file_type = http_request['path'].split(".")[-1]
174             file_path = f"{os.getcwd()}\\web-server\\website\\styles\\
    {http_request['path'].split('.')[0]}.scss"
175
176
177
178             if not os.path.exists(file_path):
179                 response["response_code"] = "404 Not Found"
180                 response["headers"]["Content-Type"] = "text/html"
181                 response["body"] = "<h1>404 Not Found</h1>"
182                 return response
183             else:
184                 with open(file_path, "r") as file:
185                     file_content = file.read()
186                     response["response_code"] = "200 OK"
187
188             match file_type:
189                 case "css":
190                     response["headers"]["Content-Type"] = "text/css"
191                     response["body"] = file_content
192                 case "scss":
193                     css_content = sass.compile(string=file_content)
194                     response["headers"]["Content-Type"] = "text/x-scss"
195                     response["body"] = css_content
196                 case _:
197                     response["headers"]["Content-Type"] = "application/octet-stream"
198                     response["body"] = file_content
199             return response
200         except Exception as e:
201             response["response_code"] = "500 Internal Server Error"
```

```
            response["headers"]["Content-Type"] = "text/html"
            response["body"] = json.dumps({"failed": "couldn't fetch resource", "message": "couldn't fetch styles"})
            return response
```

web-server\utils\Files.py

```python
import json
from database.db import DB
import os
from Crypto.Cipher import AES
import zlib
import base64
import hashlib
class Files:

    @staticmethod
    def create_folder(info, response):
        """
        Creates a new folder entry in the database based on the provided information.
        Args:
            info (dict): A dictionary containing request information, including:
                - "body": JSON string with 'folder_name', 'parent_id', and 'server_key'.
                - "cookies": List of tuples representing cookies, used to extract 'auth_cookie'.
            response (dict): A dictionary to be populated with the response body, headers, and response code.
        Returns:
            dict: The updated response dictionary with appropriate status, headers, and body.
        Raises:
            Exception: If any error occurs during folder creation or database access.
        Response Codes:
            201 CREATED: Folder was successfully created.
            400: Missing required folder information in the request.
            500: An internal error occurred during folder creation.
        """

        try:
            database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
            folder_name = json.loads(info["body"])['folder_name']
            parent_id = json.loads(info["body"])['parent_id']
            server_key = json.loads(info["body"])['server_key']


            if not (folder_name and server_key and parent_id):
                response["body"] = json.dumps({"failed": "missing folder info"})
                response["response_code"] = "400"
                return response


            auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
            database_access.add_file(auth_cookie_value, folder_name, parent_id, server_key, 0, 0, 0)

            response["body"] = json.dumps({"success": "folder created "})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "201 CREATED"
```

```
48              return response
49
50          except Exception as e:
51              response["body"] = json.dumps({"failed": "couldn't upload file",
52              "message": "problem uploading file info"})
53              response["headers"] = {"Content-Type": "application/json"}
54              response["response_code"] = "500"
55              return response
56
57      @staticmethod
58      def folder_content(info, response):
59          """
60          Retrieves user data from the database based on authentication cookie.
61          Args:
62              info (dict): Dictionary containing request information including cookies
63              response (dict): Dictionary to store response data
64          Returns:
65              dict: Modified response dictionary containing:
66                  - 'body': JSON string with user information or error message
67                  - 'response_code': HTTP status code ('200' for success, '500' for error)
68          Raises:
69              Exception: If database access fails or user information cannot be retrieved
70          Note:
71              Expects 'auth_cookie' to be present in the cookies list within info
    dictionary.
72              Uses DB class to interface with the database located at 'web-
    server/database/data'.
73          """
74          auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] ==
    "auth_cookie"), None)[1]
75          database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
76          try:
77              parent = info["query_params"]["parent"]
78          except:
79              parent = None
80
81          try:
82              response["body"] =
    json.dumps(database_access.get_folders_summary(auth_cookie_value, parent_id=parent))
83              response["headers"] = {"Content-Type": "application/json"}
84              response["response_code"] = "200"
85          except Exception as e:
86              response["body"] = json.dumps({"failed": "couldnt fetch file summery",
87              "message": "problem fetching file summery"})
88              response["headers"] = {"Content-Type": "application/json"}
89              response["response_code"] = "500"
90
91
92          return response
93
94
95      @staticmethod
96      def delete_file(info, response):
97          """
```

```
        Deletes a file from the server based on the provided authentication and key.
        Args:
            info (dict): A dictionary containing request information.
                - "query_params" (dict): Contains the query parameters, including:
                    - "key" (str): The unique key identifying the file to be deleted.
                - "cookies" (list): A list of cookies, where each cookie is a tuple
                (cookie_name, cookie_value). The "auth_cookie" is required for authentication.
            response (dict): A dictionary to store the HTTP response.
                - "body" (str): The response body, which will be updated with the result of the operation.
                - "response_code" (str): The HTTP response code, which will indicate success or failure.
        Returns:
            dict: The updated response dictionary containing the result of the file deletion operation.
        Action:
            - Authenticates the user using the "auth_cookie" from the cookies.
            - Deletes the file identified by the "key" from the database.
            - Updates the response with a success message and a "200 OK" status code if the operation succeeds.
            - If an error occurs, updates the response with an error message and a "505 OK" status code,
            and logs the error to the console.
        """

        database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
        server_key = info["query_params"]["key"]

        try:
            auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
            database_access.remove_file(auth_cookie_value, server_key)

            response["body"] = json.dumps({"success": "this file just got deleted"})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "200 OK"
            return response

        except Exception as e:
            response["body"] = json.dumps({"failed": "encountered an error while deleting a file."})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "505 OK"
            return response



    @staticmethod
    def unlock_file(info, response):
        """
            Unlocks an encrypted file associated with a user and generates a temporary shareable download link.

            Args:
```

```
144                 info (dict): A dictionary containing request data, including cookies and
    request body.
145                 response (dict): A dictionary to store the HTTP response body, headers,
    and code.
146 
147             Returns:
148                 dict: The updated response dictionary containing either the download link
    and cookie information,
149                     or an error message and status code.
150         """
151         #helper functions
152         def evp_kdf(password, salt, key_size=32, iv_size=16):
153             """
154             Derives a key and IV from a password and salt using OpenSSL-compatible
    EVP_BytesToKey method (MD5-based).
155 
156             Args:
157                 password (bytes): The password used for key derivation.
158                 salt (bytes): The salt used in derivation (8 bytes).
159                 key_size (int): Desired length of the key in bytes. Default is 32.
160                 iv_size (int): Desired length of the IV in bytes. Default is 16.
161 
162             Returns:
163                 tuple: A tuple containing the derived key and IV.
164             """
165             d = b''
166             while len(d) < key_size + iv_size:
167                 d_i = hashlib.md5(d[-16:] + password + salt) if d else
    hashlib.md5(password + salt)
168                 d += d_i.digest()
169             return d[:key_size], d[key_size:key_size+iv_size]
170 
171         def pad(data: bytes, block_size=16):
172             """
173             Pads the input data to a multiple of the block size using PKCS7 padding.
174 
175             Args:
176                 data (bytes): The data to be padded.
177                 block_size (int): The block size to pad to. Default is 16 bytes.
178 
179             Returns:
180                 bytes: The padded data.
181             """
182             pad_len = block_size - (len(data) % block_size)
183             return data + bytes([pad_len] * pad_len)
184 
185         def unpad(data: bytes):
186             """
187             Removes PKCS7 padding from the data.
188 
189             Args:
190                 data (bytes): The padded data.
191 
192             Returns:
```

```
            bytes: The unpadded original data.
        """
        pad_len = data[-1]
        return data[:-pad_len]

    def decrypt_string(encrypted_b64: str, password: str) -> str:
        """
        Decrypts a base64-encoded AES-encrypted string using a password and salt.

        The encryption must have used OpenSSL-compatible format with "Salted__" header.

        Args:
            encrypted_b64 (str): The base64-encoded encrypted string.
            password (str): The password used for decryption.

        Returns:
            str: The decrypted string.
        """
        encrypted = base64.b64decode(encrypted_b64)
        assert encrypted[:8] == b"Salted__", "Invalid header"
        salt = encrypted[8:16]
        ciphertext = encrypted[16:]
        key, iv = evp_kdf(password.encode(), salt)
        cipher = AES.new(key, AES.MODE_CBC, iv)
        decrypted_padded = cipher.decrypt(ciphertext)
        decrypted = unpad(decrypted_padded)
        return decrypted.decode('utf-8')

    def decrypt_and_decompress_chunk(encrypted_chunk: str, key: str) -> bytes:
        """
        Decrypts a single base64-encoded chunk of data using AES and decompresses it using zlib.

        The chunk must have been encrypted using OpenSSL-compatible AES with "Salted__" format and
        then double base64-encoded (first for encryption, then after compression).

        Args:
            encrypted_chunk (str): The encrypted and base64-encoded string chunk.
            key (str): The password used for AES decryption.

        Returns:
            bytes: The decompressed binary data of the original chunk.

        Raises:
            Exception: If decryption or decompression fails.
        """
        try:
            encrypted = base64.b64decode(encrypted_chunk)

            if encrypted[:8] != b"Salted__":
                raise ValueError("Missing OpenSSL salt header")

```

```
                salt = encrypted[8:16]
                ciphertext = encrypted[16:]

                key_bytes, iv = evp_kdf(key.encode('utf-8'), salt)

                cipher = AES.new(key_bytes, AES.MODE_CBC, iv)
                decrypted = cipher.decrypt(ciphertext)

                pad_len = decrypted[-1]
                decrypted = decrypted[:-pad_len]

                base64_str = decrypted.decode('utf-8')
                raw_binary = base64.b64decode(base64_str)
                return zlib.decompress(raw_binary)

            except Exception as e:
                raise Exception("Decryption or decompression failed") from e









        #end helper functions
        try:
            auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
            db_path = os.path.join(os.getcwd(), "web-server", "database", "data.sql")
            database_access = DB(db_path)

            user_id = database_access.check_cookie(auth_cookie_value)

            body_data = json.loads(info["body"])
            server_key = body_data["server_key"]
            password = body_data["password"]

            metadata = database_access.get_metadata(server_key, user_id)
            name = metadata[4]

        except Exception as e:
            response["body"] = json.dumps({"failed": "missing info or invalid cookie", "message": str(e)})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "400"
            return response

        try:
            file_path = os.path.join("web-server", "database", "files", user_id, f"{server_key}.txt")
            if not os.path.exists(file_path):
                raise FileNotFoundError("Encrypted file not found")

            with open(file_path, "r", encoding="utf-8") as file:
```

```
                encrypted_chunks = file.read().splitlines()

            decrypted_chunks = []
            for chunk in encrypted_chunks:
                if not chunk.strip():
                    continue
                decrypted_data = decrypt_and_decompress_chunk(chunk.strip(), password)
                decrypted_chunks.append(decrypted_data)

            final_content = b''.join(decrypted_chunks)


            share_cookie = database_access.create_cookie(user_id, "share_cookie")

            decrypted_name = decrypt_string(name, password)
            file_extension = decrypted_name.split('.')[-1]
            temp_file_id = f"{share_cookie[1]}.{file_extension}"
            temp_dir = os.path.join("web-server", "tempdata")
            os.makedirs(temp_dir, exist_ok=True)
            temp_file_path = os.path.join(temp_dir, f"{temp_file_id}")

            with open(temp_file_path, "wb") as temp_file:
                temp_file.write(final_content)


            share_link = f"https://a9035.kcyber.net/share/{temp_file_id}"
            response["headers"]["Content-Type"] = "application/json"
            response["body"] = json.dumps({
                "success": True,
                "share_link": share_link,
                "cookie": {
                    "key": share_cookie[0],
                    "value": share_cookie[1],
                    "expires": share_cookie[2]
                }
            })
            response["response_code"] = "200"

        except Exception as e:
            response["body"] = json.dumps({"failed": "couldn't unlock file", "message":
str(e)})
            response["response_code"] = "500"

        return response


    @staticmethod
    def get_file_content(info, response):
        """
        Retrieves file content from the files that was uploaded in hex format.
        Args:
            info (dict): Dictionary containing request information including cookies
            response (dict): Dictionary to store response data
        Returns:
```

```
            dict: Modified response dictionary containing:
                - 'body': File content (already in hex format) or JSON error message
                - 'response_code': HTTP status code ('200' for success, '500' for error)
        """
        auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
        database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")

        try:
            server_key = info["query_params"]["key"]
            index = info["query_params"]["index"]
        except Exception as e:
            response["body"] = json.dumps({"failed": "couldn't fetch file content", "message": str(e)})
            response["response_code"] = "500"
            return response

        try:
            file_content = database_access.get_file(auth_cookie_value, server_key, int(index))
            response["headers"]["Content-Type"] = "text/plain"
            response["body"] = file_content
            response["response_code"] = "200"

        except Exception as e:
            response["body"] = json.dumps({"failed": "couldn't fetch file content", "message": str(e)})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "500"

        return response


    @staticmethod
    def get_shared_file(info, response):
        """
            Serves a shared file if a valid temporary cookie and file ID are provided.

            This function handles HTTP requests for shared files by:
            - Extracting the cookie and file identifier from the URL path.
            - Verifying the validity of the cookie using the database.
            - Checking if the temporary file exists.
            - Returning the file content as a downloadable attachment if valid.
            - Returning an error response if the file is missing or the cookie is invalid.

            Parameters:
                info (dict): Dictionary containing request data, including the file path (`info["path"]`).
                response (dict): Dictionary to populate with the HTTP response content.

            Returns:
                dict: Updated `response` dictionary with status code, headers,
                and either file content or error details.
        """
```

```
        try:
            temp_file_id = info["path"]
            cookie_part = temp_file_id.split(".")[0]

            db_path = os.path.join(os.getcwd(), "web-server", "database", "data.sql")
            database_access = DB(db_path)
            database_access.check_cookie(cookie_part)

            temp_file_path = os.path.join("web-server", "tempdata", temp_file_id)
            if not os.path.exists(temp_file_path):
                raise FileNotFoundError("Shared file not found or expired")

            with open(temp_file_path, "rb") as f:
                file_data = f.read()

            response["headers"]["Content-Type"] = "application/octet-stream"
            response["headers"]["Content-Disposition"] = f'attachment; filename="
{temp_file_id}"'
            response["body"] = file_data
            response["response_code"] = "200"

        except Exception as e:
            response["headers"]["Content-Type"] = "application/json"
            response["body"] = json.dumps({
                "error": "File not found or invalid link",
                "message": str(e)
            })
            response["response_code"] = "404"

        return response


    @staticmethod
    def upload_chunk(info, response):
        """
        Upload a chunk of data to the server.
        This function handles the upload of a file chunk to the server,
        associating it with a user's authentication
        and a specific server key.
        Args:
            info (dict): A dictionary containing:
                - body (str): JSON string with:
                    - index (int): The chunk index
                    - server_key (str): Unique identifier for the file
                    - content (str): The chunk data
                - cookies (list): List of cookie tuples, must include auth_cookie
            response (dict): The response object to be modified and returned
        Returns:
            dict: Modified response dictionary containing:
                - body (str): JSON string with success/failure message
                - response_code (str): HTTP status code
                Success response includes:
                    - {"success": "your chunk was uploaded "}, "200 OK"
                Failure response includes:
```

```
                    - {"failed": "boohoo "}
        Raises:
            Exception: Handles any errors during the upload process
        """
        try:
            database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
            index = json.loads(info["body"])["index"] + 1
            server_key = json.loads(info["body"])["key"]
            content = json.loads(info["body"])["data"]

            if not(index and server_key and content):
                response["body"] = json.dumps({"failed": "boohoo "})
                return response


            auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
            database_access.upload_chunk(auth_cookie_value, server_key, index, content)

            response["body"] = json.dumps({"success": "your chunk was uploaded "})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "200 OK"
            return response

        except Exception as e:
            response["body"] = json.dumps({"failed": "couldn't upload chunk",
            "message": "problem uploading chunk"})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "500"
            return response


    @staticmethod
    def upload_file_info(info, response):
        """
        Uploads file information to the database.
        This function processes file upload information and stores it in the database.
        It validates the required
        fields and authenticates the user through a cookie before adding the file information.
        Args:
            info (dict): A dictionary containing request information with the following keys:
                - body (str): JSON string containing file details (file_name, server_key, chunk_count, size)
                - cookies (list): List of cookie tuples containing authentication information
            response (dict): A dictionary to store the response information
        Returns:
            dict: Modified response dictionary containing:
                - body (str): JSON string with success/failure message
                - response_code (str): HTTP response code (only on success)
        Raises:
            Exception: If database operation fails or authentication is invalid
```

```
        Example:
            >>> info = {
            ...     "body": '{"file_name": "test.txt", "server_key": "abc123", "chunk_count": 5, "size": 1024}',
            ...     "cookies": [("auth_cookie", "user123")]
            ... }
            >>> response = {}
            >>> upload_file_info(info, response)
            {'body': '{"success": "your file info was uploaded "}', 'response_code': '200 OK'}
        """
        try:
            database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
            file_name = json.loads(info["body"])['file_name']
            parent_id = json.loads(info["body"])['parent_id']
            server_key = json.loads(info["body"])['server_key']
            chunk_count = json.loads(info["body"])['chunk_count']
            size = json.loads(info["body"])['size']

            if not (file_name and server_key and chunk_count and parent_id and size):
                response["body"] = json.dumps({"failed": "missing file info"})
                response["response_code"] = "400"
                return response


            auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
            database_access.add_file(auth_cookie_value, file_name, parent_id, server_key, chunk_count, size)

            response["body"] = json.dumps({"success": "your file info was uploaded "})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "201"
            return response

        except Exception as e:
            response["body"] = json.dumps({"failed": "couldn't upload file",
            "message": "problem uploading file info {" + str(e) + "}"})
            response["headers"] = {"Content-Type": "application/json", }
            response["response_code"] = "403 Forbidden"
            return response

```

web-server\utils\User.py

```python
import json
from database.db import DB
import os

class User:

    @staticmethod
    def admin_info(info, response):
        """
        Fetches and returns admin data based on the authentication cookie.
        Args:
            info (dict): A dictionary containing request information, including cookies.
            response (dict): A dictionary to be populated with the response data.
        Returns:
            dict: The updated response dictionary with the admin data in JSON format,
            appropriate headers, and response code.
        Raises:
            Exception: If there is a problem fetching the admin data from the database,
            returns a 500 response with an error message.
        """

        auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
        database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
        try:
            response["body"] = json.dumps(database_access.get_admin_data(auth_cookie_value))
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "200"
        except Exception as e:
            response["body"] = json.dumps({"failed": "couldnt fetch data","message": "problem fetching data"})
            response["response_code"] = "500"

        return response

    @staticmethod
    def auth(info, response):
        """
        Authenticate user based on auth_cookie from request information.
        This function attempts to verify the authentication cookie from the request information
        and updates the response accordingly. If authentication is successful, it returns a
        success message with 200 status code. If authentication fails, it returns an error
        message with 401 status code.
        Parameters:
            info (str or dict): Request information containing cookies. Can be JSON string or dict.
            response (dict): Response dictionary to be modified based on authentication result.
                             Should contain 'body' and 'response_code' keys.
```

```
        Returns:
            dict: Modified response dictionary with authentication results:
                - On success: {'body': '{"success": "logged in"}', 'response_code': "200 OK"}
                - On failure: {'body': '{"failed": "couldnt authenticate", "message": "<error>"}',
                              'response_code': "401"}
        Raises:
            None: Exceptions are caught and handled within the function.
        """
        try:
            info = json.loads(info)
        except Exception as e:
            pass
        database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
        try:
            auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
            database_access.check_cookie(auth_cookie_value)
            priv = database_access.check_privilages(auth_cookie_value)
            response["body"] = json.dumps({"success": "logged in", "elevation": priv})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "200 OK"
        except Exception as e:
            response["body"] = json.dumps({"failed": "couldnt authenticate", "message": str(e)})
            response["response_code"] = "401"

        return response

    @staticmethod
    def info(info, response):
        """
        Retrieves user data from the database based on authentication cookie.
        Args:
            info (dict): Dictionary containing request information including cookies
            response (dict): Dictionary to store response data
        Returns:
            dict: Modified response dictionary containing:
                - 'body': JSON string with user information or error message
                - 'response_code': HTTP status code ('200' for success, '500' for error)
        Raises:
            Exception: If database access fails or user information cannot be retrieved
        Note:
            Expects 'auth_cookie' to be present in the cookies list within info dictionary.
            Uses DB class to interface with the database located at 'web-server/database/data'.
        """
        auth_cookie_value = next((cookie for cookie in info["cookies"] if cookie[0] == "auth_cookie"), None)[1]
        database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
        try:
            response["body"] = json.dumps(database_access.get_user_info(auth_cookie_value))
```

```
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "200"
        except Exception as e:
            response["body"] = json.dumps({"failed": "couldnt fetch data","message": "problem fetching data"})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "500"

        return response

    @staticmethod
    def login(info, response):
        """
        Handles user login authentication and response management.
        This function processes login information, authenticates the user against the database,
        and prepares the HTTP response accordingly.
        Args:
            info (str): JSON string containing login credentials with "username" and "password" fields
            response (dict): Dictionary to store response data including body, response code, and cookies
        Returns:
            dict: Modified response dictionary containing:
                - body (str): JSON string with success/failure message
                - response_code (str): HTTP status code ("200 OK" for success, "500" for failure)
                - cookies (list): List containing authentication cookie if login successful
        Raises:
            Exception: Any database or authentication errors are caught and included in the error response
        """
        info = json.loads(info)
        database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
        try:
            cookie = database_access.login(info["username"], info["password"])
            priv = database_access.check_privilages(cookie[1])
            response["body"] = json.dumps({"success": "logged in", "elevation": priv })
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "200 OK"
            response["cookies"] = [cookie]
        except Exception as e:
            response["body"] = json.dumps({"failed": "couldnt confirm login attempt",
            "message": "error logging in"})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "500"

        return response

    @staticmethod
    def signup(info, response):
        """
        Handles user signup requests by creating a new user account in the database.
```

```
        Args:
            info (str): JSON string containing user signup information with fields:
                - username: desired username for new account
                - password: password for new account
            response (dict): Dictionary to store response information with fields:
                - body: Response message
                - response_code: HTTP response code
                - cookies: List of cookies to set
        Returns:
            dict: Updated response dictionary containing:
                - body: JSON string with success/failure message
                - response_code: "200 OK" on success, "500" on failure
                - cookies: List containing session cookie on success
        Raises:
            Exception: If account creation fails, exception details included in response
        """
        info = json.loads(info)
        database_access = DB(os.getcwd() + "\\web-server\\database\\data.sql")
        try:
            cookie = database_access.add_user(info["username"], info["password"])
            response["body"] = json.dumps({"success": "your account was created
successfully "})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "200 OK"
            response["cookies"] = [cookie]
        except Exception as e:
            response["body"] = json.dumps({"failed": "couldnt create account",
            "message": "problem creating account, try logging in instead. Error: " +
str(e)})
            response["headers"] = {"Content-Type": "application/json"}
            response["response_code"] = "500"

        return response
```

web-server\database\db.py

```python
import os
import sqlite3
import time
import uuid
from datetime import datetime, timedelta
from os import makedirs
from hashlib import sha256


class DB:

    MAX_STORAGE_BYTES = 20 * 1024 * 1024 * 1024  # 20 GB

    def __init__(self, db_path: str) -> None:
        self.db_connection = sqlite3.connect(db_path)
        self.cursor = self.db_connection.cursor()
        self.check_tables()

    def check_tables(self) -> None:
        """
            Ensures all required database tables exist by creating them if they do not.

            This method defines and executes SQL statements to create the following tables:

            Tables:
                - `users`: Stores user credentials and usage statistics.
                    Columns:
                        - `id` (TEXT, PRIMARY KEY): Unique user identifier.
                        - `username` (TEXT, NOT NULL): Username of the account.
                        - `password` (TEXT, NOT NULL): Hashed user password.
                        - `creation_time` (TEXT, NOT NULL): Timestamp of account creation.
                        - `data_uploaded` (INTEGER, NOT NULL): Bytes uploaded by the user.
                        - `data_downloaded` (INTEGER, NOT NULL): Bytes downloaded by the user.

                - `files`: Stores metadata for uploaded files and folders.
                    Columns:
                        - `id` (INTEGER, PRIMARY KEY AUTOINCREMENT): Unique file ID.
                        - `server_key` (TEXT, NOT NULL): Unique server-side identifier for the file.
                        - `size` (INTEGER): File size in bytes.
                        - `created` (TEXT, NOT NULL): Timestamp of file creation.
                        - `file_name` (TEXT, NOT NULL): Original name of the file.
                        - `chunk_count` (INTEGER, NOT NULL): Number of chunks stored.
                        - `owner_id` (INTEGER, NOT NULL): ID of the file owner (foreign key to `users.id`).
                        - `parent_id` (TEXT): ID of the parent folder, if applicable.
                        - `type` (INTEGER): Type of entry (e.g., file or folder).
                        - `status` (INTEGER): File status (e.g., active or deleted).

                - `cookies`: Stores session data for user authentication.
```

```
                    Columns:
                        - `id` (INTEGER, PRIMARY KEY AUTOINCREMENT): Unique cookie ID.
                        - `key` (TEXT, NOT NULL): Session cookie key.
                        - `value` (TEXT, NOT NULL): Session cookie value.
                        - `expiration` (TEXT, NOT NULL): Expiration timestamp.
                        - `owner_id` (TEXT, NOT NULL): Associated user ID (foreign key to `users.id`).

            After execution, all changes are committed to the database.
        """

        users_table_check_sql = """
        CREATE TABLE IF NOT EXISTS users (
            id TEXT PRIMARY KEY NOT NULL UNIQUE,
            username TEXT NOT NULL,
            password TEXT NOT NULL,
            creation_time TEXT NOT NULL,
            data_uploaded INTEGER NOT NULL,
            data_downloaded INTEGER NOT NULL,
            is_admin BOOLEAN DEFAULT 0
        )
        """

        files_table_check_sql = """
                CREATE TABLE IF NOT EXISTS files (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                server_key TEXT NOT NULL,
                size INTEGER,
                created TEXT NOT NULL,
                file_name TEXT NOT NULL,
                chunk_count INTEGER NOT NULL,
                owner_id INTEGER NOT NULL,
                parent_id TEXT,
                type INTEGER,
                status BOOLEAN DEFAULT 0,
                FOREIGN KEY (owner_id) REFERENCES users(id)
            )
            """

        cookie_table_check_sql = """
            CREATE TABLE IF NOT EXISTS cookies (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                key TEXT NOT NULL,
                value TEXT NOT NULL,
                expiration TEXT NOT NULL,
                owner_id TEXT NOT NULL,
                FOREIGN KEY (owner_id) REFERENCES users (id)
            )
            """


        self.cursor.execute(users_table_check_sql)
        self.cursor.execute(files_table_check_sql)
        self.cursor.execute(cookie_table_check_sql)
```

```
102         self.db_connection.commit()
103 
104     def create_cookie(self, user_id: str, _cookie_key: str = "auth_cookie", days: int = 1)
    -> None:
105         """
106             Creates a new authentication cookie for a given user and stores it in the
    database.
107 
108             The method generates a unique cookie value, sets an expiration date of 7 days
    from the
109             current time, and inserts the cookie information into the `cookies` table. If
    an error
110             occurs during the insertion, the transaction is rolled back.
111 
112             Args:
113                 user_id (str): The unique ID of the user for whom the cookie is being
    created.
114 
115             Returns:
116                 tuple: A tuple containing the key, cookie value, and expiration date of
    the created cookie.
117 
118             Raises:
119                 Exception: If there is any issue during the database operation.
120         """
121 
122         create_cookie_sql = """
123         INSERT INTO cookies (key, value, expiration, owner_id)
124         VALUES (?, ?, ?, ?)
125         """
126         try:
127             cookie_value = str(uuid.uuid4())
128             expiration_date = (datetime.now() + timedelta(days=1)).strftime("%Y-%m-%d
    %H:%M:%S")
129             key = _cookie_key
130 
131             self.cursor.execute(create_cookie_sql, (key, cookie_value, expiration_date,
    user_id))
132             self.db_connection.commit()
133             return (key,cookie_value,expiration_date)
134 
135         except Exception as e:
136             self.db_connection.rollback()
137             raise e
138 
139     def get_metadata(self, server_key, user_id) -> None:
140         """
141             Retrieves metadata for a file based on its server key.
142 
143             This method queries the `files` table to get the metadata associated with the
    provided
144             server key. If the file exists, it returns the metadata; otherwise, it raises
    an exception.
145 
146             Args:
```

```
147                 server_key (str): The unique key of the file on the server.
148
149             Returns:
150                 tuple: A tuple containing the file's metadata.
151
152             Raises:
153                 Exception: If the file does not exist or if there is any issue during the
    database operation.
154         """
155
156         get_metadata_sql = """SELECT * FROM files WHERE server_key = ? AND owner_id = ?"""
157         self.cursor.execute(get_metadata_sql, (server_key, user_id))
158         query_result = self.cursor.fetchone()
159
160         if query_result:
161             return query_result
162         else:
163             raise Exception("File not found") from None
164
165     def check_privilages(self, cookie_value: str) -> int:
166         """
167             Checks the privileges of a user based on their session cookie.
168
169             Args:
170                 cookie_value (str): The value of the cookie to be checked.
171
172             Returns:
173                 int: The privilege level of the user (0 for normal user, 1 for admin).
174
175             Raises:
176                 Exception: If the cookie is invalid or has expired.
177         """
178         check_cookie_sql = """SELECT owner_id, expiration FROM cookies WHERE value = ?"""
179         self.cursor.execute(check_cookie_sql, (cookie_value,))
180         query_result = self.cursor.fetchone()
181
182         if query_result:
183             owner_id, expiration = query_result
184             current_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
185
186             if expiration > current_time:
187                 check_user_sql = """SELECT is_admin FROM users WHERE id = ?"""
188                 self.cursor.execute(check_user_sql, (owner_id,))
189                 query_result = self.cursor.fetchone()
190                 if query_result:
191                     is_admin = query_result[0]
192                     if is_admin == 1:
193                         return 1
194                     else:
195                         return 0
196                 else:
197                     raise Exception("User not found") from None
198             else:
199                 raise Exception("Cookie has expired") from None
```

```
        else:
            raise Exception("Invalid cookie") from None


    def get_admin_data(self, cookie_value: str) -> list:
        """
            Retrieves a list of all users from the database.

            This method queries the `users` table to get a list of all users and their associated
            data. It returns a list of tuples, each containing user information.

            Returns:
                list: A list of tuples containing user information.

            Raises:
                sqlite3.Error: If there is an error with the SQLite database operations.
        """
        try:
            if self.check_privilages(cookie_value) == 1:
                get_users_sql = """SELECT id, username, creation_time, data_uploaded FROM users"""
                self.cursor.execute(get_users_sql)
                query_result = self.cursor.fetchall()
                return query_result
            else:
                raise Exception("User is not admin") from None
        except Exception as e:
            raise e

    def check_cookie(self, cookie_value: str) -> str:
        """
            Checks if the provided cookie value exists in the database and is not expired.

            This method retrieves the owner ID and expiration date associated with the given
            cookie value from the `cookies` table. If the cookie exists and has not expired,
            it returns the user ID. If the cookie is expired or invalid, an exception is raised.

            Args:
                cookie_value (str): The value of the cookie to be checked.

            Returns:
                str: The unique ID of the user associated with the valid cookie.

            Raises:
                Exception: If the cookie is invalid or has expired.
        """

        check_cookie_sql = """SELECT owner_id, expiration FROM cookies WHERE value = ?
        """
        self.cursor.execute(check_cookie_sql, (cookie_value,))
```

```
249         query_result = self.cursor.fetchone()
250 
251         if query_result:
252             user_id, expiration = query_result
253             current_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
254 
255             if expiration > current_time:
256                 return user_id
257             else:
258                 raise Exception("Cookie has expired") from None
259         else:
260             raise Exception("Invalid cookie") from None
261 
262     def add_user(self, username: str, password: str) -> None:
263         """
264             Adds a new user to the database and creates an authentication cookie for them.
265 
266             This method checks if a user with the specified username and password already
    exists in the
267             `users` table. If the user does exist, a `Exception` is raised. If not, a new
    user is created
268             with a unique ID, and an authentication cookie is generated and stored in the
    database.
269 
270             Args:
271                 username (str): The username of the user to be added.
272                 password (str): The password for the user to be added.
273 
274             Returns:
275                 tuple: A tuple containing the key, cookie value, and expiration date of
    the created cookie.
276 
277             Raises:
278                 Exception: If the user already exists or if there is any issue during the
    database operation.
279             sqlite3.Error: If there is an SQLite-related error.
280         """
281         user_check_sql = "SELECT 1 FROM users WHERE username = ?"
282         user_adding_sql = "INSERT INTO users (id, username, password, creation_time,
    data_uploaded, data_downloaded) VALUES (?, ?, ?, ?, ?, ?)"
283         try:
284             self.cursor.execute(user_check_sql, (username,))
285             if self.cursor.fetchone():
286                 raise Exception("user already exists") from None
287 
288             user_id = str(uuid.uuid4())
289             hash_str = username + password
290             pass_hash = sha256(hash_str.encode('utf-8')).hexdigest()
291             self.cursor.execute(
292                 user_adding_sql,
293                 (
294                     user_id,
295                     username,
296                     pass_hash,
```

```
                    datetime.now().strftime("%d/%m/%Y %H:%M"),
                    0,
                    0,
                ),
            )

            cookie = self.create_cookie(user_id)

            self.db_connection.commit()
            return cookie

        except sqlite3.Error as e:
            self.db_connection.rollback()
            raise e
        except Exception as e:
            self.db_connection.rollback()
            raise e

    def login(self, username: str, password: str) -> str:
        """
            Authenticates a user and generates an authentication cookie upon successful login.

            This method checks if the provided username and password match an existing user in the
            `users` table. If the credentials are correct, it creates a new authentication cookie
            and returns it. If the credentials are incorrect, an exception is raised.

            Args:
                username (str): The username of the user attempting to log in.
                password (str): The password of the user attempting to log in.

            Returns:
                str: A tuple containing the key, cookie value, and expiration date of the created cookie.

            Raises:
                Exception: If the login information is incorrect or if there is any issue during the
                                database operation.
                sqlite3.Error: If there is an SQLite-related error.
        """
        hash_str = username + password
        pass_hash = sha256(hash_str.encode('utf-8')).hexdigest()
        id_retrieving_sql = "SELECT id FROM users WHERE username = ? AND password = ?"
        try:
            self.cursor.execute(id_retrieving_sql, (username, pass_hash))
            query_result = self.cursor.fetchone()

            if query_result is not None:
                user_id = query_result[0]

                cookie = self.create_cookie(user_id)
```

```
346
347                 return cookie
348             else:
349                 raise Exception("Wrong login information")
350         except sqlite3.Error as e:
351             self.db_connection.rollback()
352             raise e
353         except Exception as e:
354             self.db_connection.rollback()
355             raise e
356
357     def get_user_info(self, cookie_value: str) -> str:
358         """
359             Retrieves user information based on the provided cookie value.
360             Args:
361                 cookie_value (str): The value of the user's cookie.
362             Returns:
363                 dict: A dictionary containing the user's information including:
364                     - username (str): The username of the user.
365                     - creation_time (str): The account creation time.
366                     - uploaded (int): The amount of data uploaded by the user.
367                     - downloaded (int): The amount of data downloaded by the user.
368                     - success (str): A message indicating successful login.
369                     - fileCount (int): The number of files owned by the user.
370             Raises:
371                 Exception: If the user is not found or an unexpected error occurs.
372                 sqlite3.Error: If a database error occurs.
373         """
374
375         try:
376             user_id = self.check_cookie(cookie_value)
377         except Exception as e:
378             raise e
379
380         user_info_sql = """SELECT username, creation_time, data_uploaded, data_downloaded
    FROM users WHERE id = ?"""
381         file_count_sql = """SELECT id FROM files WHERE owner_id = ?"""
382         try:
383             fileCount = len(self.cursor.execute(file_count_sql, (user_id,)).fetchall())
384             self.cursor.execute(user_info_sql, (user_id,))
385             query_result = self.cursor.fetchone()
386
387             if query_result:
388                 user_info = {
389                     "username": query_result[0],
390                     "creation_time": query_result[1],
391                     "uploaded": query_result[2],
392                     "downloaded": query_result[3],
393                     "success": "logged in",
394                     "fileCount": fileCount,
395                 }
396                 return user_info
397             else:
398                 raise Exception("User not found")
```

```
399
400         except sqlite3.Error as e:
401             raise e
402         except Exception as e:
403             raise Exception(f"An unexpected error occurred: {e}") from None
404
405     def add_file(self, cookie_value: str, file_name: str, parent_id: str, server_key: str, chunk_count: int, size: int, type=1) -> None:
406         """
407             Adds a file entry to the database and creates the corresponding file on the server.
408             Args:
409                 cookie_value (str): The cookie value to identify the user.
410                 file_name (str): The name of the file to be added.
411                 parent_name (str): The name of the parent directory.
412                 server_key (str): The server key for the file.
413                 chunk_count (int): The number of chunks the file is divided into.
414                 size (int): The size of the file in bytes.
415                 type (int): The type of the file (1 for folder, 0 for regular file).
416             Raises:
417                 sqlite3.Error: If there is an error with the SQLite database operations.
418                 ValueError: If there is a value error during the process.
419                 Exception: If the user's total data usage would exceed 20GB.
420         """
421         try:
422             user_id = self.check_cookie(cookie_value)
423
424             user_upload_sql = """
425             SELECT data_uploaded FROM users WHERE id = ?
426             """
427             current_uploaded = self.cursor.execute(user_upload_sql, (user_id,)).fetchone()[0]
428             new_total = current_uploaded + size
429
430             if new_total > self.MAX_STORAGE_BYTES:
431                 raise Exception("Upload denied: user storage quota exceeded (20GB limit).")
432
433             file_insertion_sql = """
434             INSERT INTO files (owner_id, file_name,  server_key, chunk_count, size, created, parent_id, type, status)
435             VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
436             """
437             status = 0
438             if type == 0:
439                 status = 1
440
441             self.cursor.execute(
442                 file_insertion_sql,
443                 (
444                     user_id,
445                     file_name,
446                     server_key,
447                     chunk_count,
```

```
                    size,
                    datetime.now().strftime("%d/%m/%Y %H:%M"),
                    parent_id,
                    type,
                    status,
                ),
            )
            makedirs(f"web-server\\database\\files\\{user_id}", exist_ok=True)
            open(f"web-server\\database\\files\\{user_id}\\{server_key}.txt", "wb")

            update_user_upload = """
            UPDATE users SET data_uploaded = ? WHERE id = ?
            """
            self.cursor.execute(update_user_upload, (new_total, user_id))

            self.db_connection.commit()

        except sqlite3.Error as e:
            self.db_connection.rollback()
            raise e
        except ValueError as ve:
            self.db_connection.rollback()
            raise ve
        except Exception as ex:
            self.db_connection.rollback()
            raise ex

    def upload_chunk(self, cookie_value: str, server_key: str, index: int, content: str) -
> None:
        """
            Uploads a chunk of data to the server and updates the file status if all
chunks are uploaded.
            Chunks are processed strictly in order starting from index 0.

            Args:
                cookie_value (str): The cookie value used to identify the user.
                server_key (str): The unique key for the file on the server.
                index (int): The index of the current chunk (0-based).
                content (str): The content of the chunk to be uploaded.
        """
        try:
            user_id = self.check_cookie(cookie_value)
            file_path = f"web-server\\database\\files\\{user_id}\\{server_key}.txt"

            while True:
                try:
                    if index == 0:
                        if (
                            not os.path.exists(file_path)
                            or os.path.getsize(file_path) == 0
                        ):
                            with open(file_path, "w") as file:
                                file.write(content)
                            break
```

```
                    else:
                        with open(file_path, "r") as file:
                            chunk_count = file.read().count("\n") + 1

                        if chunk_count == index:
                            with open(file_path, "a") as file:
                                file.write("\n" + content)
                            break

                    time.sleep(0.05)

                except Exception as e:
                    time.sleep(0.05)
                    continue

            check_file_complete = """
            SELECT chunk_count FROM files WHERE server_key = ?
            """
            count = (
                self.cursor.execute(check_file_complete, (server_key,)).fetchone()[0]
            )
            if index + 1 == count:
                self.cursor.execute(
                    "UPDATE files SET status = 1 WHERE server_key = ?", (server_key,)
                )
                self.db_connection.commit()

        except sqlite3.Error as e:
            self.db_connection.rollback()
            raise e
        except ValueError as ve:
            self.db_connection.rollback()
            raise ve

    def remove_file(self, cookie_value: str, server_key: str) -> None:
        """
            Recursively removes a file or folder from both the database and the
filesystem.

            If the specified server key corresponds to a folder, all its child entries
            will be recursively deleted before removing the folder itself.

            Args:
                cookie_value (str): The session cookie used to identify and authenticate
the user.
                server_key (str): The unique key corresponding to the file or folder to be
deleted.

            Raises:
                ValueError: If the file or folder is not found in the database or file
size is missing.
                sqlite3.Error: If a database error occurs during the process.
        """
        try:
```

```
            user_id = self.check_cookie(cookie_value)

            self.cursor.execute(
                "SELECT id, type FROM files WHERE owner_id = ? AND server_key = ?",
                (user_id, server_key),
            )
            result = self.cursor.fetchone()
            if not result:
                raise ValueError("File or folder not found in database")

            file_id, file_type = result

            if file_type == 0:
                self.cursor.execute(
                    "SELECT server_key FROM files WHERE owner_id = ? AND parent_id = ?",
                    (user_id, server_key),
                )
                children = self.cursor.fetchall()
                for (child_key,) in children:
                    self.remove_file(cookie_value, child_key)

            file_size_result = self.cursor.execute(
                "SELECT size FROM files WHERE owner_id = ? AND server_key = ?",
                (user_id, server_key),
            ).fetchone()

            if file_size_result is None:
                raise ValueError("File size not found in database")

            file_size = file_size_result[0]
            current_data_uploaded = self.cursor.execute(
                "SELECT data_uploaded FROM users WHERE id = ?", (user_id,)
            ).fetchone()[0]

            self.cursor.execute(
                "UPDATE users SET data_uploaded = ? WHERE id = ?",
                (current_data_uploaded - file_size, user_id),
            )

            self.cursor.execute(
                "DELETE FROM files WHERE owner_id = ? AND server_key = ?",
                (user_id, server_key),
            )

            self.db_connection.commit()

            file_path = f"web-server/database/files/{user_id}/{server_key}.txt"
            if os.path.exists(file_path):
                os.remove(file_path)

        except sqlite3.Error as e:
            self.db_connection.rollback()
            raise e
        except ValueError as ve:
```

```
            self.db_connection.rollback()

    def get_folders_summary(self, cookie_value: str, parent_id: str = "-1") -> dict:
        """
            Retrieves a summary of files and folders under a specified parent directory for the authenticated user.

            Args:
                cookie_value (str): The session cookie used to identify and authenticate the user.
                parent_id (str, optional): The ID of the parent folder. Defaults to "-1", which typically refers to the root directory.

            Returns:
                dict: A dictionary mapping each file's unique database ID to a summary containing:
                    - id (int): File's database ID.
                    - server_key (str): Unique server-side key for the file.
                    - file_name (str): Name of the file or folder.
                    - size (int): Size of the file in bytes.
                    - created (str): Timestamp of file creation.
                    - parent_id (str): ID of the parent folder.
                    - type (int): Type of entry (e.g., 0 = folder, 1 = file).
                    - chunk_count (int): Number of chunks the file is divided into.

            Raises:
                sqlite3.Error: If an SQLite-related error occurs.
                Exception: If any other unexpected error occurs during processing.
        """
        try:
            user_id = self.check_cookie(cookie_value)
            get_files_sql = """
            SELECT id, server_key, file_name, size, created, parent_id, type, chunk_count
            FROM files WHERE owner_id = ? AND parent_id = ?
            """

            self.cursor.execute(get_files_sql, (user_id, parent_id))
            rows = self.cursor.fetchall()

            files_summary = {}
            for row in rows:
                file_summary = {
                    'id': row[0],
                    'server_key': row[1],
                    'file_name': row[2],
                    'size': row[3],
                    'created': row[4],
                    'parent_id': row[5],
                    'type': row[6],
                    'chunk_count': row[7]
                }
                files_summary[row[0]] = file_summary
            return files_summary
        except sqlite3.Error as e:
```

```
654             raise e
655         except Exception as e:
656             raise e
657 
658     def get_file(self, cookie_value: str, server_key: str, index: int) -> str:
659         """
660             Retrieves a specific chunk from a file associated with a user and updates the user's data usage.
661             Args:
662                 cookie_value (str): The cookie value used to authenticate the user.
663                 server_key (str): The unique key identifying the file on the server.
664                 index (int): The zero-based index of the line to retrieve from the file.
665             Returns:
666                 str: The content of the specified chunk in the file.
667             Raises:
668                 sqlite3.Error: If there is an error with the database operations.
669                 Exception: If the file does not exist or the index is invalid.
670         """
671 
672         try:
673             user_id = self.check_cookie(cookie_value)
674         except sqlite3.Error as e:
675             raise e
676 
677         try:
678             file_path = f"web-server\\database\\files\\{user_id}\\{server_key}.txt"
679             with open(file_path, "r") as file:
680                 file_content = file.read().split("\n")
681 
682             file_size_sql = "SELECT size FROM files WHERE server_key = ?"
683             file_size_result = self.cursor.execute(
684                 file_size_sql, (server_key,)
685             ).fetchone()
686             if file_size_result is None:
687                 raise Exception("File does not exist") from None
688             file_size = file_size_result[0]
689             update_downloaded_sql = (
690                 "UPDATE users SET data_downloaded = data_downloaded + ? WHERE id = ?"
691             )
692             self.cursor.execute(update_downloaded_sql, (file_size, user_id))
693             self.db_connection.commit()
694             return file_content[index + 1]
695         except TypeError:
696             raise Exception("File does not exist") from None
697         except sqlite3.Error as e:
698             self.db_connection.rollback()
699             raise e
700 
```

web-server\file_cleanup.py

```python
import os
import time
from datetime import datetime, timedelta

TEMP_DIR = os.path.join(os.path.dirname(__file__), 'tempdata')
DELETE_OLDER_THAN = timedelta(days=1)
SLEEP_INTERVAL = 3600

def delete_old_files():
    try:
        now = datetime.now()
        for filename in os.listdir(TEMP_DIR):
            file_path = os.path.join(TEMP_DIR, filename)
            if os.path.isfile(file_path):
                try:
                    file_mtime = datetime.fromtimestamp(os.path.getmtime(file_path))
                    if now - file_mtime > DELETE_OLDER_THAN:
                        os.remove(file_path)
                        print(f"Deleted: {file_path}")
                except Exception as e:
                    print(f"Failed to process {file_path}: {e}")
    except FileNotFoundError:
        os.makedirs(TEMP_DIR)

def main():
    while True:
        delete_old_files()
        time.sleep(SLEEP_INTERVAL)

if __name__ == "__main__":
    main()
```

web-server\website\pages\index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="/resources/favicon.ico" type="image/x-icon">
    <link rel="stylesheet" href="/styles/main.css">
    <title>RAFT home</title>
</head>
<body>
    <div class="container">
        <header>
            <h1>Welcome to the RAFT website</h1>
        </header>
        <main>
            <section class="scrollable-section info">
                <h2>About the Project</h2>
                <p>We offer a quick way to save your files to the cloud, this allowes you to never lose your time with unnecessarily bloated sites.
                    you can save any type of file to our system and regain access to it from anywhere at anytime!
                </p>
                <nav>
                    <button id="website-button">Try it now</button>
                </nav>
            </section>
            <section class="scrollable-section links">
                <h3>Contact Us</h3>
                <div class="contact-info">
                    <p>Email: <a href="mailto:st0645995@kshapira.ort.org.il">support-mail</a></p>
                    <p>GitHub: <a href="https://github.com/alonsta/final_project" target="_blank">RAFT-github</a></p>
                    <p>Available on multiple platforms!</p>
                </div>
            </section>
        </main>
        <footer>
            <p>&copy; 2024 RAFT - Real-time Access File Transmition. All rights reserved.</p>
        </footer>
    </div>

    <script src="/scripts/main.js" defer></script>
</body>
</html>
```

web-server\website\scripts\main.js

```
document.getElementById('website-button').addEventListener('click', function() {
    window.location.href = '/pages/login.html';
});

```

web-server\website\pages\login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="icon" href="/resources/favicon.ico" type="image/x-icon">
  <title>Login / Signup</title>
  <link rel="stylesheet" href="/styles/login_signup.css">
</head>
<body>
  <div class="container">
    <div class="link_wrapper">
      <a href="/" id="re">
        &#8592;
      </a>
    </div>
    <h1 id="form-title">Login</h1>
    <form id="auth-form">
      <input type="text" id="username" placeholder="Username" required>
      <input type="password" id="password" placeholder="Password" required>
      <button id="submit_button" type="submit">Submit</button>
    </form>
    <p id="toggle-text">Don't have an account? <a href="#" id="toggle-link">Sign up</a></p>
  </div>

  <script src="/scripts/pako.min.js"></script>
  <script src="/scripts/crypto-js.js"></script>
  <script src="/scripts/chart.js"></script>

  <script src="/scripts/login_signup.js"></script>
  <script src="/scripts/Authentication.js"></script>
</body>
</html>
```

web-server\website\scripts\login_signup.js

```
async function hashString(password, salt) {
  let saltedInput = salt + password;
  let hash = CryptoJS.SHA256(saltedInput);
  return hash.toString(CryptoJS.enc.Hex);
}


document.getElementById("toggle-link").addEventListener("click", function (e) {
  e.preventDefault();

  const formTitle = document.getElementById("form-title");
  const submitButton = document.getElementById("submit_button");
  const toggleLink = document.getElementById("toggle-link");
  const form = document.querySelector("form");

  if (formTitle.textContent === "Login") {
    formTitle.textContent = "Sign Up";
    submitButton.textContent = "Sign Up";
    toggleLink.textContent = "Login";

    const input = document.createElement("input");
    input.setAttribute('type', 'password');
    input.setAttribute('placeholder', 'Re-enter password');
    input.setAttribute('id', 'Re_enter_password');
    input.setAttribute('name', 'Reenter_password');
    input.required = true;

    form.insertBefore(input, submitButton);

    toggleLink.parentElement.firstChild.textContent = "Already have an account? ";

  } else {
    const Re_enterInput = document.getElementById('Re_enter_password');
    if (Re_enterInput) {
      form.removeChild(Re_enterInput);
    }

    formTitle.textContent = "Login";
    submitButton.textContent = "Submit";
    toggleLink.textContent = "Sign Up";
    toggleLink.parentElement.firstChild.textContent = "Don't have an account? ";
  }
});

document.getElementById("auth-form").addEventListener("submit", async function (e) {
  e.preventDefault();
  const formTitle = document.getElementById("form-title");
  const username = sanitizeInput(document.getElementById("username").value);
  let password = sanitizeInput(document.getElementById("password").value);
  const Repassword = sanitizeInput(document.getElementById("Re_enter_password")?.value ||
'');

```

```
52    if (formTitle.textContent === "Sign Up" && Repassword) {
53      if (Repassword === password && password.length >= 8) {
54        fetch("/users/signup", {
55          method: 'POST',
56          headers: {
57            'Content-Type': 'application/json'
58          },
59          body: JSON.stringify({ username, password })
60        }).then(response => response.json())
61          .then(data => {
62            if (data.success) {
63              document.cookie = "pass=" + password +"; path=/; max-age=86400; secure;
    samesite=strict";
64              window.location.href = "/pages/dashboard.html";
65            } else {
66              alert(data.message);
67            }
68          });
69      } else if (password.length >= 8) {
70        alert("Passwords do not match");
71      } else {
72        alert("Password must be at least 8 characters long");
73      }
74    } else if (username && password) {
75      fetch("/users/login", {
76        method: 'POST',
77        headers: {
78          'Content-Type': 'application/json'
79        },
80        body: JSON.stringify({ username, password })
81      })
82      .then(response => response.json())
83      .then(data => {
84        if (data.success) {
85          if(data.elevation == 0){
86          document.cookie = "pass=" + password +"; path=/; max-age=86400; secure;
    samesite=strict";
87          window.location.href = "/pages/dashboard.html";
88          } else if(data.elevation == 1){
89            window.location.href = "/pages/admin.html";
90          }
91        } else {
92          alert(data.message);
93          new Promise(resolve => setTimeout(resolve, 1000));
94          alert("You have been locked out for 5 seconds due to failed login attempt.");
95          // Wait for 10 seconds before allowing another attempt
96          new Promise(resolve => setTimeout(resolve, 10000));
97          alert("You can try again now.");
98        }
99      })
100     .catch(error => {
101       console.error("Error:", error);
102     });
103   } else {
```

```
    alert("Please fill out both fields.");
  }
});

function sanitizeInput(input) {
  const div = document.createElement('div');
  div.appendChild(document.createTextNode(input));
  return div.innerHTML;
}
```

web-server\website\scripts\Authentication.js

```
async function checkAuthentication() {
    try {
        const response = await fetch('/auth', {
            method: 'GET',
            credentials: 'include'
        });

        if (response.status === 401 && window.location.pathname !== '/pages/login.html') {
            window.location.href = '/pages/login.html';
        } else if (response.ok) {
            const data = await response.json();

            if (window.location.pathname === '/pages/login.html') {
                if (data.elevation === 0) {
                    window.location.href = '/pages/dashboard.html';
                } else if (data.elevation === 1) {
                    window.location.href = '/pages/admin.html';
                }
            }
        } else {
            console.error("An error occurred:", response.status);
        }
    } catch (error) {
        console.error("Network or server error:", error);
    }
}

checkAuthentication();
setInterval(() => {
    checkAuthentication();
}, 100000);
```

web-server\website\pages\admin.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Admin - User Dashboard</title>
  <style>
    body {
      margin: 0;
      font-family: Arial, sans-serif;
      background-color: #f0fff0; /* Light greenish background */
      color: #333;
    }

    header {
      background-color: #2e8b57; /* Sea green */
      color: white;
      padding: 20px;
      display: flex;
      justify-content: space-between;
      align-items: center;
    }

    header h1 {
      margin: 0;
    }

    .logout-btn {
      background-color: #dc3545; /* Bootstrap red */
      border: none;
      color: white;
      padding: 10px 20px;
      cursor: pointer;
      border-radius: 4px;
      font-weight: bold;
    }

    .logout-btn:hover {
      background-color: #c82333;
    }

    .container {
      padding: 30px;
    }

    table {
      width: 100%;
      border-collapse: collapse;
      margin-top: 20px;
    }

```

```
    th, td {
      padding: 12px;
      text-align: left;
      border-bottom: 1px solid #ddd;
    }

    th {
      background-color: #228b22;
      color: white;
    }

    tr:hover {
      background-color: #f1f1f1;
    }
  </style>
</head>
<body>
  <header>
    <h1>Admin Panel</h1>
    <button class="logout-btn" id="logoutBtn">Logout</button>
  </header>

  <div class="container">
    <h2>User List</h2>
    <table>
      <thead>
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>creation time</th>
          <th>storage used</th>
        </tr>
      </thead>
      <tbody id="user-list">

      </tbody>
    </table>
  </div>
  <script src ="/scripts/Admin.js"></script>
  <script src ="/scripts/Authentication.js"></script>
</body>
</html>

```

web-server\website\scripts\Admin.js

```
function formatFileSize(bytes) {
  if (bytes === 0) return '0 Bytes';

  const units = ['Bytes', 'KB', 'MB', 'GB', 'TB'];
  const unitIndex = Math.floor(Math.log(bytes) / Math.log(1024));
  const size = (bytes / Math.pow(1024, unitIndex)).toFixed(2);

  return `${size} ${units[unitIndex]}`;
}

async function appendUserRow(tuple) {
  const tableBody = document.getElementById("user-list");

  if (!Array.isArray(tuple) || tuple.length !== 4) {
    console.error("Invalid tuple data. Expected 4 elements.");
    return;
  }

  const tr = document.createElement("tr");

  tuple.forEach((data, index) => {
    const td = document.createElement("td");

    // Format the 4th element (index 3) with file size formatter
    if (index === 3) {
      td.textContent = formatFileSize(Number(data));
    } else {
      td.textContent = data;
    }

    tr.appendChild(td);
  });

  tableBody.appendChild(tr);
}

async function fetchUsers() {
  try {
    const response = await fetch("/users/admin", {
      method: 'GET',
      credentials: 'include'
    });

    if (response.ok) {
      const data = await response.json();

      const tableBody = document.getElementById("user-list");
      tableBody.innerHTML = ''; // 💥 Clear before appending

      data.forEach(user => {
        appendUserRow(user);
```

```
      });
    } else {
      console.error("Failed to fetch users:", response.status);
    }
  } catch (error) {
    console.error("Network error:", error);
  }
}

document.addEventListener("DOMContentLoaded", () => {
    fetchUsers();
    setInterval(fetchUsers, 10000);
});

document.getElementById("logoutBtn").addEventListener("click", () => {
  // Clear localStorage and sessionStorage
  localStorage.clear();
  sessionStorage.clear();

  // Clear all cookies
  document.cookie.split(";").forEach(cookie => {
    const eqPos = cookie.indexOf("=");
    const name = eqPos > -1 ? cookie.substring(0, eqPos) : cookie;
    document.cookie = name + "=;expires=Thu, 01 Jan 1970 00:00:00 GMT;path=/";
  });

  // Reload the page
  location.reload();
});
```

web-server\website\pages\dashboard.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="/resources/favicon.ico" type="image/x-icon">
    <link rel="stylesheet" href="/styles/dashboard.css">
    <title>Dashboard</title>
</head>
<body>
    <div id="progress-indicator" class="progress-container" style="display: none;">
        <div class="progress-bar"></div>
        <span class="progress-text">Processing...</span>
    </div>

    <div class="dashboard">
        <div class="sidebar">
            <div class="tab" id="overview-tab" onclick="switchTab('overview');">
                <img src="/resources/chart-icon.png" alt="Overview">
            </div>
            <div class="tab" id="files-tab" onclick="switchTab('files')">
                <img src="/resources/file-icon.png" alt="Files">
            </div>
            <div class="tab" id="account-tab" onclick="switchTab('account')">
                <img src="/resources/gear-icon.png" alt="Account">
            </div>
        </div>
        <div class="content">
            <div id="overview" class="content-section hidden" style="padding-bottom:
10px;">
                <canvas id="statsChart" style="max-width: 90%; max-height: 90%;"></canvas>
                <div id="file_count"></div>
            </div>
            <div id="files" class="content-section hidden">
                <input type="file" id="file" style="display: none;" multiple>
                <div id="path_container" style="font-size: 20px; font-weight: bold;">

                    <h1 id="folder_path" current-id="-1" last-id ="-1" style="max-width:
calc(100% - 100px);">
                        <button id="go_back_btn" style="font-size: 20px; cursor: pointer;
background: none; border: none; margin-right: 10px;"><</button>
                        <span id="path_display" style="
                            font-family: monospace, sans-serif;
                            padding: 0.2em 0.4em;
                            border-radius: 5px;
                            white-space: nowrap;
                            overflow: hidden;
                            text-overflow: ellipsis;
                            display: inline-block;
                            max-width: calc(100% - 200px);
                            vertical-align: middle;">
                            Myfiles
```

```
                        </span>

                        <div id="bubble"></div>
                        <button id="create_folder_btn" style="font-size: 24px; cursor: pointer; background: none; border: none; font-weight: bold; color:rgb(154, 226, 226)">_</button>
                    </h1>

                    <div id="folder_popup" style="
                        display: none;
                        position: fixed;
                        background: white;
                        border: 1px solid #ccc;
                        padding: 8px;
                        border-radius: 6px;
                        box-shadow: 0 2px 8px rgba(0,0,0,0.2);
                        z-index: 999;">
                        <input type="text" id="folder_name_input" placeholder="Enter folder name" style="padding: 4px 8px; font-size: 16px;">
                    </div>
                </div>



                <div id = "files_grid" class="file-grid-container" style="    display: grid; grid-template-columns: repeat(auto-fill, minmax(200px, 1fr)); gap: 10px;">
                    </div>

            </div>
            <div id="account" class="content-section hidden">
                <button id="logout-btn">Logout</button>

            </div>
        </div>

    </div>

    <script src="/scripts/pako.min.js"></script>
    <script src="/scripts/crypto-js.js"></script>
    <script src="/scripts/chart.js"></script>

    <script src ="/scripts/Authentication.js"></script>
    <script src="/scripts/dashboard.js"></script>
    <script src="/scripts/upload_file.js"></script>
</body>
</html>
```

web-server\website\scripts\dashboard.js

```
const storedPassword = getCookie("pass");
let folderIdStack = [];
let currentLoadToken = null; // Token to track the current load operation
let backButtonCooldown = false;
const progressIndicator = document.getElementById('progress-indicator');
const progressBar = progressIndicator.querySelector('.progress-bar');
const progressText = progressIndicator.querySelector('.progress-text');

const createBtn = document.getElementById('create_folder_btn');
const popup = document.getElementById('folder_popup');
const input = document.getElementById('folder_name_input');

function formatFileSize(bytes) {
  if (bytes === 0) return '0 Bytes';

  const units = ['Bytes', 'KB', 'MB', 'GB', 'TB'];
  const unitIndex = Math.floor(Math.log(bytes) / Math.log(1024));
  const size = (bytes / Math.pow(1024, unitIndex)).toFixed(2);

  return `${size} ${units[unitIndex]}`;
}

document.getElementById('go_back_btn').addEventListener('click', () => {
  if (backButtonCooldown) return;

  backButtonCooldown = true;
  setTimeout(() => {
    backButtonCooldown = false;
  }, 300); // 0.3 second cooldown

  // Cancel all pending file loads
  currentLoadToken = null;

  if (folderIdStack.length > 0) {
    const previousId = folderIdStack.pop();

    document.getElementById('folder_path').setAttribute('current-id', previousId);
    document.getElementById('folder_path').setAttribute('last-id',
folderIdStack[folderIdStack.length - 1] || "-1");

    let currentPath = document.getElementById('path_display').textContent;
    let pathParts = currentPath.split('/');
    pathParts.pop(); // remove last
    document.getElementById('path_display').textContent = pathParts.join('/');

    loadUserFiles(); // safe because prior load is now cancelled
  }
});

createBtn.addEventListener('click', (e) => {
  e.stopPropagation();

```

```
  // Get button position
  const rect = createBtn.getBoundingClientRect();
  const offsetX = 10;
  const offsetY = 5;

  // Position popup next to the button
  popup.style.left = `${rect.right + offsetX}px`;
  popup.style.top = `${rect.top + offsetY}px`;
  popup.style.display = 'block';

  input.value = '';
  input.focus();
});

document.addEventListener('click', (e) => {
  if (!popup.contains(e.target) && e.target !== createBtn) {
    popup.style.display = 'none';
  }
});

input.addEventListener('keydown', (e) => {
  if (e.key === 'Enter') {
    const folderName = input.value.trim();
    let parentId = document.getElementById('folder_path').getAttribute('current-id');
    if (folderName) {
      //here look for the parent id of the folder from the parent div. also create an id for the folder and encrypt it with the secret sauce
      createFolder(folderName, parentId);
      popup.style.display = 'none';
    }
  }
});

async function createFolder(name, parentId) {
  if(name.length > 32)
    {alert("Folder name too long!"); return;}

  let server_key = generateRandomId();
  let password = getCookie("pass");
  let key = generateEncryptionKey(password, server_key);
  let encryptedFolderName = CryptoJS.AES.encrypt(CryptoJS.enc.Utf8.parse(name), key).toString();

  let response = await fetch('/files/create/folder', {
    method: 'POST',
    body: JSON.stringify({
      server_key: server_key,
      folder_name: encryptedFolderName,
      parent_id: parentId
    })
  });

  if (response.ok) {
    console.log('Folder created successfully!');
```

```
104
105     // Stack push
106     folderIdStack.push(parentId);
107
108     document.getElementById('folder_path').setAttribute('last-id', parentId);
109     document.getElementById('folder_path').setAttribute('current-id', server_key);
110     document.getElementById('path_display').textContent += "/" + name;
111     loadUserFiles();
112   }
113 }
114 // --- Helper function to show/update/hide indicator ---
115 function updateProgress(state, message, percentage = null, isError = false, isDownload =
    false) {
116     if (state === 'show') {
117         progressIndicator.classList.remove('error', 'download'); // Reset classes
118         if(isError) progressIndicator.classList.add('error');
119         if(isDownload) progressIndicator.classList.add('download');
120
121         progressText.textContent = message;
122         progressBar.style.width = (percentage !== null) ? `${percentage}%` : '0%';
123         progressIndicator.style.display = 'block';
124         progressIndicator.style.opacity = '1';
125     } else if (state === 'update') {
126          progressText.textContent = message;
127          if (percentage !== null) {
128              progressBar.style.width = `${percentage}%`;
129          }
130     } else if (state === 'hide') {
131         progressIndicator.style.opacity = '0';
132         // Wait for fade out before hiding completely
133         setTimeout(() => {
134             progressIndicator.style.display = 'none';
135             progressIndicator.classList.remove('error', 'download'); // Clean up classes
136         }, 500); // Matches CSS transition duration
137     }
138 }
139
140 // --- Add this to your existing event listeners ---
141 document.addEventListener('DOMContentLoaded', () => {
142   switchTab('overview'); // Default tab on load
143   loadUserStats();
144   loadUserFiles();
145
146   setInterval(() => {
147       loadUserFiles();
148       loadUserStats();
149     }, 30000); // Refresh every 15 seconds
150   });
151
152
153 /**
154  * Switches between different content sections
155  * @param {string} tabName - The ID of the tab to switch to
156  */
```

```
157 function switchTab(tabName) {
158   const sections = document.querySelectorAll('.content-section');
159   sections.forEach(section => section.classList.add('hidden'));
160   const tabs = document.querySelectorAll('.sidebar .tab');
161   tabs.forEach(tab => tab.classList.remove('active'));
162   const activeSection = document.getElementById(tabName);
163   const activeTab = document.getElementById(tabName + '-tab');
164   if (activeSection) {
165     activeSection.classList.remove('hidden');
166     activeTab.classList.add('active');
167   }
168 }
169
170 document.getElementById('logout-btn').addEventListener('click', () => {
171
172   localStorage.clear();
173   sessionStorage.clear();
174
175
176   document.cookie.split(";").forEach(cookie => {
177     const eqPos = cookie.indexOf("=");
178     const name = eqPos > -1 ? cookie.substr(0, eqPos) : cookie;
179     document.cookie = `${name}=;expires=Thu, 01 Jan 1970 00:00:00 GMT;path=/`;
180   });
181   location.reload();
182 });
183
184 /**
185  * Loads and displays user statistics
186  */
187 let userChart = null; // Store the Chart instance globally
188
189 async function loadUserStats() {
190   function waitForElement(id, timeout = 3000) {
191     return new Promise((resolve, reject) => {
192       const start = Date.now();
193       (function check() {
194         const el = document.getElementById(id);
195         if (el) return resolve(el);
196         if (Date.now() - start > timeout) return reject(`Element #${id} not found`);
197         requestAnimationFrame(check);
198       })();
199     });
200   }
201
202   const overviewDiv = await waitForElement('overview');
203   const fileCountDiv = await waitForElement('file_count');
204
205   if (!overviewDiv || !fileCountDiv) {
206     console.error('Required DOM elements not found');
207     showBubble("❌ Failed to load user stats. Please try again later.");
208     return;
209   }
210
```

```
  // Clear and recreate layout
  overviewDiv.innerHTML = `
    <div id="overviewStats"></div>
    <canvas id="statsChart" style="max-width: 600px; margin: 30px auto; display: block;"></canvas>
    <div id="overviewError" style="color: red; text-align: center; margin-top: 10px;"></div>
  `;

  const canvas = document.getElementById('statsChart');
  const ctx = canvas?.getContext('2d');
  const statsDiv = document.getElementById('overviewStats');
  const errorDiv = document.getElementById('overviewError');
  fileCountDiv.innerHTML = '';
  errorDiv.innerHTML = '';

  if (!ctx || !statsDiv) {
    console.error('Failed to prepare layout for stats');
    errorDiv.textContent = 'Internal layout error. Please reload.';
    return;
  }

  // Reset previous chart if exists
  if (userChart) {
    userChart.destroy();
    userChart = null;
  }

  try {
    const response = await fetch('/users/info', {
      credentials: 'include',
      method: 'GET'
    });

    if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);

    const stats = await response.json();
    let { username, uploaded, downloaded, fileCount, creation_time } = stats;

    let uploadeds = formatFileSize(uploaded);
    let downloadeds = formatFileSize(downloaded);
    function formatCustomDate(dateStr) {
  // Expects "DD/MM/YYYY HH:mm"
      const [datePart, timePart] = dateStr.split(' ');
      const [day, month, year] = datePart.split('/').map(Number);
      const [hours, minutes] = timePart.split(':').map(Number);

      const dateObj = new Date(year, month - 1, day, hours, minutes);
      return dateObj.toLocaleString(undefined, {
        year: 'numeric',
        month: 'short',
        day: 'numeric',
        hour: '2-digit',
        minute: '2-digit'
```

```
      });
    }

    let accountAge = formatCustomDate(creation_time);

    statsDiv.innerHTML = `
      <h2 style="text-align:center;">Welcome back, <strong>${username}</strong>! 🎉</h2>
      <div style="display: grid; grid-template-columns: repeat(auto-fit, minmax(180px, 1fr)); gap: 20px; margin-top: 20px;">
        <div style="background: #f0f9ff; padding: 15px; border-radius: 10px; text-align: center;">
          <h3>📁 Files</h3>
          <p style="font-size: 1.5em;">${fileCount}</p>
          <small>Total files on system</small>
        </div>
        <div style="background: #fff8e1; padding: 15px; border-radius: 10px; text-align: center;">
          <h3>📅 Account creation</h3>
          <p style="font-size: 1.5em;">${accountAge}</p>
        </div>
        <div style="background: #fff3e0; padding: 15px; border-radius: 10px; text-align: center;">
          <h3>💾 Server Space</h3>
          <div style="background: #e0e0e0; border-radius: 10px; overflow: hidden; height: 20px;">
            <div style="background: #4caf50; width: ${Math.min((uploaded / (20 * 1024 ** 3)) * 100, 100).toFixed(2)}%; height: 100%;"></div>
          </div>
          <small>Space used out of 20GB</small>
        </div>
      </div>
    `;

    userChart = new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Uploaded', 'Downloaded'],
        datasets: [{
          label: 'Data Transfer (in MB)',
          data: [uploaded / (1024 * 1024), downloaded / (1024 * 1024)],
          backgroundColor: ['#4caf50', '#2196f3'],
          borderRadius: 10,
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: { display: false },
          title: {
            display: true,
            text: `📊 Your Data Journey`
          }
        },
        scales: {
```

```
          y: {
            beginAtZero: true,
            ticks: { stepSize: 50 }
          }
        }
      }
    });

    fileCountDiv.innerHTML = `<strong>📁 Total Files:</strong> ${fileCount}`;

  } catch (error) {
    console.error('Error loading user stats:', error);
    errorDiv.textContent = '❌ Failed to load user statistics. Retrying in 3s...';
    await new Promise(r => setTimeout(r, 3000));
    loadUserStats(); // Retry
  }
}



async function loadUserFiles() {
    const thisToken = Symbol();
    currentLoadToken = thisToken; // Set the current load token
    const password = getCookie("pass");
    const fileSection = document.getElementById('files_grid');

    fileSection.querySelectorAll('.file-item').forEach(item => item.remove());


    try {
      let parent_id = document.getElementById('folder_path').getAttribute('current-id');
        const response = await fetch(`/files/folder?parent=${parent_id}`, {
            credentials: 'include',
            method: 'GET'
        });
        if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`); // Check response ok

        const files = await response.json();

        for (const key in files) {
            if (currentLoadToken !== thisToken) return;
            const file = files[key];
            const server_key = file.server_key;
            const encryptionKey = generateEncryptionKey(password, server_key);
            let decryptedFileName = "Filename Error"; // Default filename in case of decryption error
             try {
                decryptedFileName = CryptoJS.AES.decrypt(file.file_name, encryptionKey).toString(CryptoJS.enc.Utf8);
                if (!decryptedFileName) { // Handle cases where decryption results in empty string
                    decryptedFileName = "corrupt/unnamed File";
                }
```

```
            } catch (e) {
                 console.error(`Failed to decrypt filename for key ${server_key}:`, e);
            }

            const size = file.size;
            const chunk_count = file.chunk_count; // Get chunk_count for progress calculation
            let type = file.type; // Get type for file type check

            // Create UI elements (Your existing code)
            const fileContainer = document.createElement('div');
            fileContainer.className = 'file-item';

            if(type === 0){
              fileContainer.addEventListener('dblclick', async () => {
                if (progressIndicator.style.display === 'block') {
                  alert("Another operation is already in progress."); // Prevent concurrent operations on the same indicator
                  return;
                }
                try {
                  let currentId = document.getElementById('folder_path').getAttribute('current-id');
                  // Stack push
                  folderIdStack.push(currentId);

                  document.getElementById('folder_path').setAttribute('last-id', currentId);
                  document.getElementById('folder_path').setAttribute('current-id', server_key);

                  const pathDisplay = document.getElementById('path_display');
                  pathDisplay.textContent += "/" + decryptedFileName;

                  loadUserFiles(); // Reload files to show the new folder
                } catch (error) {alert('Failed to open folder. Please try again later.');}

              });

            }
            else{
              fileContainer.addEventListener('dblclick', async () => {
                  if (progressIndicator.style.display === 'block') {
                      alert("Another operation is already in progress."); // Prevent concurrent operations on the same indicator
                      return;
                  }

                  try {
                      const chunks = [];
                      let chunkIndex = 0;

                      // --- Show Progress Indicator for Download ---
```

```
                    updateProgress('show', `Starting download: ${decryptedFileName}`, 0, false, true); // isDownload = true

                    while (chunkIndex < chunk_count) {
                        const response = await fetch(`/files/download?key=${server_key}&index=${chunkIndex}`, {
                            method: 'GET',
                            credentials: 'include'
                        });

                        if (response.ok) {
                            const encryptedChunk = await response.text();
                            const decryptedChunk = await decryptAndDecompressChunk(encryptedChunk, encryptionKey);
                            chunks.push(decryptedChunk);
                            chunkIndex++;

                            // --- Update Download Progress ---
                            const percentComplete = Math.round((chunkIndex / chunk_count) * 100);
                            updateProgress('update', `Downloading ${decryptedFileName} ${percentComplete}%`, percentComplete, false, true);

                        } else {
                            throw new Error(`Chunk download failed (Index: ${chunkIndex}, Status: ${response.status})`);
                        }
                    }

                    // --- All chunks downloaded, prepare Blob ---
                    updateProgress('update', `Download complete: ${decryptedFileName}. Preparing file...`, 100, false, true);

                    const blob = new Blob(chunks);
                    const url = URL.createObjectURL(blob);
                    const tempLink = document.createElement('a');
                    tempLink.href = url;
                    tempLink.download = decryptedFileName;
                    document.body.appendChild(tempLink); // Required for Firefox
                    tempLink.click();
                    document.body.removeChild(tempLink); // Clean up
                    URL.revokeObjectURL(url);

                    // --- Hide indicator after success ---
                    setTimeout(() => updateProgress('hide'), 1000); // Hide shortly after click initiated

                } catch (error) {
                    console.error('Download failed:', error);
                    // --- Show error and hide ---
                    updateProgress('show', `Download failed: ${decryptedFileName}`, null, true); // isError = true
                    setTimeout(() => updateProgress('hide'), 3000);
                }
            });}
```

```
454               // --- END OF MODIFIED Listener ---
455
456             function getFileIcon(extension) {
457               extension = extension.toLowerCase();
458
459               const imageExts = ['png', 'jpg', 'jpeg', 'gif', 'bmp', 'webp', 'svg'];
460               const videoExts = ['mp4', 'mov', 'avi', 'mkv', 'webm'];
461               const audioExts = ['mp3', 'wav', 'ogg', 'flac', 'm4a'];
462               const docExts   = ['doc', 'docx'];
463               const pptExts   = ['ppt', 'pptx'];
464               const xlsExts   = ['xls', 'xlsx'];
465               const codeExts  = ['js', 'ts', 'jsx', 'tsx', 'html', 'css', 'py', 'java',
    'c', 'cpp', 'rb', 'php', 'cs', 'go', 'swift', 'sql', 'bash', 'sh', 'pl', 'r', 'dart',
    'kotlin', 'lua', 'yaml', 'json'];
466
467               if (imageExts.includes(extension)) return '🖼️';
468               if (videoExts.includes(extension)) return '🎬';
469               if (audioExts.includes(extension)) return '🎧';
470               if (extension === 'pdf') return '📄';
471               if (['zip', 'rar', '7z', 'tar', 'gz'].includes(extension)) return '🗜️';
472               if (['txt', 'md', 'log'].includes(extension)) return '📃';
473               if (['csv'].includes(extension)) return '📊';
474               if (xlsExts.includes(extension)) return '📊';
475               if (pptExts.includes(extension)) return '📽️';
476               if (docExts.includes(extension)) return '📝';
477               if (['json', 'xml'].includes(extension)) return '📜';
478               if (['db', 'sqlite'].includes(extension)) return '🗄️';
479               if (['apk'].includes(extension)) return '📱';
480               if (['exe', 'msi'].includes(extension)) return '💻';
481               if (codeExts.includes(extension)) return '💻';
482               if (extension === '') return '📁';
483               return '🤓'; // Default/fallback
484             }
485             // Create file type icon
486             function handleDelete(fileId) {
487               fetch(`/files/delete/file?key=${fileId}`, {
488                   method: 'DELETE',
489                   credentials: 'include'
490               })
491               .then(response => {
492                   if (response.ok) {
493                       console.log(`File ${decryptedFileName} deleted successfully.`);
494                       fileContainer.remove();
495                   } else {
496                       console.error(`Failed to delete file ${decryptedFileName}. Status:
    ${response.status}`);
497                   }
498               })
499               .catch(error => {
500                   console.error(`Error deleting file ${decryptedFileName}:`, error);
501               });
502           }
503           const iconSpan = document.createElement('span');
```

```
504         const extension = decryptedFileName.includes('.') ?
    decryptedFileName.split('.').pop() : '';
505         iconSpan.textContent = getFileIcon(extension);
506         fileContainer.appendChild(iconSpan);
507
508         // File name
509         const fileLabel = document.createElement('span');
510         fileLabel.className = 'file-label';
511         fileLabel.textContent = decryptedFileName;
512
513         // File size
514         const fileSize = document.createElement('span');
515         fileSize.className = 'file-size';
516         fileSize.textContent = formatFileSize(file.size);
517         if(type === 0) {fileSize.style.display = 'none';} // Hide size for folders
518         // Dropdown toggle arrow
519         const toggleArrow = document.createElement('span');
520         toggleArrow.className = 'dropdown-toggle';
521         toggleArrow.textContent = '▼';
522
523         // Dropdown menu
524         const dropdownMenu = document.createElement('div');
525         dropdownMenu.className = 'dropdown-menu';
526         dropdownMenu.style.display = 'none'; // Initially hidden
527
528         const shareBtn = document.createElement('button');
529         shareBtn.className = 'dropdown-btn';
530         shareBtn.textContent = 'Share';
531         if(type === 0) {shareBtn.style.display = 'none';} // Hide share button for folders
532
533         const deleteBtn = document.createElement('button');
534         deleteBtn.className = 'dropdown-btn';
535         deleteBtn.textContent = 'Delete';
536
537         shareBtn.onclick = (event) => {
538           event.stopPropagation();
539           handleShare(server_key, encryptionKey);
540         };
541
542       deleteBtn.onclick = (event) => {
543           event.stopPropagation();
544           handleDelete(server_key);
545         };
546
547         dropdownMenu.appendChild(shareBtn);
548         dropdownMenu.appendChild(deleteBtn);
549
550         toggleArrow.onclick = (event) => {
551           event.stopPropagation();
552
553           // Remove any existing dropdown to avoid duplicates
554           const existingMenu = document.querySelector('.dropdown-menu.global');
555           if (existingMenu) existingMenu.remove();
556
```

```
            const rect = toggleArrow.getBoundingClientRect();

            // Clone dropdown to body
            const globalMenu = dropdownMenu.cloneNode(true);
            globalMenu.classList.add('global');
            globalMenu.style.display = 'block';
            globalMenu.style.position = 'fixed';
            globalMenu.style.top = `${rect.bottom}px`;
            globalMenu.style.left = `${rect.left}px`;

            // Add functionality back to buttons
            globalMenu.querySelector('.dropdown-btn:nth-child(1)').onclick = () => handleShare(server_key, encryptionKey);
            globalMenu.querySelector('.dropdown-btn:nth-child(2)').onclick = () => handleDelete(server_key);

            document.body.appendChild(globalMenu);
        };

        document.addEventListener('click', () => {
          const existingMenu = document.querySelector('.dropdown-menu.global');
          if (existingMenu) existingMenu.remove();
        });

          fileContainer.appendChild(dropdownMenu);
          fileContainer.appendChild(fileLabel);
          fileContainer.appendChild(fileSize);
          fileContainer.appendChild(toggleArrow);

          fileSection.appendChild(fileContainer);
          }
    } catch (error) {
        console.error('Error loading user files:', error);
        fileSection.innerHTML = '<p style="color: red;">Error loading files. Please check console or try again later.</p>';
    }

}

async function decryptAndDecompressChunk(encryptedChunk, key) {
    // Decrypt
    const decrypted = CryptoJS.AES.decrypt(encryptedChunk, key);
    const base64 = decrypted.toString(CryptoJS.enc.Utf8);

    // Convert base64 to binary
    const binary = atob(base64);
    const bytes = new Uint8Array(binary.length);
    for (let i = 0; i < binary.length; i++) {
        bytes[i] = binary.charCodeAt(i);
    }

    // Decompress
    return pako.inflate(bytes);
}
```

```
608
609 function generateEncryptionKey(password, fileId) {
610     const salt = CryptoJS.enc.Utf8.parse(fileId); // סלט ייחודי לכל קובץ
611     const key = CryptoJS.PBKDF2(password, salt, {
612         keySize: 256 / 32,
613         iterations: 100000,
614         hasher: CryptoJS.algo.SHA256
615     });
616     return key.toString();
617 }
618 async function hashString(password, salt) {
619   let saltedInput = salt + password;
620   let hash = CryptoJS.SHA256(saltedInput);
621   return hash.toString(CryptoJS.enc.Hex);
622 }
623
624 function generateRandomId() {
625   return Math.random().toString(36).substring(2, 15);
626 }
627
628 function getCookie(name) {
629   const value = `; ${document.cookie}`;
630   const parts = value.split(`; ${name}=`);
631   if (parts.length === 2) return parts.pop().split(';').shift();
632   return null;
633 }
634
635 function handleShare(fileId, key) {
636             fetch('/share', {
637                 method: 'POST',
638                 credentials: 'include',
639                 headers: {
640                     'Content-Type': 'application/json'
641                 },
642                 body: JSON.stringify({
643                     password: key,
644                     server_key: fileId
645                 })
646             })
647             .then(response => {
648                 if (!response.ok) {
649                     throw new Error(`Failed to unlock file. Status: ${response.status}`);
650                 }
651                 return response.json();
652             })
653             .then(data => {
654                 if (data && data.share_link) {
655                     navigator.clipboard.writeText(data.share_link)
656                         .then(() => {
657                             showBubble("✅ Share link copied to clipboard!");
658                         })
659                         .catch(() => {
660                             showBubble("⚠️ Couldn't copy to clipboard");
661                         });
```

```
                } else {
                    showBubble("❌ Invalid response from server.");
                }
            })
            .catch(error => {
                console.error("Error:", error);
                showBubble("❌ Failed to unlock or copy link.");
            });
        }

function showBubble(message) {
    const bubble = document.getElementById("bubble");
    bubble.textContent = message;
    bubble.style.opacity = 1;

    setTimeout(() => {
        bubble.style.opacity = 0;
    }, 3000); // hide after 3 seconds
}

```

web-server\website\scripts\upload_file.js

```
const dropZone = document.getElementById('files');
const fileInput = document.getElementById('file');
const CHUNK_SIZE = 1024 * 1024 * 2 ; // 2MB chunks

dropZone.addEventListener('dragenter', (e) => e.preventDefault());
dropZone.addEventListener('dragover', (e) => e.preventDefault());
dropZone.addEventListener('drop', handleFileDrop);

function handleFileDrop(e) {
    e.preventDefault();
    fileInput.files = e.dataTransfer.files;
    processFiles(storedPassword, Array.from(fileInput.files));
}

function handleShare(fileId, key) {
            fetch('/share', {
                method: 'POST',
                credentials: 'include',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    password: key,
                    server_key: fileId
                })
            })
            .then(response => {
                if (!response.ok) {
                    throw new Error(`Failed to unlock file. Status: ${response.status}`);
                }
                return response.json();
            })
            .then(data => {
                if (data && data.share_link) {
                    navigator.clipboard.writeText(data.share_link)
                        .then(() => {
                            showBubble("✅ Share link copied to clipboard!");
                        })
                        .catch(() => {
                            showBubble("⚠️ Couldn't copy to clipboard");
                        });
                } else {
                    showBubble("❌ Invalid response from server.");
                }
            })
            .catch(error => {
                console.error("Error:", error);
                showBubble("❌ Failed to unlock or copy link.");
            });
        }

```

```
52 function showBubble(message) {
53     const bubble = document.getElementById("bubble");
54     bubble.textContent = message;
55     bubble.style.opacity = 1;
56 
57     setTimeout(() => {
58         bubble.style.opacity = 0;
59     }, 3000); // hide after 3 seconds
60 }
61 
62 
63 async function processFiles(password, files) {
64     let parent_id = document.getElementById('folder_path').getAttribute("current-id")
65     for (const file of files) {
66         let fileId;
67         let totalChunks = 0;
68         try {
69             fileId = generateRandomId();
70             const key = generateEncryptionKey(password, fileId);
71             const encryptedFileName =
CryptoJS.AES.encrypt(CryptoJS.enc.Utf8.parse(file.name), key).toString();
72 
73             // Calculate chunk count
74             totalChunks = Math.ceil(file.size / CHUNK_SIZE);
75 
76             // --- Show progress for the current file ---
77             updateProgress('show', `Starting upload: ${file.name}`, 0);
78 
79             // Send file metadata first
80             const metadataResult = await sendFileMetadata(fileId, encryptedFileName,
totalChunks, file.size, parent_id);
81 
82             if (!metadataResult.ok) {
83                 if (metadataResult.reason === "quota") {
84                     showBubble("🚫 Upload denied: you've reached your 20GB storage
limit.");
85                 } else {
86                     showBubble("❌ Failed to send file metadata.");
87                 }
88                 return;
89             }
90 
91 
92             // Process file in chunks
93             for (let i = 0; i < totalChunks; i++) { // Use totalChunks
94                 const start = i * CHUNK_SIZE;
95                 const end = Math.min(start + CHUNK_SIZE, file.size);
96                 const chunk = file.slice(start, end);
97 
98                 // Read chunk
99                 const chunkData = await readChunk(chunk);
100                 // Compress and encrypt chunk
101                 const processedChunk = await compressAndEncryptChunk(chunkData, key);
102 
```

```
103                 // Send chunk
104                 await uploadChunk(fileId, i, processedChunk);
105 
106                 // --- Update progress after each chunk ---
107                 const percentComplete = Math.round(((i + 1) / totalChunks) * 100);
108                 updateProgress('update', `Uploading ${file.name} ${percentComplete}%`,
    percentComplete);
109             }
110 
111              // --- Show success and hide after delay ---
112              setTimeout(() => {
113                 updateProgress('update', `Upload complete: ${file.name}`);
114 
115                 // Now schedule hiding AFTER 2 seconds
116                 setTimeout(() => {
117                   updateProgress('hide');
118                 }, 2000);
119               }, 0); // Start immediately (or after a small delay if needed)
120 
121             // Update UI for the file (Your existing UI update code)
122             const fileSection = document.getElementById('files_grid');
123             const fileContainer = document.createElement('div');
124             fileContainer.className = 'file-item';
125 
126 
127 
128             fileContainer.addEventListener('dblclick', async () => {
129                  if (progressIndicator.style.display === 'block') {
130                     alert("Another operation is already in progress."); // Prevent
    concurrent operations on the same indicator
131                     return;
132                 }
133 
134                 try {
135                     const chunks = [];
136                     let chunkIndex = 0;
137 
138                     // --- Show Progress Indicator for Download ---
139                      updateProgress('show', `Starting download: ${file.name}`, 0, false,
    true); // isDownload = true
140 
141                     while (chunkIndex < totalChunks) { // Use totalChunks
142                         const response = await fetch(`/files/download?
    key=${fileId}&index=${chunkIndex}`, {
143                             method: 'GET',
144                             credentials: 'include'
145                         });
146 
147                         if (response.ok) {
148                             const encryptedChunk = await response.text();
149                             const decryptedChunk = await decryptAndDecompress-
    Chunk(encryptedChunk, key);
150                             chunks.push(decryptedChunk);
151                             chunkIndex++;
```

```
152
153                             // --- Update Download Progress ---
154                             const percentComplete = Math.round((chunkIndex / totalChunks)
    * 100);
155                             updateProgress('update', `Downloading ${file.name}
    ${percentComplete}%`, percentComplete, false, true);
156
157                         } else {
158                             throw new Error(`Chunk download failed (Index: ${chunkIndex},
    Status: ${response.status})`);
159                         }
160                     }
161
162                     // --- All chunks downloaded, prepare Blob ---
163                     updateProgress('update', `Download complete: ${file.name}. Preparing
    file...`, 100, false, true);
164
165                     const blob = new Blob(chunks);
166                     const url = URL.createObjectURL(blob);
167                     const tempLink = document.createElement('a');
168                     tempLink.href = url;
169                     tempLink.download = file.name;
170                     document.body.appendChild(tempLink); // Required for Firefox
171                     tempLink.click();
172                     document.body.removeChild(tempLink); // Clean up
173                     URL.revokeObjectURL(url);
174
175                     // --- Hide indicator after success ---
176                     setTimeout(() => updateProgress('hide'), 1000); // Hide shortly after
    click initiated
177
178                 } catch (error) {
179                     console.error('Download failed:', error);
180                     // --- Show error and hide ---
181                     updateProgress('show', `Download failed: ${file.name}`, null, true);
    // isError = true
182                     setTimeout(() => updateProgress('hide'), 3000);
183                 }
184             });
185
186             function handleDelete(fileId) {
187                 fetch(`/files/delete/file?key=${fileId}`, {
188                     method: 'DELETE',
189                     credentials: 'include'
190                 })
191                 .then(response => {
192                     if (response.ok) {
193                         console.log(`File ${file.name} deleted successfully.`);
194                         fileContainer.remove();
195                     } else {
196                         console.error(`Failed to delete file ${file.name}. Status:
    ${response.status}`);
197                     }
198                 })
```

```
                .catch(error => {
                    console.error(`Error deleting file ${file.name}:`, error);
                });
            }
            function getFileIcon(extension) {
                extension = extension.toLowerCase();

                const imageExts = ['png', 'jpg', 'jpeg', 'gif', 'bmp', 'webp', 'svg'];
                const videoExts = ['mp4', 'mov', 'avi', 'mkv', 'webm'];
                const audioExts = ['mp3', 'wav', 'ogg', 'flac', 'm4a'];
                const docExts   = ['doc', 'docx'];
                const pptExts   = ['ppt', 'pptx'];
                const xlsExts   = ['xls', 'xlsx'];
                const codeExts  = ['js', 'ts', 'jsx', 'tsx', 'html', 'css', 'py', 'java', 'c', 'cpp', 'rb', 'php', 'cs', 'go', 'swift', 'sql', 'bash', 'sh', 'pl', 'r', 'dart', 'kotlin', 'lua', 'yaml', 'json'];
                if (imageExts.includes(extension)) return '🖼️';
                if (videoExts.includes(extension)) return '🎬';
                if (audioExts.includes(extension)) return '🎧';
                if (extension === 'pdf') return '📄';
                if (['zip', 'rar', '7z', 'tar', 'gz'].includes(extension)) return '🗜️';
                if (['txt', 'md', 'log',"json"].includes(extension)) return '📃';
                if (['csv'].includes(extension)) return '📊';
                if (xlsExts.includes(extension)) return '📊';
                if (pptExts.includes(extension)) return '📽️';
                if (docExts.includes(extension)) return '📝';
                if (['json', 'xml'].includes(extension)) return '📜';
                if (['db', 'sqlite'].includes(extension)) return '🗄️';
                if (['apk'].includes(extension)) return '📱';
                if (['exe', 'msi'].includes(extension)) return '💻';
                if (codeExts.includes(extension)) return '💻';

                return '🤓'; // Default/fallback
            }
            const iconSpan = document.createElement('span');
            iconSpan.textContent = getFileIcon(file.name.split('.').pop());
            fileContainer.appendChild(iconSpan);

            // File name
            const fileLabel = document.createElement('span');
            fileLabel.className = 'file-label';
            fileLabel.textContent = file.name;

            // File size
            const fileSize = document.createElement('span');
            fileSize.className = 'file-size';
            fileSize.textContent = formatFileSize(file.size);

            // Dropdown toggle arrow
            const toggleArrow = document.createElement('span');
            toggleArrow.className = 'dropdown-toggle';
            toggleArrow.textContent = '▼';

            // Dropdown menu
```

```
            const dropdownMenu = document.createElement('div');
            dropdownMenu.className = 'dropdown-menu';
            dropdownMenu.style.display = 'none'; // Initially hidden

            const shareBtn = document.createElement('button');
            shareBtn.className = 'dropdown-btn';
            shareBtn.textContent = 'Share';

            const deleteBtn = document.createElement('button');
            deleteBtn.className = 'dropdown-btn';
            deleteBtn.textContent = 'Delete';

            shareBtn.onclick = (event) => {
              event.stopPropagation();
              handleShare(fileId, key);
            };

            deleteBtn.onclick = (event) => {
              event.stopPropagation();
              handleDelete(fileId);
            };

            dropdownMenu.appendChild(shareBtn);
            dropdownMenu.appendChild(deleteBtn);

            toggleArrow.onclick = (event) => {
              event.stopPropagation();

              // Remove any existing dropdown to avoid duplicates
              const existingMenu = document.querySelector('.dropdown-menu.global');
              if (existingMenu) existingMenu.remove();

              const rect = toggleArrow.getBoundingClientRect();

              // Clone dropdown to body
              const globalMenu = dropdownMenu.cloneNode(true);
              globalMenu.classList.add('global');
              globalMenu.style.display = 'block';
              globalMenu.style.position = 'fixed';
              globalMenu.style.top = `${rect.bottom}px`;
              globalMenu.style.left = `${rect.left}px`;

              // Add functionality back to buttons
              globalMenu.querySelector('.dropdown-btn:nth-child(1)').onclick = () => handleShare(fileId, key);
              globalMenu.querySelector('.dropdown-btn:nth-child(2)').onclick = () => handleDelete(fileId);

              document.body.appendChild(globalMenu);
            };

            document.addEventListener('click', () => {
              const existingMenu = document.querySelector('.dropdown-menu.global');
              if (existingMenu) existingMenu.remove();
```

```
            });
              fileContainer.appendChild(dropdownMenu);
              fileContainer.appendChild(fileLabel);
              fileContainer.appendChild(fileSize);
              fileContainer.appendChild(toggleArrow);


              fileSection.appendChild(fileContainer);
              console.log(`File metadata sent for ${file.name}`);

          } catch (error) {
              console.error(`Error processing file ${file.name}:`, error);
               // --- Show error and hide after delay ---
               updateProgress('show', `Upload failed: ${file.name}`, null, true); // isError = true
               setTimeout(() => updateProgress('hide'), 3000); // Hide after 3 seconds
          }
      }
}


async function readChunk(chunk) {
    return new Promise((resolve) => {
        const reader = new FileReader();
        reader.onload = (e) => resolve(e.target.result);
        reader.readAsArrayBuffer(chunk);
    });
}

async function compressAndEncryptChunk(chunkData, key) {
    // Compress chunk
    const compressed = pako.deflate(new Uint8Array(chunkData));

    // Convert to base64 in smaller chunks to avoid stack overflow
    const chunkSize = 32768; // 32KB chunks for string conversion
    let binary = '';
    for (let i = 0; i < compressed.length; i += chunkSize) {
        const slice = compressed.subarray(i, i + chunkSize);
        binary += String.fromCharCode.apply(null, slice);
    }

    // Convert to base64
    const base64 = btoa(binary);

    // Encrypt
    return CryptoJS.AES.encrypt(base64, key).toString();
}


async function uploadChunk(fileId, index, chunk) {
    const response = await fetch('/files/upload/chunk', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
```

```
356         },
357         body: JSON.stringify({
358             key: fileId,
359             index: index,
360             data: chunk
361         }),
362         credentials: 'include'
363     });
364 
365     if (!response.ok) {
366         throw new Error(`Failed to upload chunk ${index}`);
367     }
368 }
369 
370 function generateRandomId() {
371     return Math.random().toString(36).substring(2, 15);
372 }
373 
374 function generateEncryptionKey(password, fileId) {
375     const salt = CryptoJS.enc.Utf8.parse(fileId);
376     const key = CryptoJS.PBKDF2(password, salt, {
377         keySize: 256 / 32,
378         iterations: 100000,
379         hasher: CryptoJS.algo.SHA256
380     });
381     return key.toString();
382 }
383 
384 async function sendFileMetadata(fileId, encryptedFileName, chunkCount, size, parentId) {
385     try {
386         const response = await fetch('/files/upload/file', {
387             method: 'POST',
388             headers: { 'Content-Type': 'application/json' },
389             body: JSON.stringify({
390                 server_key: fileId,
391                 file_name: encryptedFileName,
392                 chunk_count: chunkCount,
393                 size: size,
394                 parent_id: parentId
395             })
396         });
397 
398         const result = await response.json();
399 
400         if (!response.ok || result.failed) {
401             // Check if the message contains quota exceeded info
402             const msg = result.message?.toString() || "";
403             if (msg.includes("quota")) {
404                 return { ok: false, reason: "quota" };
405             }
406 
407             return { ok: false };
408         }
409 
```

```
        console.log("File metadata uploaded");
        return { ok: true };
    } catch (error) {
        console.error('Error sending file metadata:', error);
        return { ok: false };
    }
}


function formatFileSize(bytes) {
    if (bytes === 0) return '0 Bytes';

    const units = ['Bytes', 'KB', 'MB', 'GB', 'TB'];
    const unitIndex = Math.floor(Math.log(bytes) / Math.log(1024));
    const size = (bytes / Math.pow(1024, unitIndex)).toFixed(2);

    return `${size} ${units[unitIndex]}`;
  }

```