VenueSender Codebase Overview

Created by Spencer Lievens

August 22, 2023

Contents

1 main.cpp

Source Code

```cpp
#include "include/main.h"

using namespace confPaths;
using namespace std;

ConfigManager configManager;
CsvReader csvReader;
CurlHandleWrapper curlWrapper;
EmailManager emailManager;
EncryptionManager encryptionManager;
MenuManager menuManager;
VenueFilter venueFilter;
VenueUtilities venueUtilities;

#ifndef UNIT_TESTING

int main() {
  // ... (truncated for brevity)
  return 0;
}
#endif // UNIT_TESTING

```

Explanation

- **Namespaces**: The standard namespace (`std`) and `confPaths` namespace are utilized throughout the program.
- **Global Objects**: Several global objects representing various functionalities (e.g., configuration management, email management, venue filtering) are declared and used throughout the program.
- **Main Function**: The main function is the entry point for the VenueSender application. Within this function:
  - Variables necessary for the program are initialized.
  - Configurations are loaded from a JSON file.
  - The password is decrypted.
  - CURL is set up and initialized.
  - The main menu loop prompts the user for various actions (e.g., filtering venues, sending emails).
  - Before exiting, configurations are reset and resources are cleaned up.
- **Unit Testing**: The main function is guarded by the `UNIT_TESTING` preprocessor directive. When this is defined, the main function is excluded, which is useful for unit testing scenarios.

1.1 main.h

Header File

```cpp
#ifndef MAIL_H
#define MAIL_H

#include "curl.h"
#include "fileutils.h"
#include "menu.h"
#include "errorhandler.h"
#include "structs.h"

#include <ctime>
```

```
#include <filesystem>
#include <regex>
#include <thread>

class EmailManager {
  public:
  static inline const int MAX_MESSAGE_LENGTH = 2000;
  static inline const int MAX_SUBJECT_LENGTH = 50;
  const size_t MAX_ATTACHMENT_SIZE = 24 * 1024 * 1024;

  std::string getCurrentDateRfc2822();
  std::string sanitizeSubject(std::string& subject);
  void viewEmailSettings(bool useSSL, bool verifyPeer, bool verifyHost, bool verbose,
  const std::string& senderEmail, int smtpPort, const std::string& smtpServer);
  bool isValidEmail(const std::string& email);
  void constructEmail(std::string &subject, std::string &message, std::string &
attachmentPath,
  std::string &attachmentName, std::string &attachmentSize, std::istream &in = std::cin);
  bool sendIndividualEmail(CURL* curl,
  const SelectedVenue& selectedVenue,
  const std::string& senderEmail,
  std::string& subject,
  std::string& message,
  const std::string& smtpServer,
  int smtpPort,
  std::string& attachmentName,
  std::string& attachmentSize,
  const std::string& attachmentPath);
  void viewEmailSendingProgress(const std::string& senderEmail);
};

#endif // MAIL_H

```

Explanation

- **Header Guards**: `MAIL_H` is used as the header guard. This ensures that the header file's content is included only once in the compilation process.
- **Dependencies**: This header file includes dependencies related to CURL operations, file utilities, menu options, error handling, and venue structures. Additionally, there are standard library includes for time, file system operations, regular expressions, and threading.
- **EmailManager Class**:
  - **Constants**: Defines maximum lengths for the email message body and subject. There's also a constant for the maximum attachment size.
  - **Utility Functions**: Provides methods for getting the current date in RFC 2822 format, sanitizing the email subject, and displaying the email settings.
  - **Validation**: Contains a method to validate the format of an email address.
  - **Email Construction**: Function to construct the content of an email, including the subject, message, and attachment details.
  - **Email Sending**: Functions to send individual emails and view the progress of the sending process.

2 menu.cpp

Source Code

```
#include "include/menu.h"

using namespace std;

```

```
const int MenuManager::FILTER_BY_GENRE_OPTION = static_cast<int>(MenuManager::MenuOption::FilterByGenre);
const int MenuManager::FILTER_BY_STATE_OPTION = static_cast<int>(MenuManager::MenuOption::FilterByState);
const int MenuManager::FILTER_BY_CITY_OPTION = static_cast<int>(MenuManager::MenuOption::FilterByCity);
const int MenuManager::FILTER_BY_CAPACITY_OPTION = static_cast<int>(MenuManager::MenuOption::FilterByCapacity);

const int MenuManager::CLEAR_SELECTED_VENUES_OPTION = static_cast<int>(MenuManager::MenuOption::ClearSelectedVenues);
const int MenuManager::VIEW_SELECTED_VENUES_OPTION = static_cast<int>(MenuManager::MenuOption::ViewSelectedVenues);
const int MenuManager::SHOW_EMAIL_SETTINGS_OPTION = static_cast<int>(MenuManager::MenuOption::ShowEmailSettings);
const int MenuManager::VIEW_EDIT_EMAILS_OPTION = static_cast<int>(MenuManager::MenuOption::ViewEditEmail);
const int MenuManager::FINISH_AND_SEND_EMAILS_OPTION = static_cast<int>(MenuManager::MenuOption::FinishAndSendEmail);
const int MenuManager::EXIT_OPTION = static_cast<int>(MenuManager::MenuOption::Exit);

bool MenuManager::isValidMenuChoice(int choice) {
  return choice >= static_cast<int>(MenuOption::FilterByGenre) &&
  choice <= static_cast<int>(MenuOption::Exit);
}

int MenuManager::displayMenuOptions() {
  #ifdef UNIT_TESTING
  // ... (truncated for brevity)
  #else
  // ... (truncated for brevity)
  #endif
}

void MenuManager::displaySelectedVenues(const vector<SelectedVenue>& selectedVenues) {
  // ... (truncated for brevity)
}

```

Explanation

- **Namespace Utilization**: The standard namespace (`std`) is used throughout the code.
- **Menu Constants Initialization**: Constants representing different menu options are initialized using static casts.
- **Menu Validation**: The ‘isValidMenuChoice‘ function checks whether a given choice falls within the range of valid menu options.
- **Menu Display**: The ‘displayMenuOptions‘ function handles the display of the main menu. Depending on whether `UNIT_TESTING` is defined, different behaviors are executed.
- **Selected Venues Display**: The ‘displaySelectedVenues‘ function shows the venues chosen by the user. If no venues are selected, an error message is displayed.

2.1 menu.h

Header File

This header file defines the `MenuManager` class responsible for managing menu-related operations within VenueSender.

Explanation

- **Header Guards**: `MENU_H` is used to prevent double inclusion.
- **Dependencies**:

  - `errorhandler.h`: For handling errors.
  - `fileutils.h`: Utility functions for file operations.
  - `structs.h`: Contains the structure definitions used in the program.
  - `iostream`: For standard input/output operations.
  - `vector`: For using the vector data structure.

- **Class Definition**:
  - `MenuManager`: A class dedicated to manage menu-related operations.
  - **MenuOption Enumeration**: Represents the available options in the main menu.
  - **Constants for Menu Options**: These constants are used to represent the different menu options available to the user.
  - **isValidMenuChoice(int choice)**: Validates if the user's menu choice is within the range of valid options.
  - **displayMenuOptions()**: Displays the available menu options to the user and returns the user's choice.
  - **displaySelectedVenues(const std::vector<SelectedVenue>& selectedVenues)**: Displays the list of venues that the user has selected.

3 filtercriteria.cpp

Source Code

```
#include "include/filtercriteria.h"

// Use the standard namespace
using namespace std;

// Utility function to convert a Venue object to a SelectedVenue object
SelectedVenue VenueUtilities::convertToSelectedVenue(const Venue& venue) {
  // ... (truncated for brevity)
}

// Utility function to get unique genres from a list of venues
set<string> VenueUtilities::getUniqueGenres(const vector<Venue>& venues) {
  // ... (truncated for brevity)
}

// Utility function to get unique states from a list of venues
set<string> VenueUtilities::getUniqueStates(const vector<Venue>& venues) {
  // ... (truncated for brevity)
}

// Utility function to get unique cities from a list of venues
set<string> VenueUtilities::getUniqueCities(const vector<Venue>& venues) {
  // ... (truncated for brevity)
}

// Utility function to get unique capacities from a list of venues
set<int> VenueUtilities::getUniqueCapacities(const vector<Venue>& venues) {
  // ... (truncated for brevity)
}

// Function to process venue selection based on user input
void VenueFilter::processVenueSelection(const vector<SelectedVenue>&
temporaryFilteredVenues,
vector<SelectedVenue>& selectedVenuesForEmail,
istream& input,
ostream& output) {
  // ... (truncated for brevity)
}

// Function to display filtered venues to the user
```

```cpp
void VenueFilter::displayFilteredVenues(const vector<SelectedVenue>& selectedVenuesForDisplay) {
  // ... (truncated for brevity)
}

// Common function for filtering venues by Genre, State, or City
vector<SelectedVenue> VenueFilter::filterByOptionCommon(const vector<Venue>& venues,
const set<string>& uniqueOptions,
const string& filterType,
vector<SelectedVenue>& temporaryFilteredVenues) {
  // ... (truncated for brevity)
}

// Function to filter venues by Genre, State, or City
vector<SelectedVenue> VenueFilter::filterByOption(const vector<Venue>& venues,
const string& filterType,
const set<string>& uniqueOptions,
vector<SelectedVenue>& temporaryFilteredVenues) {
  // ... (truncated for brevity)
}

// Function to filter venues by Capacity
vector<SelectedVenue> VenueFilter::filterByCapacity(const vector<Venue>& venues,
const set<int>& uniqueCapacities,
vector<SelectedVenue>& temporaryFilteredVenues) {
  // ... (truncated for brevity)
}

```

Explanation

- **Namespace Utilization**: The standard namespace (`std`) is used throughout the code.
- **VenueUtilities**: This class provides utility functions to convert between different types of venue objects, retrieve unique genre, state, city, and capacity values from a list of venues.
- **VenueFilter**: This class contains functions to filter and display venues based on different criteria such as genre, state, city, and capacity.
  - `processVenueSelection()`: Processes venue selection based on user input and updates the list of venues to be emailed.
  - `displayFilteredVenues()`: Displays the venues that match the applied filters.
  - `filterByOptionCommon()`: A common function used for filtering venues based on either genre, state, or city.
  - `filterByOption()`: Filters venues by genre, state, or city.
  - `filterByCapacity()`: Filters venues based on their capacity.

3.1 filtercriteria.h

Header File

```cpp
#ifndef FILTERCRITERIA_H
#define FILTERCRITERIA_H

#include "fileutils.h"
#include "structs.h"

#include <iomanip>
#include <iostream>
#include <set>
#include <vector>

// Structure to hold filter criteria for venues
struct FilterCriteria {
  std::string genre;
  std::string state;
  std::string city;
};

// Utility class to perform common venue-related operations
class VenueUtilities {
  public:
  static SelectedVenue convertToSelectedVenue(const Venue& venue);
  std::set<std::string> getUniqueGenres(const std::vector<Venue>& venues);
  std::set<std::string> getUniqueStates(const std::vector<Venue>& venues);
  std::set<std::string> getUniqueCities(const std::vector<Venue>& venues);
  std::set<int> getUniqueCapacities(const std::vector<Venue>& venues);
};

// Class to handle venue filtering logic
class VenueFilter {
  private:
  const std::string::size_type MAX_INPUT_LENGTH = 256;
  const char CSV_DELIMITER = ',';

  std::vector<SelectedVenue> filterByOptionCommon(const std::vector<Venue>& venues,
  const std::set<std::string>& uniqueOptions,
  const std::string& filterType,
  std::vector<SelectedVenue>& temporaryFilteredVenues);

  public:
  void processVenueSelection(const std::vector<SelectedVenue>& temporaryFilteredVenues,
  std::vector<SelectedVenue>& selectedVenuesForEmail,
  std::istream& input = std::cin,
  std::ostream& output = std::cout);

  void displayFilteredVenues(const std::vector<SelectedVenue>& selectedVenuesForDisplay);
  std::vector<SelectedVenue> filterByOption(const std::vector<Venue>& venues,
  const std::string& filterType,
  const std::set<std::string>& uniqueOptions,
  std::vector<SelectedVenue>& temporaryFilteredVenues);
  std::vector<SelectedVenue> filterByCapacity(const std::vector<Venue>& venues,
  const std::set<int>& uniqueCapacities,
  std::vector<SelectedVenue>& temporaryFilteredVenues);
};

#endif // FILTERCRITERIA_H

```

Explanation

- **Include Directives**: This header file includes references to other header files, namely ‘fileutils.h‘ and ‘structs.h‘, along with standard library headers.
- **FilterCriteria Structure**: Defines the criteria that can be used for filtering venues - genre, state, and city.
- **VenueUtilities Class**: Provides utility functions related to venues such as converting between venue object types and getting unique attributes from a list of venues.
- **VenueFilter Class**: Contains methods for filtering venues based on user input and criteria. This includes processing user input, displaying filtered venues, and filtering venues based on genre, state, city, and capacity.

4 structs.h

Header File

```
#ifndef STRUCTS_H
#define STRUCTS_H

#include <iostream>
```

```

struct Venue {
  std::string name;
  std::string email;
  std::string genre;
  std::string state;
  std::string city;
  int capacity;

  Venue() = default;
  Venue(const std::string& name, const std::string& email, const std::string& genre,
  const std::string& state, const std::string& city, int capacity)
  : name(name), email(email), genre(genre), state(state), city(city), capacity(capacity) {}
};

struct SelectedVenue {
  std::string name;
  std::string email;
  std::string genre;
  std::string state;
  std::string city;
  int capacity;

  SelectedVenue() = default;
  SelectedVenue(const std::string& name, const std::string& email, const std::string& genre,
  const std::string& state, const std::string& city, int capacity)
  : name(name), email(email), genre(genre), state(state), city(city), capacity(capacity) {}
};

#endif // STRUCTS_H

```

Explanation

- **Dependencies**: The header file includes the 'iostream' library for string manipulation and input/output operations.
- **Struct Definition: Venue**:
  - Represents a venue with members like name, email, genre, state, city, and capacity.
  - **Default Constructor**: Allows creating uninitialized Venue objects.
  - **Parameterized Constructor**: Initializes all members of the struct.
- **Struct Definition: SelectedVenue**:
  - Represents a selected venue with the same members as the Venue struct.
  - **Default Constructor**: Allows creating uninitialized SelectedVenue objects.
  - **Parameterized Constructor**: Initializes all members of the struct.

5 mail.cpp

Source Code

```
#include "include/mail.h"

using namespace std;
namespace fs = filesystem;

CurlHandleWrapper curlHandleWrapper;
ErrorHandler errorHandler;

string EmailManager::getCurrentDateRfc2822() {
      // Implementation to get the current date in RFC 2822 format...
    }

    string EmailManager::sanitizeSubject(string& subject) {
      // Implementation to sanitize the email subject...
    }

    void EmailManager::viewEmailSettings(bool useSSL, bool verifyPeer, bool verifyHost, bool verbose,
    const string& senderEmail, int smtpPort, const string& smtpServer) {
      // Implementation to display current email settings...
    }

    bool EmailManager::isValidEmail(const string& email) {
      // Implementation to check if a provided string is a valid email format...
    }

    void EmailManager::constructEmail(string &subject, string &message, string &attachmentName,
    string &attachmentSize, string &attachmentPath, istream &in) {
      // Implementation to guide the user in constructing an email...
    }

    bool EmailManager::sendIndividualEmail(CURL* curl,
    const SelectedVenue& selectedVenue,
    const string& senderEmail,
    string& subject,
    string& message,
    const string& smtpServer,
    int smtpPort,
    string& attachmentName,
    string& attachmentSize,
    const string& attachmentPath) {
      // Implementation to send an individual email...
    }

    void EmailManager::viewEmailSendingProgress(const string& senderEmail) {
      // Implementation to display email sending progress...
    }

```

Explanation

- **Namespace and Global Objects**: This source file utilizes the standard namespace and a namespace alias for ‘filesystem‘. Additionally, global objects for cURL handling and error handling are defined.
- **getCurrentDateRfc2822()**: Returns the current date in RFC 2822 format.
- **sanitizeSubject()**: Sanitizes the subject line of an email by replacing newline and carriage return characters with spaces.
- **viewEmailSettings()**: Displays the current email settings.
- **isValidEmail()**: Validates whether the provided string conforms to a valid email format.
- **constructEmail()**: Guides the user in constructing an email, with provisions for a subject, message, and optional attachment.
- **sendIndividualEmail()**: Sends an individual email to a selected venue based on specified configurations.
- **viewEmailSendingProgress()**: Displays the progress of email sending to the user.

5.1 mail.h

Header File

```
#ifndef MAIL_H
#define MAIL_H

#include "curl.h"
#include "fileutils.h"
#include "menu.h"
#include "errorhandler.h"
#include "structs.h"

#include <ctime>
#include <filesystem>
#include <regex>
#include <thread>

class EmailManager {
  public:
  static inline const int MAX_MESSAGE_LENGTH = 2000;
  static inline const int MAX_SUBJECT_LENGTH = 50;
  const size_t MAX_ATTACHMENT_SIZE = 24 * 1024 * 1024;

  std::string getCurrentDateRfc2822();
  std::string sanitizeSubject(std::string& subject);
  void viewEmailSettings(bool useSSL, bool verifyPeer, bool verifyHost, bool verbose,
  const std::string& senderEmail, int smtpPort, const std::string& smtpServer);
  bool isValidEmail(const std::string& email);
  void constructEmail(std::string &subject, std::string &message, std::string &
attachmentPath,
  std::string &attachmentName, std::string &attachmentSize, std::istream &in = std::cin);
  bool sendIndividualEmail(CURL* curl,
  const SelectedVenue& selectedVenue,
  const std::string& senderEmail,
  std::string& subject,
  std::string& message,
  const std::string& smtpServer,
  int smtpPort,
  std::string& attachmentName,
  std::string& attachmentSize,
  const std::string& attachmentPath);
  void viewEmailSendingProgress(const std::string& senderEmail);
};

#endif // MAIL_H

```

Explanation

- **Dependencies**: The header file includes dependencies related to cURL, file utilities, menu handling, error handling, and data structures. Additionally, it utilizes the ‘ctime‘, ‘filesystem‘, ‘regex‘, and ‘thread‘ libraries.
- **Class Definition: EmailManager**:
  - **Constants**: The class defines constants for the maximum lengths of email messages and subjects, as well as the maximum size for attachments.
  - **getCurrentDateRfc2822()**: Returns the current date in the RFC 2822 format.
  - **sanitizeSubject()**: Sanitizes the subject line of an email.
  - **viewEmailSettings()**: Displays the email settings.
  - **isValidEmail()**: Checks if a given string is a valid email format.
  - **constructEmail()**: Guides the user in constructing an email.
  - **sendIndividualEmail()**: Sends an individual email to a selected venue.
  - **viewEmailSendingProgress()**: Displays the progress of email sending.

6 fileutils.cpp

Source Code

```cpp
#include "include/fileutils.h"
#include "errorhandler.h" // Include due to circular dependency between fileutils.h and errorhandler.h

using namespace std;

namespace confPaths {
  string venuesCsvPath = "venues.csv";
  string configJsonPath = "config.json";
  string mockVenuesCsvPath = "src/test/mock_venues.csv";
  string mockConfigJsonPath = "src/test/mock_config.json";
}

string ConsoleUtils::trim(const string& str){
  const auto notSpace = [](int ch) {return !isspace(ch); };
  auto first = find_if(str.begin(), str.end(), notSpace);
  auto last = find_if(str.rbegin(), str.rend(), notSpace).base();
  return (first < last ? string(first, last) : string());
}

void ConsoleUtils::clearInputBuffer() {
  cin.clear();
  cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

void CsvReader::readCSV(vector<Venue>& venues, string& venuesCsvPath) {
  ifstream file(venuesCsvPath);
  if (!file.is_open()) {
    ErrorHandler errorHandler;
    errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::CONFIG_OPEN_ERROR, venuesCsvPath);
    return;
  }

  string line;
  getline(file, line); // Skip header

  while (getline(file, line)) {
    istringstream ss(line);
    string data;
    vector<string> rowData;
    while (getline(ss, data, ',')) {
      rowData.push_back(ConsoleUtils::trim(data));
    }

    if (rowData.size() == 6) {
      Venue venue(rowData[0], rowData[1], rowData[2], rowData[3], rowData[4], stoi(rowData[5]));
      venues.push_back(venue);
    } else {
      ErrorHandler errorHandler;
      errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::INVALID_DATA_IN_CSV, venuesCsvPath);
    }
  }

  file.close();
}

ConfigManager::ConfigManager() {}

bool ConfigManager::loadConfigSettings(bool& useSSL, bool& verifyPeer, bool& verifyHost, bool& verbose,
string& senderEmail, string& smtpUsername,
string& mailPass, int& smtpPort, string& smtpServer,
string& venuesCsvPath) {
  // Implementation includes conditional compilation for unit testing,
  // handles encryption and decryption of email passwords, and
  // validates the loaded settings.
}

void ConfigManager::resetConfigFile() {
  // Implementation reads the existing JSON configuration,
  // modifies specific keys and values, and writes the updated
  // JSON back to the file.
}

```

Explanation

- **Namespace** `confPaths`: This namespace holds configuration file paths.
- **ConsoleUtils::trim**: Function to remove leading and trailing whitespace from a given string.
- **ConsoleUtils::clearInputBuffer**: Function to clear the input buffer.
- **CsvReader::readCSV**: Function to read venue data from a CSV file.
- **ConfigManager::ConfigManager**: Default constructor for the ConfigManager class.
- **ConfigManager::loadConfigSettings**: Function to load the configuration settings. This implementation includes conditional compilation for unit testing, handles encryption and decryption of email passwords, and validates the loaded settings.
- **ConfigManager::resetConfigFile**: Function to reset the configuration file. This implementation reads the existing JSON configuration, modifies specific keys and values, and writes the updated JSON back to the file.

6.1 fileutils.h

Header File

```
#ifndef FILEUTILS_H
#define FILEUTILS_H

#include "encryption.h"
#include "structs.h"

#include <algorithm>
#include <fstream>
#include <limits>
#include <vector>
#include <json/json.h>

class ErrorHandler;

namespace confPaths {
  extern std::string venuesCsvPath;
  extern std::string configJsonPath;
  extern std::string mockVenuesCsvPath;
  extern std::string mockConfigJsonPath;
}

class ConsoleUtils {
  public:
  static void clearInputBuffer();
  static std::string trim(const std::string& str);
};

class CsvReader {
  public:
  static void readCSV(std::vector<Venue>& venues, std::string& venuesCsvPath);
};

class ConfigManager {
  private:
```

```cpp
  EncryptionManager encryptionManager;

  public:
  ConfigManager();
  bool loadConfigSettings(bool& useSSL, bool& verifyPeer, bool& verifyHost, bool& verbose,
  std::string& senderEmail, std::string& smtpUsername,
  std::string& mailPass, int& smtpPort, std::string& smtpServer,
  std::string& venuesCsvPath);
  static void resetConfigFile();
};

#endif // FILEUTILS_H

```

Explanation

- **Header Guards**: ‘FILEUTILS_H‘ is used to prevent the file from being included more than once in a single compilation.
- **Forward Declaration**: A forward declaration of ‘ErrorHandler‘ is made to resolve circular dependencies with ‘errorhandler.h‘.
- **Namespace `confPaths`**: This namespace holds string variables for configuration file paths.
- **ConsoleUtils Class**: A utility class with static functions related to console operations, such as clearing the input buffer and trimming strings.
- **CsvReader Class**: A class responsible for reading data from a CSV file.
- **ConfigManager Class**: A class responsible for managing the configuration settings. It has an instance of the ‘EncryptionManager‘ for password encryption and decryption. This class provides methods to load settings from a configuration file and reset the configuration file.

7 encryption.cpp

Source Code

```cpp
#include "include/encryption.h"

using namespace std;

array<unsigned char, crypto_secretbox_KEYBYTES> globalEncryptionKey;
array<unsigned char, crypto_secretbox_NONCEBYTES> globalEncryptionNonce;

EncryptionManager::EncryptionManager() {
  if (sodium_init() < 0) {
    ErrorHandler errorHandler;
    errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::LIBSODIUM_INIT_ERROR);
    exit(EXIT_FAILURE);
  }

  randombytes_buf(globalEncryptionKey.data(), crypto_secretbox_KEYBYTES);
  randombytes_buf(globalEncryptionNonce.data(), crypto_secretbox_NONCEBYTES);
}

bool EncryptionManager::encryptPassword(const string& decryptedPassword, string& encryptedPassword) {
  unsigned char encryptedBuffer[crypto_secretbox_MACBYTES + decryptedPassword.size()];

  if (crypto_secretbox_easy(encryptedBuffer, reinterpret_cast<const unsigned char*>(decryptedPassword.c_str()), decryptedPassword.size(),
  globalEncryptionNonce.data(), globalEncryptionKey.data()) != 0) {
    ErrorHandler errorHandler;
    errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::EMAIL_PASSWORD_ENCRYPTION_ERROR);
    return false;
  }

  string nonceStr(reinterpret_cast<const char*>(globalEncryptionNonce.data()), globalEncryptionNonce.size());
  encryptedPassword = nonceStr + string(reinterpret_cast<const char*>(encryptedBuffer), sizeof(encryptedBuffer));

  return true;
}

string EncryptionManager::decryptPassword(const string& encryptedPassword) {
  const size_t NONCE_LENGTH = crypto_secretbox_NONCEBYTES;

  if (encryptedPassword.size() < crypto_secretbox_MACBYTES + NONCE_LENGTH) {
    ErrorHandler errorHandler;
    errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::EMAIL_PASSWORD_ENCRYPTION_FORMAT_ERROR);
    exit(EXIT_FAILURE);
  }

  string nonce = encryptedPassword.substr(0, NONCE_LENGTH);
  string ciphertext = encryptedPassword.substr(NONCE_LENGTH);
  unsigned char decryptedBuffer[ciphertext.size() - crypto_secretbox_MACBYTES];

  if (crypto_secretbox_open_easy(decryptedBuffer,
  reinterpret_cast<const unsigned char*>(ciphertext.c_str()),
  ciphertext.size(),
  reinterpret_cast<const unsigned char*>(nonce.c_str()),
  globalEncryptionKey.data()) != 0) {
    ErrorHandler errorHandler;
    errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::EMAIL_PASSWORD_DECRYPTION_ERROR);
    exit(EXIT_FAILURE);
  }

  return string(reinterpret_cast<char*>(decryptedBuffer), ciphertext.size() - crypto_secretbox_MACBYTES);
}

```

Explanation

- **globalEncryptionKey & globalEncryptionNonce**: Global arrays to store the encryption key and nonce.
- **EncryptionManager::EncryptionManager**: Default constructor for the EncryptionManager class. It initializes the libsodium library and generates a random encryption key and nonce.
- **EncryptionManager::encryptPassword**: Utility function that encrypts a given password using the global encryption key and nonce.
- **EncryptionManager::decryptPassword**: Utility function that decrypts an encrypted password using the global encryption key and nonce.

7.1 encryption.h

Header File

```
#ifndef ENCRYPTION_H
#define ENCRYPTION_H

#include "errorhandler.h"

#include <array>
#include <iostream>
#include <sodium.h>

extern std::array<unsigned char, crypto_secretbox_KEYBYTES> globalEncryptionKey;
extern std::array<unsigned char, crypto_secretbox_NONCEBYTES> globalEncryptionNonce;
```

```

class EncryptionManager {
  public:
  EncryptionManager();
  static bool encryptPassword(const std::string& decryptedPassword, std::string&
encryptedPassword);
  static std::string decryptPassword(const std::string& encryptedPassword);
};

#endif // ENCRYPTION_H

```

Explanation

- **globalEncryptionKey & globalEncryptionNonce**: Global arrays that store the encryption key and nonce, respectively.
- **EncryptionManager**: This class handles functionalities related to encryption.
- **EncryptionManager::EncryptionManager**: Constructor initializes the encryption key and nonce.
- **EncryptionManager::encryptPassword**: Utility function that encrypts a password. It takes in the original password to encrypt and stores the encrypted password in the given reference.
- **EncryptionManager::decryptPassword**: Utility function that decrypts a password. It takes in an encrypted password and returns the decrypted password as a string.

8 curl.cpp

Source Code

```
#include "include/curl.h"

using namespace std;

/* CurlHandleWrapper Class Implementation */

int CurlHandleWrapper::progressCallback(void* /*clientp*/, double dltotal, double dlnow,
double /*ultotal*/, double /*ulnow*/) {
  // Implementation for displaying the progress of email sending
}

size_t CurlHandleWrapper::readCallback(void* ptr, size_t size, size_t nmemb, void* userp)
{
  // Implementation for reading the email payload
}

void CurlHandleWrapper::setSSLOptions(bool useSSL, bool verifyPeer, bool verifyHost) {
  // Implementation to set SSL options for cURL
}

CurlHandleWrapper::CurlHandleWrapper() : curl(nullptr) {
  // Constructor implementation for CurlHandleWrapper
}

CurlHandleWrapper::~CurlHandleWrapper() {
  // Destructor implementation for CurlHandleWrapper
}

CURL* CurlHandleWrapper::get() const {
  // Implementation to get the cURL handle
}

void CurlHandleWrapper::init() {
  // Implementation for global cURL initialization
}

```

```
void CurlHandleWrapper::cleanup() {
  // Implementation for global cURL cleanup
}

void CurlHandleWrapper::setEmailBeingSent(const string& email) {
  // Setter for emailBeingSent
}

string CurlHandleWrapper::getEmailBeingSent() const {
  // Getter for emailBeingSent
}

void CurlHandleWrapper::clearEmailBeingSent() {
  // Clears the emailBeingSent variable
}

CURL* setupCurlHandle(CurlHandleWrapper &curlWrapper, bool useSSL, bool verifyPeer, bool
verifyHost, bool verbose,
const string& senderEmail, const string& smtpUsername, string& mailPassDecrypted, int
smtpPort, const string& smtpServer) {
  // Function to set up a cURL handle with various settings
}

```

Explanation

- **CurlHandleWrapper Class**: This class manages the cURL handle and the associated operations.
- **progressCallback**: Function to update and display the progress of email sending.
- **readCallback**: Callback function to read email payload data for sending in the request.
- **setSSLOptions**: Sets the SSL options for the cURL handle.
- **CurlHandleWrapper Constructor & Destructor**: Initializes and cleans up the cURL handle.
- **init & cleanup**: Global functions for initializing and cleaning up the cURL library.
- **setEmailBeingSent, getEmailBeingSent, & clearEmailBeingSent**: Functions to manage the email address being sent.
- **setupCurlHandle**: Sets up a cURL handle with various options including SSL, SMTP authentication, and other settings.

8.1 curl.h

Header File

```
#ifndef CURL_H
#define CURL_H

#include "errorhandler.h"

#include <algorithm>
#include <cstring>
#include <iostream>
#include <mutex>
#include <curl/curl.h>

class CurlHandleWrapper {
  public:
  CurlHandleWrapper();
  ~CurlHandleWrapper();
  CURL* get() const;
  static void init();
  static void cleanup();
  int progressCallback(void* /*clientp*/, double dltotal, double dlnow, double /*ultotal*/
, double /*ulnow*/);
```

```
  void setEmailBeingSent(const std::string& email);
  std::string getEmailBeingSent() const;
  void clearEmailBeingSent();
  void setSSLOptions(bool useSSL = true, bool verifyPeer = true, bool verifyHost = true);
  static size_t readCallback(void* ptr, size_t size, size_t nmemb, void* userp);

  private:
  CURL* curl;
  double progress{};
  std::string emailBeingSent;
  mutable std::mutex mtx;
};

CURL* setupCurlHandle(CurlHandleWrapper &curlWrapper, bool useSSL, bool verifyPeer, bool verifyHost, bool verbose,
const std::string& senderEmail, const std::string& smtpUsername, std::string& mailPassDecrypted, int smtpPort, const std::string& smtpServer);

#endif // CURL_H

```

Explanation

- **CurlHandleWrapper**: This class manages the cURL handle and its associated operations.
- **CurlHandleWrapper::CurlHandleWrapper**: Constructor initializes the cURL handle.
- **CurlHandleWrapper:: CurlHandleWrapper**: Destructor cleans up the cURL handle.
- **CurlHandleWrapper::get**: Returns the cURL handle for use in other operations.
- **CurlHandleWrapper::init & CurlHandleWrapper::cleanup**: Static methods to initialize and cleanup the cURL library respectively.
- **CurlHandleWrapper::progressCallback**: Callback function to update the progress of ongoing operations.
- **CurlHandleWrapper::setEmailBeingSent, CurlHandleWrapper::getEmailBeingSent, & CurlHandleWrapper::clearEmailBeingSent**: Methods to set, get, and clear the email address that is currently being sent.
- **CurlHandleWrapper::setSSLOptions**: Method to set SSL options for the cURL handle.
- **CurlHandleWrapper::readCallback**: Static callback function for reading the email payload.
- **setupCurlHandle**: Global function to set up a cURL handle with the provided options.

9 errorhandler.cpp

Source Code

```
#include "include/errorhandler.h"
#include "include/fileutils.h" // Forward declaration due to circular dependency

using namespace std;

void ErrorHandler::showInfoAndReturn() {
  // Implementation for displaying an info message and waiting for user action
}

void ErrorHandler::showInfoAndRetry() {
  // Implementation to display a message and ask the user to retry
}

bool ErrorHandler::handleCurlError(CURLcode res) {
  // Implementation to handle CURL errors
}
```

```cpp

void ErrorHandler::handleErrorAndReturn(ErrorType error) {
  // Overloaded function to handle errors without extra information
}

void ErrorHandler::handleErrorAndReturn(ErrorType error, const string& extraInfo = "") {
  // Implementation to handle various types of errors and display appropriate messages
}

```

Explanation

- **showInfoAndReturn**: Displays an information message and waits for the user to press return to continue.
- **showInfoAndRetry**: Displays a message and waits for the user to press return to retry.
- **handleCurlError**: Handles errors from the cURL library and displays appropriate error messages.
- **handleErrorAndReturn (Overloaded version)**: Handles various error types without additional context or information.
- **handleErrorAndReturn**: Manages different types of errors, displaying corresponding error messages. It can also incorporate additional context or information, provided as the ‘extraInfo‘ parameter.

9.1 errorhandler.h

Header File

```cpp
#ifndef ERRORHANDLER_H
#define ERRORHANDLER_H

#include <curl/curl.h>
#include <iostream>

// Forward declarations due to circular dependency
class CsvReader;
class ConfigManager;

class ErrorHandler {
  public:
  // Enumeration for different error types
  enum class ErrorType {
    // Different error types are listed here...
  };

  // Function to display info and pause before returning to the main menu
  void showInfoAndReturn();

  // Function to display info and pause before retrying an operation
  void showInfoAndRetry();

  // Function to handle errors and display appropriate messages
  void handleErrorAndReturn(ErrorType error);

  // Overloaded function to handle errors with extra information
  void handleErrorAndReturn(ErrorType error, const std::string& extraInfo);

  // Function to handle cURL library errors
  bool handleCurlError(CURLcode res);
};

#endif // ERRORHANDLER_H

```

Explanation

- **ErrorType Enumeration**: Represents different types of errors that the application can encounter.
- **showInfoAndReturn**: Displays a message and waits for the user's input before returning to the main menu.
- **showInfoAndRetry**: Shows an error message and waits for the user's input before retrying a particular operation.
- **handleErrorAndReturn**: Manages different error types and displays corresponding error messages.
- **handleErrorAndReturn (Overloaded version)**: This version of the function allows for an additional string parameter to provide more context to the error.
- **handleCurlError**: Manages errors from the cURL library and provides feedback on whether an operation was successful or not.

10 test_venuesender.cpp

This file is used for testing and should not be deleted.

Imports and Namespace

```
#define CATCH_CONFIG_RUNNER

#include "../include/main.h"
// ... (other includes)

using namespace std;

```

CinGuard and CoutGuard Classes

Utility classes to guard and redirect the standard input and output streams respectively.

CinGuard

```
class CinGuard {
  streambuf* orig_cin;
  public:
  CinGuard(streambuf* newbuf);
  ~CinGuard();
};

```

CoutGuard

```
class CoutGuard {
  streambuf* orig_cout;
  public:
  CoutGuard(streambuf* newbuf);
  ~CoutGuard();
};

```

Test Groups

The file is organized into several test groups. Each test group corresponds to testing a specific component or functionality.

ConsoleUtils Test Group

```
TEST_CASE("ConsoleUtils::trim() functionality", "[ConsoleUtils]") {
  // ... Test cases and REQUIRE statements
}

```

CsvReader Test Group

```
TEST_CASE("CsvReader::readCSV() functionality", "[CsvReader]") {
  // ... Test cases and REQUIRE statements
}

```

EmailManager Test Group

```
TEST_CASE("EmailManager::getCurrentDateRfc2822() functionality", "[EmailManager]") {
  // ... Test cases and REQUIRE statements
}

```

MenuManager Test Group

```
TEST_CASE("MenuManager::isValidMenuChoice() functionality", "[MenuManager]") {
  // ... Test cases and REQUIRE statements
}

```

Main Function

This is the main function for the test runner which initializes the configurations, runs all tests, and cleans up afterward.

```
int main( int argc, char* argv[] ) {
  // ... Initialization and test execution
  return result;
}

```

Explanation

- The file begins by defining a macro for the Catch2 test framework and including necessary header files.
- Utility classes `CinGuard` and `CoutGuard` are defined to temporarily redirect the standard input and output streams.
- Multiple `TEST_CASE` blocks are used to define tests for different functionalities. Inside each test case, various scenarios are tested using the `REQUIRE` macro.
- The main function initializes the test runner, runs all tests, and cleans up afterward.

11 venues.csv

This file represents a CSV (Comma-Separated Values) database of music venues. Each line in the CSV file corresponds to a single venue and its attributes.

Columns

The file is organized into six columns:

1. **Venue Name**: The name of the venue.
2. **Email**: The contact email address of the venue.
3. **Genre**: The primary genre of music played at the venue.
4. **State**: The state in which the venue is located.
5. **City**: The city in which the venue is located.
6. **Capacity**: The maximum number of people the venue can accommodate.

Data

Venue Name	Email	Genre	State	City	Capacity
Venue1	venue1@mock.com	all	AL	Daphne	100
Venue2	venue2@mock.com	all	UT	Provo	300

Explanation

- The file begins with a header row that lists the names of the columns.
- Each subsequent row provides details about a specific venue, with values separated by commas.
- For example, the ”Alabama Music Box” venue is located in Mobile, Alabama, can accommodate up to 700 people, and plays music of all genres.

12 config.json

This file contains the configuration settings related to the email sending functionality and paths to other resources.

Attributes

1. **email_pass_encrypted**: A boolean that indicates whether the email password is encrypted. (Do not change this value manually.)
2. **email_password**: The password of the sender’s email account.
3. **sender_email**: The email address used to send the emails.
4. **smtp_port**: The port number used for the SMTP server connection.
5. **smtp_server**: The SMTP server address.
6. **smtp_username**: The username used for SMTP authentication.
7. **useSSL**: A boolean that indicates whether SSL should be used for the connection.
8. **venues_csv_path**: The path to the CSV file containing venue information.
9. **verbose**: A boolean that indicates whether to enable verbose logging.
10. **verifyHost**: A boolean that indicates whether to verify the host during the SMTP connection.
11. **verifyPeer**: A boolean that indicates whether to verify the peer during the SMTP connection.

Data Overview

Attribute	Example Value
email_pass_encrypted	false
email_password	”enter_email_password”
sender_email	”enter_your_sender_email”
smtp_port	587
smtp_server	”enter_your_smtp_server”
smtp_username	”enter_your_smtp_username”
useSSL	false
venues_csv_path	”venues.csv”
verbose	false
verifyHost	false
verifyPeer	false

Explanation

The ‘config.json‘ file is a configuration file that contains various settings related to the email sending process and paths to required resources. It’s crucial to ensure the correct values are set for the email sending to work properly. Do not change the ‘email_pass_encrypted‘ manually as it is controlled by the application.