VenueSender Codebase Overview

Created by Spencer Lievens

August 24, 2023

Contents

1   main.cpp

Source Code

```
1     #include "include/main.h"
2
3     using namespace confPaths;
4     using namespace std;
5
6     ConfigManager configManager;
7     CsvReader csvReader;
8     CurlHandleWrapper curlWrapper;
9     EmailManager emailManager;
10    EncryptionManager encryptionManager;
11    MenuManager menuManager;
12    VenueFilter venueFilter;
13    VenueUtilities venueUtilities;
14
15    #ifndef UNIT_TESTING
16
17    ConfigManager configManager;
18    CsvReader csvReader;
19    CurlHandleWrapper& curlWrapper = CurlHandleWrapper::getInstance();
20
21    int main() {
22      // ... (truncated for brevity)
23      return 0;
24    }
25    #endif // UNIT_TESTING
26
```

Explanation

- **Dependencies**: The main source file includes its corresponding header and makes use of two namespaces: 'confPaths' and 'std'.

- **Global Objects**: Several global objects are declared and will be used across different parts of the code. These objects handle configuration management, CSV reading, cURL operations, email management, encryption, menu display, venue filtering, and venue utility operations.

- **Main Function** (`int main()`): The entry point to the program. The function does the following:

  - Initializes various variables and loads configurations from a JSON file.
  - Sets up and initializes CURL.
  - Reads venue data from a CSV file and extracts unique genres, states, cities, and capacities.
  - Contains the main loop to display menu options, filter venues based on user criteria, view and edit email settings, and send emails to selected venues.
  - Handles exit scenarios and cleanup tasks.

- **Conditional Compilation**: The main source file's entire body uses an `#ifndef UNIT_TESTING` directive. This implies that if `UNIT_TESTING` is defined (likely during unit tests), the main function won't compile. Excluding parts of code during unit testing is a common practice.

1.1   main.h

Header File

```
1     #ifndef MAIL_H
2     #define MAIL_H
3
4     #include "curl.h"
```

```
5     #include "fileutils.h"
6     #include "menu.h"
7     #include "errorhandler.h"
8     #include "structs.h"
9
10    #include <ctime>
11    #include <filesystem>
12    #include <regex>
13    #include <thread>
14
15    class EmailManager {
16      public:
17      static inline const int MAX_MESSAGE_LENGTH = 2000;
18      static inline const int MAX_SUBJECT_LENGTH = 50;
19      const size_t MAX_ATTACHMENT_SIZE = 24 * 1024 * 1024;
20
21      std::string getCurrentDateRfc2822();
22      std::string sanitizeSubject(std::string& subject);
23      void viewEmailSettings(bool useSSL, bool verifyPeer, bool verifyHost, bool verbose,
24      const std::string& senderEmail, int smtpPort, const std::string& smtpServer);
25      bool isValidEmail(const std::string& email);
26      void constructEmail(std::string &subject, std::string &message, std::string &
      attachmentPath,
27      std::string &attachmentName, std::string &attachmentSize, std::istream &in = std::cin);
28      bool sendIndividualEmail(CURL* curl,
29      const SelectedVenue& selectedVenue,
30      const std::string& senderEmail,
31      std::string& subject,
32      std::string& message,
33      const std::string& smtpServer,
34      int smtpPort,
35      std::string& attachmentName,
36      std::string& attachmentSize,
37      std::string& attachmentPath);
38      void viewEmailSendingProgress(const std::string& senderEmail);
39    };
40
41    #endif // MAIL_H
42
```

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

```
1    #include "include/menu.h"
2
3    using namespace std;
4
5    const int MenuManager::FILTER_BY_GENRE_OPTION = static_cast<int>(MenuManager::MenuOption::
     FilterByGenre);
6    const int MenuManager::FILTER_BY_STATE_OPTION = static_cast<int>(MenuManager::MenuOption::
     FilterByState);
7    const int MenuManager::FILTER_BY_CITY_OPTION = static_cast<int>(MenuManager::MenuOption::
     FilterByCity);
8    const int MenuManager::FILTER_BY_CAPACITY_OPTION = static_cast<int>(MenuManager::MenuOption
     ::FilterByCapacity);
9
10   const int MenuManager::CLEAR_SELECTED_VENUES_OPTION = static_cast<int>(MenuManager::
     MenuOption::ClearSelectedVenues);
11   const intconst int MenuManager::CLEAR_BOOKING_TEMPLATE_OPTION = static_cast<int>(MenuManager
     ::MenuOption::ClearBookingTemplate); MenuManager::VIEW_SELECTED_VENUES_OPTION = static_cast
     <int>(MenuManager::MenuOption::ViewSelectedVenues);
12   const int MenuManager::SHOW_EMAIL_SETTINGS_OPTION = static_cast<int>(MenuManager::MenuOption
     ::ShowEmailSettings);
13   const int MenuManager::VIEW_EDIT_EMAILS_OPTION = static_cast<int>(MenuManager::MenuOption::
     ViewEditEmail);
14   const int MenuManager::VENUE_BOOKING_TEMPLATE_OPTION = static_cast<int>(MenuManager::
     MenuOption::VenueBookingTemplate);
15   const int MenuManager::EMAIL_CUSTOM_ADDRESS_OPTION = static_cast<int>(MenuManager::
     MenuOption::EmailCustomAddress);
16   const int MenuManager::FINISH_AND_SEND_EMAILS_OPTION = static_cast<int>(MenuManager::
     MenuOption::FinishAndSendEmail);
17   const int MenuManager::EXIT_OPTION = static_cast<int>(MenuManager::MenuOption::Exit);
18
19   bool MenuManager::isValidMenuChoice(int choice) {
20     return choice >= static_cast<int>(MenuOption::FilterByGenre) &&
21     choice <= static_cast<int>(MenuOption::Exit);
22   }
23
24   int MenuManager::displayMenuOptions() {
25     #ifdef UNIT_TESTING
26     // ... (truncated for brevity)
27     #else
28     // ... (truncated for brevity)
29     #endif
30   }
31
32   void MenuManager::displaySelectedVenues(const vector<SelectedVenue>& selectedVenues) {
33     // ... (truncated for brevity)
34   }
35
```

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

```
1        #ifndef MENU_H
2        #define MENU_H
3
4        #include "errorhandler.h"
5        #include "fileutils.h"
6        #include "structs.h"
7
8        #include <iostream>
9        #include <vector>
10
11       // Class to manage menu-related operations
12       class MenuManager {
13         public:
14
15         // Enumeration representing the available options in the main menu
16         enum class MenuOption {
17           FilterByGenre = 1,
18           FilterByState,
19           FilterByCity,
20           FilterByCapacity,
21           ClearSelectedVenues,
22           ClearBookingTemplate,
23           ViewSelectedVenues,
24           ShowEmailSettings,
25           ViewEditEmail,
26           VenueBookingTemplate,
27           EmailCustomAddress,
28           FinishAndSendEmail,
29           Exit
30         };
31
32         // Constants for menu options related to filtering venues
33         static constexpr int FILTER_BY_GENRE_OPTION = static_cast<int>(MenuOption::FilterByGenre
         );
34         static constexpr int FILTER_BY_STATE_OPTION = static_cast<int>(MenuOption::FilterByState
         );
35         static constexpr int FILTER_BY_CITY_OPTION = static_cast<int>(MenuOption::FilterByCity);
36         static constexpr int FILTER_BY_CAPACITY_OPTION = static_cast<int>(MenuOption::
         FilterByCapacity);
37
38         // Constants for other menu options
39         static constexpr int CLEAR_SELECTED_VENUES_OPTION = static_cast<int>(MenuOption::
         ClearSelectedVenues);
40         static constexpr int CLEAR_BOOKING_TEMPLATE_OPTION = static_cast<int>(MenuOption::
         ClearBookingTemplate);
41         static constexpr int VIEW_SELECTED_VENUES_OPTION = static_cast<int>(MenuOption::
         ViewSelectedVenues);
42         static constexpr int SHOW_EMAIL_SETTINGS_OPTION = static_cast<int>(MenuOption::
         ShowEmailSettings);
43         static constexpr int VIEW_EDIT_EMAILS_OPTION = static_cast<int>(MenuOption::
         ViewEditEmail);
44         static constexpr int VENUE_BOOKING_TEMPLATE_OPTION = static_cast<int>(MenuOption::
         VenueBookingTemplate);
45         static constexpr int EMAIL_CUSTOM_ADDRESS_OPTION = static_cast<int>(MenuOption::
         EmailCustomAddress);
46         static constexpr int FINISH_AND_SEND_EMAILS_OPTION = static_cast<int>(MenuOption::
         FinishAndSendEmail);
47         static constexpr int EXIT_OPTION = static_cast<int>(MenuOption::Exit);
48
49         // Function to validate if the user's choice is a valid menu option
```

```
50        inline bool isValidMenuChoice(int choice) const {
51          return choice >= FILTER_BY_GENRE_OPTION && choice <= EXIT_OPTION;
52        }
53
54        // Function to display the available menu options to the user
55        int displayMenuOptions();
56
57        // Function to display the list of venues selected by the user
58        void displaySelectedVenues(const std::vector<SelectedVenue>& selectedVenues);
59
60        // Function to handle the exiting/cleanup of the program
61        bool handleExitOption();
62      };
63
64      #endif // MENU_H
65
```

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

```
1      #include "include/filtercriteria.h"
2
3      // Use the standard namespace
4      using namespace std;
5
6      // Utility function to convert a Venue object to a SelectedVenue object
7      SelectedVenue VenueUtilities::convertToSelectedVenue(const Venue& venue) {
8        // ... (truncated for brevity)
9      }
10
11      // Utility function to get unique genres from a list of venues
12      set<string> VenueUtilities::getUniqueGenres(const vector<Venue>& venues) {
13        // ... (truncated for brevity)
```

```
14        }
15
16        // Utility function to get unique states from a list of venues
17        set<string> VenueUtilities::getUniqueStates(const vector<Venue>& venues) {
18            // ... (truncated for brevity)
19        }
20
21        // Utility function to get unique cities from a list of venues
22        set<string> VenueUtilities::getUniqueCities(const vector<Venue>& venues) {
23            // ... (truncated for brevity)
24        }
25
26        // Utility function to get unique capacities from a list of venues
27        set<int> VenueUtilities::getUniqueCapacities(const vector<Venue>& venues) {
28            // ... (truncated for brevity)
29        }
30
31        // Function to process venue selection based on user input
32        void VenueFilter::processVenueSelection(const vector<SelectedVenue>& temporaryFilteredVenues
          ,
33        vector<SelectedVenue>& selectedVenuesForEmail,
34        istream& input,
35        ostream& output) {
36            // ... (truncated for brevity)
37        }
38
39        // Function to display filtered venues to the user
40        void VenueFilter::displayFilteredVenues(const vector<SelectedVenue>&
          selectedVenuesForDisplay) {
41            // ... (truncated for brevity)
42        }
43
44        // Common function for filtering venues by Genre, State, or City
45        vector<SelectedVenue> VenueFilter::filterByOptionCommon(const vector<Venue>& venues,
46        const set<string>& uniqueOptions,
47        const string& filterType,
48        vector<SelectedVenue>& temporaryFilteredVenues) {
49            // ... (truncated for brevity)
50        }
51
52        // Function to filter venues by Genre, State, or City
53        vector<SelectedVenue> VenueFilter::filterByOption(const vector<Venue>& venues,
54        const string& filterType,
55        const set<string>& uniqueOptions,
56        vector<SelectedVenue>& temporaryFilteredVenues) {
57            // ... (truncated for brevity)
58        }
59
60        // Function to filter venues by Capacity
61        vector<SelectedVenue> VenueFilter::filterByCapacity(const vector<Venue>& venues,
62        const set<int>& uniqueCapacities,
63        vector<SelectedVenue>& temporaryFilteredVenues) {
64            // ... (truncated for brevity)
65        }
66
```

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

```
1       #ifndef FILTERCRITERIA_H
2       #define FILTERCRITERIA_H
3
4       #include "fileutils.h"
5       #include "structs.h"
6
7       #include <iomanip>
8       #include <iostream>
9       #include <set>
10      #include <vector>
11
12      // Structure to hold filter criteria for venues
13      struct FilterCriteria {
14        std::string genre;
15        std::string state;
16        std::string city;
17      };
18
19      // Utility class to perform common venue-related operations
20      class VenueUtilities {
21        public:
22        // Utility function to convert a Venue object to a SelectedVenue object
23        inline static SelectedVenue convertToSelectedVenue(const Venue& venue) {
24          // Create a SelectedVenue instance based on Venue data
25          SelectedVenue selectedVenue;
26          selectedVenue.name = venue.name;
27          selectedVenue.email = venue.email;
28          selectedVenue.genre = venue.genre;
29          selectedVenue.state = venue.state;
30          selectedVenue.city = venue.city;
31          selectedVenue.capacity = venue.capacity;
32          return selectedVenue;
33        }
34
35        // Utility function to get unique genres from a list of venues
36        inline std::set<std::string> getUniqueGenres(const std::vector<Venue>& venues) {
37          std::set<std::string> uniqueGenres;
38          for (const auto& venue : venues) {
39            if (uniqueGenres.find(venue.genre) == uniqueGenres.end()) {
40              uniqueGenres.insert(venue.genre);
41            }
42          }
43          return uniqueGenres;
44        }
45
46        // Utility function to get unique states from a list of venues
47        inline std::set<std::string> getUniqueStates(const std::vector<Venue>& venues) {
48          std::set<std::string> uniqueStates;
49          for (const auto& venue : venues) {
50            if (uniqueStates.find(venue.state) == uniqueStates.end()) {
51              uniqueStates.insert(venue.state);
52            }
53          }
54          return uniqueStates;
55        }
```

```
56
57          // Utility function to get unique cities from a list of venues
58          inline std::set<std::string> getUniqueCities(const std::vector<Venue>& venues) {
59            std::set<std::string> uniqueCities;
60            for (const auto& venue : venues) {
61              if (uniqueCities.find(venue.city) == uniqueCities.end()) {
62                uniqueCities.insert(venue.city);
63              }
64            }
65            return uniqueCities;
66          }
67
68          // Utility function to get unique capacities from a list of venues
69          inline std::set<int> getUniqueCapacities(const std::vector<Venue>& venues) {
70            std::set<int> uniqueCapacities;
71            for (const Venue& venue : venues) {
72              uniqueCapacities.insert(venue.capacity);
73            }
74            return uniqueCapacities;
75          }
76        };
77
78        // Class to handle venue filtering logic
79        class VenueFilter {
80          private:
81          // Max length for venue selection indices input
82          const std::string::size_type MAX_INPUT_LENGTH = 256;
83
84          // Delimiter used in CSV files
85          const char CSV_DELIMITER = ','; // Delimiter used in CSV files
86
87          // Common function to handle venue filtering by options like Genre, State, City
88          std::vector<SelectedVenue> filterByOptionCommon(const std::vector<Venue>& venues,
89          const std::set<std::string>& uniqueOptions,
90          const std::string& filterType,
91          std::vector<SelectedVenue>& temporaryFilteredVenues);
92
93          std::vector<SelectedVenue> selectedVenuesForEmail;
94
95          public:
96          // Function to process user input and select venues based on it
97          void processVenueSelection(const std::vector<SelectedVenue>& temporaryFilteredVenues,
98          std::vector<SelectedVenue>& selectedVenuesForEmail,
99          std::istream& input = std::cin,
100         std::ostream& output = std::cout);
101
102         // Function to display the filtered venues to the user
103         void displayFilteredVenues(const std::vector<SelectedVenue>& selectedVenuesForDisplay);
104
105
106         // Function to filter venues based on a specific option (Genre, State, etc.)
107         std::vector<SelectedVenue> filterByOption(const std::vector<Venue>& venues,
108         const std::string& filterType,
109         const std::set<std::string>& uniqueOptions,
110         std::vector<SelectedVenue>& temporaryFilteredVenues);
111
112         // Function to filter venues based on capacity
113         std::vector<SelectedVenue> filterByCapacity(const std::vector<Venue>& venues,
114         const std::set<int>& uniqueCapacities,
115         std::vector<SelectedVenue>& temporaryFilteredVenues);
116
117         size_t getSelectedVenuesCount() const { return selectedVenuesForEmail.size(); }
118       };
119
120       #endif // FILTERCRITERIA_H
121
```

Explanation

- **Dependencies**: The header file includes dependencies related to file utilities, struct definitions, and various standard libraries like iostream, vector, set, and iomanip.

- **Struct Definition: FilterCriteria**:
  - Represents the filter criteria used to filter venues based on genre, state, and city.

- **Class Definition: VenueUtilities**:
  - A utility class that provides several static methods related to venues. It offers functions to extract unique genres, states, cities, and capacities from a list of venues. Additionally, there's a function to convert a 'Venue' object to a 'SelectedVenue' object.

- **Class Definition: VenueFilter**:
  - A class that manages venue filtering logic. It contains methods for processing user input related to venue selection, displaying filtered venues, filtering venues based on specific criteria like genre, state, city, and capacity. The class also maintains a list of venues selected for emailing.

4   structs.h

Header File

```
1    #ifndef STRUCTS_H
2    #define STRUCTS_H
3
4    #include <iostream>
5
6    struct Venue {
7      std::string name;
8      std::string email;
9      std::string genre;
10     std::string state;
11     std::string city;
12     int capacity;
13
14     Venue() = default;
15     Venue(const std::string& name, const std::string& email, const std::string& genre,
16     const std::string& state, const std::string& city, int capacity)
17     : name(name), email(email), genre(genre), state(state), city(city), capacity(capacity) {}
18   };
19
20   struct SelectedVenue {
21     std::string name;
22     std::string email;
23     std::string genre;
24     std::string state;
25     std::string city;
26     int capacity;
27
28     SelectedVenue() = default;
29     SelectedVenue(const std::string& name, const std::string& email, const std::string& genre,
30     const std::string& state, const std::string& city, int capacity)
31     : name(name), email(email), genre(genre), state(state), city(city), capacity(capacity) {}
32   };
33
34   #endif // STRUCTS_H
35
```

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

```
1     #include "include/mail.h"
2
3     using namespace std;
4     namespace fs = filesystem;
5
6     CurlHandleWrapper& curlHandleWrapper = CurlHandleWrapper::getInstance();
7     ErrorHandler errorHandler;
8
9     static int successfulSends = 0; // Counter for successful email sends
10    int totalEmails;
11
12    string EmailManager::getCurrentDateRfc2822() {
13       // Implementation to get the current date in RFC 2822 format...
14    }
15
16    string EmailManager::sanitizeSubject(string& subject) {
17       // Implementation to sanitize the email subject...
18    }
19
20    void EmailManager::viewEmailSettings(bool useSSL, bool verifyPeer, bool verifyHost, bool
      verbose,
21    const string& senderEmail, int smtpPort, const string& smtpServer) {
22       // Implementation to display current email settings...
23    }
24
25    bool EmailManager::isValidEmail(const string& email) {
26       // Implementation to check if a provided string is a valid email format...
27    }
28
29    void EmailManager::constructEmail(string &subject, string &message, string &attachmentName
      ,
30    string &attachmentSize, string &attachmentPath, istream &in) {
31       // Implementation to guide the user in constructing an email...
32    }
33
34    void EmailManager::viewEditEmails(CURL* curl, const string& smtpServer, int smtpPort,
      vector<SelectedVenue>& selectedVenuesForEmail, const string& senderEmail,
35    string &subject, string &message, string& attachmentName, string& attachmentSize, string&
      attachmentPath, bool& templateExists) {
36       // Implementation to view and edit emails...
37
38    bool EmailManager::sendIndividualEmail(CURL* curl,
39    const SelectedVenue& selectedVenue,
```

```
40        const string& senderEmail,
41        string& subject,
42        string& message,
43        const string& smtpServer,
44        int smtpPort,
45        string& attachmentName,
46        string& attachmentSize,
47        string& attachmentPath,
48        vector<SelectedVenue>& selectedVenuesForEmail) {
49           // Implementation to send an individual email...
50
51        void EmailManager::createBookingTemplate(CURL* curl,
52        const string& senderEmail,
53        string& subject,
54        string& message,
55        const string& smtpServer,
56        int smtpPort,
57        string& attachmentName,
58        string& attachmentSize,
59        string& attachmentPath,
60        vector<SelectedVenue>& selectedVenuesForEmail,
61        bool& templateExists) {
62           // Implementation to create an email from a booking template...
63
64        void EmailManager::emailCustomAddress(CURL* curl,
65        const std::string& senderEmail,
66        std::string& subject,
67        std::string& message,
68        const std::string& smtpServer,
69        int smtpPort,
70        std::string& attachmentName,
71        std::string& attachmentSize,
72        std::string& attachmentPath) {
73           // Implementation to send an individual email to a custom address...
74
75        void EmailManager::confirmSendEmail(CURL* curl,
76        vector<SelectedVenue>& selectedVenuesForEmail,
77        const string& senderEmail,
78        string& subject,
79        string& message,
80        const string& smtpServer,
81        int smtpPort,
82        string& attachmentName,
83        string& attachmentSize,
84        string& attachmentPath) {
85           // Implementation to confirm emails before sending...
86
87
```

Explanation

- **Namespace, Global Objects, and Counters**: This source file utilizes the standard namespace and a namespace alias for 'filesystem'. There are global objects for cURL handling and error handling. There are also global progress counters for successful email sends and total emails.

- **getCurrentDateRfc2822()**: Returns the current date in RFC 2822 format.

- **sanitizeSubject()**: Sanitizes the subject line of an email.

- **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, including subject, message, and optional attachment.

- **viewEditEmails()**: Views the current email (if one has been created) and allows the user to edit the email.

- **sendIndividualEmail()**: Sends an individual email to a selected venue based on specified configurations and updates the progress of email sending.

- **createBookingTemplate()**: Allows the user to create an email based off of a pre-designed booking template.

- **emailCustomAddress()**: Sends an individual email to a custom address based on specified configurations and updates the progress of email sending.

- **confirmSendEmail()**: Allows the user to view the email and confirm whether to send or return to the menu.

5.1 mail.h

Header File

```
1    #ifndef MAIL_H
2    #define MAIL_H
3
4    #include "curl.h"
5    #include "encryption.h"
6    #include "fileutils.h"
7    #include "menu.h"
8    #include "errorhandler.h"
9    #include "structs.h"
10
11   #include <ctime>
12   #include <filesystem>
13   #include <regex>
14   #include <thread>
15
16   // Class responsible for managing email-related operations
17   class EmailManager {
18     public:
19     // Maximum length for email message body
20     static inline const int MAX_MESSAGE_LENGTH = 2000;
21
22     // Maximum length for email subject
23     static inline const int MAX_SUBJECT_LENGTH = 50;
24
25     // Maximum attachment size (24 MB)
26     const size_t MAX_ATTACHMENT_SIZE = 24 * 1024 * 1024;  // 24 MB in bytes
27
28     // Function to get the current date in RFC 2822 format
29     inline std::string getCurrentDateRfc2822() {
30       char buffer[100];
31       time_t now;
32       struct tm *tm_now;
33       time(&now);
34       tm_now = gmtime(&now);
35       strftime(buffer, sizeof buffer, "%a, %d %b %Y %H:%M:%S %Z", tm_now);
36       return buffer;
37     }
38
39     // Function to sanitize the subject line of an email by replacing newline and carriage
     return characters with spaces
40     inline std::string sanitizeSubject(std::string& subject) {
41       std::string sanitized = subject;
42       replace(sanitized.begin(), sanitized.end(), '\n', ' '); // replace newlines with
     spaces
43       replace(sanitized.begin(), sanitized.end(), '\r', ' '); // replace carriage returns
     with spaces
44       return sanitized;
45     }
46
47     inline void clearAllEmailData(std::string &subject, std::string &message, std::string &
```

```
         attachmentName, std::string &attachmentSize, std::string &attachmentPath) {
48           subject.clear();
49           message.clear();
50           attachmentName.clear();
51           attachmentSize.clear();
52           attachmentPath.clear();
53       }
54
55       inline void clearBookingTemplate(std::string &subject, std::string &message, std::string
      &attachmentName, std::string &attachmentSize, std::string &attachmentPath, bool&
      templateExists) {
56           subject.clear();
57           message.clear();
58           attachmentName.clear();
59           attachmentSize.clear();
60           attachmentPath.clear();
61           templateExists = false;
62           ConsoleUtils::setColor(ConsoleUtils::Color::GREEN);
63           std::cout << "Booking template cleared." << std::endl;
64           ConsoleUtils::resetColor();
65       }
66
67       inline void clearSelectedVenues(std::vector<SelectedVenue>& selectedVenuesForEmail) {
68           selectedVenuesForEmail.clear();
69           ConsoleUtils::setColor(ConsoleUtils::Color::GREEN);
70           std::cout << "Selected venues cleared." << std::endl;
71           ConsoleUtils::resetColor();
72       }
73
74       inline void clearAttachmentData(std::string &attachmentName, std::string &attachmentSize
      , std::string &attachmentPath) {
75           attachmentName.clear();
76           attachmentSize.clear();
77           attachmentPath.clear();
78       }
79
80       inline void clearSubjectMessageData(std::string &subject, std::string &message) {
81           subject.clear();
82           message.clear();
83       }
84
85
86       // Function to display the email settings from the configuration file
87       void viewEmailSettings(bool useSSL, bool verifyPeer, bool verifyHost, bool verbose,
88       const std::string& senderEmail, int smtpPort, const std::string& smtpServer);
89
90       // Function to validate an email address format
91       bool isValidEmail(const std::string& email);
92
93       // Function to construct an email, including the subject, message, and attachment
      details
94       void constructEmail(std::string &subject, std::string &message, std::string &
      attachmentPath, std::string &attachmentName, std::string &attachmentSize, std::istream &in
      = std::cin);
95
96       // Function to allow the user to modify the email
97       void viewEditEmails(CURL* curl,
98       const std::string& smtpServer,
99       int smtpPort,
100      std::vector<SelectedVenue>& selectedVenuesForEmail,
101      const std::string& senderEmail,
102      std::string &subject,
103      std::string &message,
104      std::string& attachmentName,
105      std::string& attachmentSize,
106      std::string& attachmentPath,
107      bool& templateExists);
108
109      // Function to send an individual email to a selected venue
```

```
110        bool sendIndividualEmail(CURL* curl,
111        const SelectedVenue& selectedVenue,
112        const std::string& senderEmail,
113        std::string& subject,
114        std::string& message,
115        const std::string& smtpServer,
116        int smtpPort,
117        std::string& attachmentName,
118        std::string& attachmentSize,
119        std::string& attachmentPath,
120        std::vector<SelectedVenue>& selectedVenuesForEmail);
121
122        // Function to send a booking template email
123        void createBookingTemplate(CURL* curl,
124        const std::string& senderEmail,
125        std::string& subject,
126        std::string& message,
127        const std::string& smtpServer,
128        int smtpPort,
129        std::string& attachmentName,
130        std::string& attachmentSize,
131        std::string& attachmentPath,
132        std::vector<SelectedVenue>& selectedVenuesForEmail,
133        bool& templateExistss);
134
135        // Function to send to a custom email address
136        void emailCustomAddress(CURL* curl,
137        const std::string& senderEmail,
138        std::string& subject,
139        std::string& message,
140        const std::string& smtpServer,
141        int smtpPort,
142        std::string& attachmentName,
143        std::string& attachmentSize,
144        std::string& attachmentPath);
145
146        // Function to confirm the email before sending
147        void confirmSendEmail(CURL* curl,
148        std::vector<SelectedVenue>& selectedVenuesForEmail,
149        const std::string& senderEmail,
150        std::string& subject,
151        std::string& message,
152        const std::string& smtpServer,
153        int smtpPort,
154        std::string& attachmentName,
155        std::string& attachmentSize,
156        std::string& attachmentPath);
157    };
158
159    #endif // MAIL_H
160
```

Explanation

- **Dependencies**: The header file includes dependencies related to cURL, file utilities, menu handling, error handling, console utilities, 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.

  – **clearAllEmailData()**: Clears subject, message, attachment name/path/size.

- **clearBookingTemplate()**: Clears subject, message, attachment name/path/size and sets the templateExists boolean to false.
- **clearSelectedVenues()**: Clears the selectedVenuesForEmail vector.
- **clearAttachmentData()**: Clears attachment name/path/size.
- **clearSubjectMessageData()**: Clears subject and message.
- **viewEmailSettings()**: Displays the email settings.
- **isValidEmail()**: Checks if a given string is a valid email format.
- **constructEmail()**: Guides the user in constructing an email with options for subject, message, and attachment.
- **viewEditEmails()**: Views the current email (if one has been created) and allows the user to edit the email.
- **sendIndividualEmail()**: Sends an individual email to a selected venue, updating email sending progress dynamically.
- **emailCustomAddress()**: Sends an individual email to a custom address based on specified configurations and updates the progress of email sending.
- **confirmSendEmail()**: Allows the user to view the email and confirm whether to send or return to the menu.

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 ConsoleUtils::setColor(ConsoleUtils::Color color) {
  // Platform-specific code to set console colors.
  // Windows:
  #ifdef _WIN32
  HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
  switch(color) {
    case ConsoleUtils::Color::RED:
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
    break;
    case ConsoleUtils::Color::GREEN:
    SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
    break;
```

```
37          case ConsoleUtils::Color::BLUE:
38          SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE);
39          break;
40          case ConsoleUtils::Color::MAGENTA:
41          SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_BLUE);
42          break;
43          case ConsoleUtils::Color::YELLOW:
44          SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN);
45          break;
46          case ConsoleUtils::Color::CYAN:
47          SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN | FOREGROUND_BLUE);
48          break;
49          case ConsoleUtils::Color::LIGHT_BLUE:
50          SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE | FOREGROUND_INTENSITY);
51          break;
52          default:
53          SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
     // Default to white
54          break;
55        }
56      #else
57      // UNIX-like systems (using ANSI escape codes):
58      switch(color) {
59          case ConsoleUtils::Color::RED:
60          std::cout << "\033[31m";
61          break;
62          case ConsoleUtils::Color::GREEN:
63          std::cout << "\033[32m";
64          break;
65          case ConsoleUtils::Color::BLUE:
66          std::cout << "\033[34m";
67          break;
68          case ConsoleUtils::Color::MAGENTA:
69          std::cout << "\033[35m";
70          break;
71          case ConsoleUtils::Color::YELLOW:
72          std::cout << "\033[33m";
73          break;
74          case ConsoleUtils::Color::CYAN:
75          std::cout << "\033[36m";
76          break;
77          case ConsoleUtils::Color::LIGHT_BLUE:
78          std::cout << "\033[94m";
79          break;
80          default:
81          std::cout << "\033[0m"; // Reset to default
82          break;
83        }
84      #endif
85    }
86
87
88    void ConsoleUtils::resetColor() {
89      #ifdef _WIN32
90      HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
91      SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE); //
       Reset to white
92      #else
93      std::cout << "\033[0m"; // Reset to default for UNIX-like systems
94      #endif
95    }
96
97    void CsvReader::readCSV(vector<Venue>& venues, string& venuesCsvPath) {
98      ifstream file(venuesCsvPath);
99      if (!file.is_open()) {
100       ErrorHandler errorHandler;
101       errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::CONFIG_OPEN_ERROR,
      venuesCsvPath);
102       return;
```

```
103          }
104
105      string line;
106      getline(file, line); // Skip header
107
108      while (getline(file, line)) {
109        istringstream ss(line);
110        string data;
111        vector<string> rowData;
112        while (getline(ss, data, ',')) {
113          rowData.push_back(ConsoleUtils::trim(data));
114        }
115
116        if (rowData.size() == 6) {
117          Venue venue(rowData[0], rowData[1], rowData[2], rowData[3], rowData[4], stoi(rowData
      [5]));
118          venues.push_back(venue);
119        } else {
120          ErrorHandler errorHandler;
121          errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::INVALID_DATA_IN_CSV,
      venuesCsvPath);
122        }
123      }
124
125      file.close();
126    }
127
128    ConfigManager::ConfigManager() {}
129
130    bool ConfigManager::loadConfigSettings(bool& useSSL, bool& verifyPeer, bool& verifyHost,
      bool& verbose,
131    string& senderEmail, string& smtpUsername,
132    string& mailPass, int& smtpPort, string& smtpServer,
133    string& venuesCsvPath) {
134      // Implementation includes conditional compilation for unit testing,
135      // handles encryption and decryption of email passwords, and
136      // validates the loaded settings.
137    }
138
139    void ConfigManager::resetConfigFile() {
140      // Implementation reads the existing JSON configuration,
141      // modifies specific keys and values, and writes the updated
142      // JSON back to the file.
143    }
144
```

Explanation

- **Namespace** `confPaths`: This namespace holds configuration file paths.

- **ConsoleUtils::trim**: Function to remove leading and trailing whitespace from a given string.

- **ConsoleUtils::setColor**: Function to set console colours.

- **ConsoleUtils::resetColor()**: Function to reset console colours back to defaults.

- **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

```
1    #ifndef FILEUTILS_H
2    #define FILEUTILS_H
3
4    #include "encryption.h"
5    #include "structs.h"
6
7    #include <algorithm>
8    #include <fstream>
9    #include <limits>
10   #include <vector>
11
12   // Include headers for JSON processing
13   #include <jsoncpp/json/json.h>
14
15   // Forward declaration to resolve circular dependency with errorhandler.h
16   class ErrorHandler; // Forward declaration due to circular dependency between fileutils.h
     and errorhandler.h
17
18   // Namespace to hold configuration file paths
19   namespace confPaths {
20     // Declare external string variables for file paths
21     extern std::string venuesCsvPath;
22     extern std::string configJsonPath;
23     extern std::string mockVenuesCsvPath;
24     extern std::string mockConfigJsonPath;
25   }
26
27   // Class for utility functions related to the console
28   class ConsoleUtils {
29     public:
30     enum class Color {
31       RED,
32       GREEN,
33       BLUE,
34       MAGENTA,
35       YELLOW,
36       CYAN,
37       LIGHT_BLUE,
38       DEFAULT
39     };
40
41     // Function to clear the input buffer
42     inline static void clearInputBuffer() {
43       // Clear the input buffer
44       std::cin.clear();
45       std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Clear input
     buffer
46     }
47
48     // Method to set console text color
49     static void setColor(Color color);
50
51     // Method to reset console text color to default
52     static void resetColor();
53
54     // Method to trim leading and trailing spaces from a string
55     static std::string trim(const std::string& str);
56   };
57
```

```
58    // Class for reading data from a CSV file
59    class CsvReader {
60      public:
61      // Method to read CSV data and populate a vector of Venue objects
62      static void readCSV(std::vector<Venue>& venues, std::string& venuesCsvPath);
63    };
64
65    // Class for managing configuration settings
66    class ConfigManager {
67      private:
68      // Instance of EncryptionManager for password encryption/decryption
69      EncryptionManager encryptionManager;
70
71      public:
72      // Default constructor
73      ConfigManager();
74
75      // Method to load settings from a configuration file
76      bool loadConfigSettings(bool& useSSL, bool& verifyPeer, bool& verifyHost, bool& verbose,
77      std::string& senderEmail, std::string& smtpUsername,
78      std::string& mailPass, int& smtpPort, std::string& smtpServer,
79      std::string& venuesCsvPath);
80
81      // Static method to reset the configuration file
82      static void resetConfigFile();
83    };
84
85    #endif // FILEUTILS_H
86
```

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

```
1    #include "include/encryption.h"
2
3    using namespace std;
4
5    array<unsigned char, crypto_secretbox_KEYBYTES> globalEncryptionKey;
6    array<unsigned char, crypto_secretbox_NONCEBYTES> globalEncryptionNonce;
7
8    EncryptionManager::EncryptionManager() {
9      if (sodium_init() < 0) {
10       ErrorHandler errorHandler;
11       errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::LIBSODIUM_INIT_ERROR);
```

```
12        exit(EXIT_FAILURE);
13     }
14
15     randombytes_buf(globalEncryptionKey.data(), crypto_secretbox_KEYBYTES);
16     randombytes_buf(globalEncryptionNonce.data(), crypto_secretbox_NONCEBYTES);
17 }
18
19 bool EncryptionManager::encryptPassword(const string& decryptedPassword, string&
   encryptedPassword) {
20     unsigned char encryptedBuffer[crypto_secretbox_MACBYTES + decryptedPassword.size()];
21
22     if (crypto_secretbox_easy(encryptedBuffer, reinterpret_cast<const unsigned char*>(
   decryptedPassword.c_str()), decryptedPassword.size(),
23     globalEncryptionNonce.data(), globalEncryptionKey.data()) != 0) {
24        ErrorHandler errorHandler;
25        errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::
   EMAIL_PASSWORD_ENCRYPTION_ERROR);
26        return false;
27     }
28
29     string nonceStr(reinterpret_cast<const char*>(globalEncryptionNonce.data()),
   globalEncryptionNonce.size());
30     encryptedPassword = nonceStr + string(reinterpret_cast<const char*>(encryptedBuffer),
   sizeof(encryptedBuffer));
31
32     return true;
33 }
34
35 string EncryptionManager::decryptPassword(const string& encryptedPassword) {
36     const size_t NONCE_LENGTH = crypto_secretbox_NONCEBYTES;
37
38     if (encryptedPassword.size() < crypto_secretbox_MACBYTES + NONCE_LENGTH) {
39        ErrorHandler errorHandler;
40        errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::
   EMAIL_PASSWORD_ENCRYPTION_FORMAT_ERROR);
41        exit(EXIT_FAILURE);
42     }
43
44     string nonce = encryptedPassword.substr(0, NONCE_LENGTH);
45     string ciphertext = encryptedPassword.substr(NONCE_LENGTH);
46     unsigned char decryptedBuffer[ciphertext.size() - crypto_secretbox_MACBYTES];
47
48     if (crypto_secretbox_open_easy(decryptedBuffer,
49     reinterpret_cast<const unsigned char*>(ciphertext.c_str()),
50     ciphertext.size(),
51     reinterpret_cast<const unsigned char*>(nonce.c_str()),
52     globalEncryptionKey.data()) != 0) {
53        ErrorHandler errorHandler;
54        errorHandler.handleErrorAndReturn(ErrorHandler::ErrorType::
   EMAIL_PASSWORD_DECRYPTION_ERROR);
55        exit(EXIT_FAILURE);
56     }
57
58     return string(reinterpret_cast<char*>(decryptedBuffer), ciphertext.size() -
   crypto_secretbox_MACBYTES);
59 }
60
```

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

```
1      #ifndef ENCRYPTION_H
2      #define ENCRYPTION_H
3
4      #include "errorhandler.h"
5
6      #include <array>
7      #include <iostream>
8
9      // Include libsodium for encryption/decryption functionalities
10     #include <sodium.h>
11
12     // Declare global encryption key and nonce arrays
13     extern std::array<unsigned char, crypto_secretbox_KEYBYTES> globalEncryptionKey;
14     extern std::array<unsigned char, crypto_secretbox_NONCEBYTES> globalEncryptionNonce;
15
16     // EncryptionManager class to handle encryption-related functionalities
17     class EncryptionManager {
18       public:
19       // Constructor will initialize encryption key and nonce
20       EncryptionManager();
21
22       // Utility function to encrypt a password
23       // Parameters:
24       // decryptedPassword: The original password to encrypt
25       // encryptedPassword: The encrypted password will be stored here
26       // Return: true if encryption is successful, false otherwise
27       static bool encryptPassword(const std::string& decryptedPassword, std::string&
       encryptedPassword);
28
29       // Utility function to decrypt a password
30       // Parameter: encryptedPassword: The encrypted password to decrypt
31       // Return: The decrypted password as a string
32       static std::string decryptPassword(const std::string& encryptedPassword);
33     };
34
35     #endif // ENCRYPTION_H
36
```

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

```
1        #include "include/curl.h"
2
3        #include "include/fileutils.h" // Forward declaration due to circular dependency between
         fileutils.h and errorhandler.h
4
5        using namespace std;
6
7        // CurlHandleWrapper Class Implementation
8        // ...
9
10       // Progress callback to show progress for email sending
11       int CurlHandleWrapper::progressCallback(...){...}
12
13       // Callback function to read data for sending
14       size_t CurlHandleWrapper::readCallback(...){...}
15
16       // Function to set SSL options
17       void CurlHandleWrapper::setSSLOptions(...){...}
18
19       // Constructor
20       CurlHandleWrapper::CurlHandleWrapper() {...}
21
22       // Destructor
23       CurlHandleWrapper::~CurlHandleWrapper() {...}
24
25       // Get the underlying cURL handle
26       CURL* CurlHandleWrapper::get() const {...}
27
28       // Global cURL initialization
29       void CurlHandleWrapper::init() {...}
30
31       // Global cURL cleanup
32       void CurlHandleWrapper::cleanup() {...}
33
34       // Setter for emailBeingSent
35       void CurlHandleWrapper::setEmailBeingSent(const string& email) {...}
36
37       // Getter for emailBeingSent
38       string CurlHandleWrapper::getEmailBeingSent() const {...}
39
40       // Clear the email being sent
41       void CurlHandleWrapper::clearEmailBeingSent() {...}
42
43       // Setup a cURL handle
44       CURL* setupCurlHandle(...) {...}
45
```

Explanation

- **CurlHandleWrapper Class**: This class provides a wrapper around the cURL handle to manage its behavior and settings.

- **progressCallback()**: A callback to show email sending progress.

- **readCallback()**: A callback function to read the data being sent.

- **setSSLOptions()**: Configures SSL options for the cURL session.

- **Constructor and Destructor**: Manages the lifecycle of the cURL handle.

- **get()**: Returns the underlying cURL handle.

- **init() and cleanup()**: Global initialization and cleanup functions for cURL.

- **setEmailBeingSent(), getEmailBeingSent(), and clearEmailBeingSent()**: Manage the email currently being sent.

- **setupCurlHandle()**: Configures a cURL handle with various settings.

8.1   curl.h

Header File

```
1       #ifndef CURL_H
2       #define CURL_H
3
4       #include "errorhandler.h"
5
6       #include <algorithm>
7       #include <cstring>
8       #include <iostream>
9       #include <mutex>
10
11      // Include headers for email processing
12      #include <curl/curl.h>
13
14      // Class to manage the cURL handle and associated operations
15      class CurlHandleWrapper {
16        public:
17        // Deleted copy constructor and assignment operator
18        CurlHandleWrapper(const CurlHandleWrapper&) = delete;
19        CurlHandleWrapper& operator=(const CurlHandleWrapper&) = delete;
20
21        // Static method to return an instance of the class
22        static CurlHandleWrapper& getInstance() {
23          static CurlHandleWrapper instance; // Guaranteed to be destroyed, instantiated on
    first use.
24          return instance;
25        }
26
27        // Get the cURL handle for use in other operations
28        CURL* get() const;
29
30        // Initialize the cURL library (should be called at program start)
31        static void init();
32
33        // Clean up the cURL library (should be called at program end)
34        static void cleanup();
35
36        inline double getProgress() const {
37          return progress;
38        }
39
40        // Callback function to update the progress of ongoing operations
41        int progressCallback(void* /*clientp*/, double dltotal, double dlnow, double /*ultotal*/
    , double /*ulnow*/);
42
43        // Set the email address that is currently being sent
44        void setEmailBeingSent(const std::string& email);
45
46        // Get the email address that is currently being sent
47        std::string getEmailBeingSent() const;
48
49        // Clear the email address that was set
50        void clearEmailBeingSent();
51
52        // Set SSL options for the cURL handle
53        void setSSLOptions(bool useSSL = true, bool verifyPeer = true, bool verifyHost = true);
54
55        // Callback function for reading the email payload
56        static size_t readCallback(void* ptr, size_t size, size_t nmemb, void* userp);
```

```
57
58        private:
59        CurlHandleWrapper();  // Made private to enforce Singleton pattern
60        ~CurlHandleWrapper(); // Destructor
61
62        // The actual cURL handle
63        CURL* curl;
64
65        // Progress of the current cURL operation
66        double progress{};
67
68        // Email address of the recipient for the email being sent
69        std::string emailBeingSent;
70
71        // Mutex for thread safety
72        mutable std::mutex mtx;
73    };
74
75    // Function to set up a cURL handle with the given options
76    CURL* setupCurlHandle(CurlHandleWrapper &curlWrapper, bool useSSL, bool verifyPeer, bool
    verifyHost, bool verbose,
77        const std::string& senderEmail, const std::string& smtpUsername, std::string&
    mailPassDecrypted, int smtpPort, const std::string& smtpServer);
78
79    #endif // CURL_H
80
```

Explanation

- **Dependencies**: The header file includes the 'errorhandler.h' for error handling, standard libraries, the cURL library, and utilities for multi-threading.

- **Class Definition: CurlHandleWrapper**:

  – This class provides a Singleton pattern to ensure a single instance of the cURL handle.
  – The copy constructor and assignment operator are deleted to prevent copying.
  – **getInstance()**: Provides access to the single instance of this class.
  – **get()**: Returns the cURL handle.
  – **init() and cleanup()**: Static methods for initializing and cleaning up the cURL library.
  – **getProgress()**: Inline method to get the progress of the ongoing cURL operation.
  – **progressCallback()**: Callback to update the progress of ongoing operations.
  – **setEmailBeingSent(), getEmailBeingSent(), and clearEmailBeingSent()**: Manage the email currently being sent.
  – **setSSLOptions()**: Configures SSL options for the cURL session.
  – **readCallback()**: Callback function to read the email payload.
  – **setupCurlHandle()**: A utility function to set up and configure a cURL handle with the given options.

9   errorhandler.cpp

Source Code

```
1        #include "include/errorhandler.h"
2        #include "include/fileutils.h"
3
4        using namespace std;
5
6        std::mutex ErrorHandler::outputMutex;
7
8        bool ErrorHandler::handleCurlError(CURLcode res) {
```

```
9        if (res != CURLE_OK) {
10          cerr << "CURL Error: " << curl_easy_strerror(res) << endl;
11          handleErrorAndReturn(ErrorType::LIBCURL_ERROR);
12          return false;
13        }
14        return true;
15      }
16
17      void ErrorHandler::handleErrorAndReturn(ErrorType error) {
18        handleErrorAndReturn(error, "");
19      }
20
21      void ErrorHandler::handleErrorAndReturn(ErrorType error, const string& extraInfo) {
22        std::lock_guard<std::mutex> lock(outputMutex);
23        switch (error) {
24          // ... (All the error handling cases go here)
25        }
26      }
27
```

Explanation

- **Dependencies**: This source file imports the error handling and file utilities header files for proper functioning.

- **Mutex Initialization**: A mutex is defined for thread safety. It ensures that multiple threads don't concurrently access the error handling function, which could lead to interleaved error messages.

- **handleCurlError()**: This function checks for any cURL-related errors. If one occurs, it displays a relevant error message and returns 'false'. Otherwise, it returns 'true'.

- **handleErrorAndReturn() (Overloaded)**: Handles various error types and displays appropriate error messages. One version accepts only the error type, while the overloaded version accepts an error type and additional information for a more detailed error message.

- **Error Handling**: A comprehensive switch-case block is used to handle different types of errors, each providing a specific error message to help in debugging and user feedback.

9.1   errorhandler.h

Header File

```
1        #ifndef ERRORHANDLER_H
2        #define ERRORHANDLER_H
3
4        #include <curl/curl.h>
5
6        #include <iostream>
7        #include <mutex>
8
9        // Forward declarations due to circular dependency between fileutils.h and errorhandler.h
10       class CsvReader;
11       class ConfigManager;
12       class ConsoleUtils;
13
14       // ErrorHandler class to manage different types of errors
15       class ErrorHandler {
16         private:
17         static std::mutex outputMutex; // Mutex for thread-safe output
18         public:
19         // Enumeration to represent different types of errors that can be encountered
20         enum class ErrorType {
21           INVALID_INPUT_ERROR,
22           INVALID_MENU_INPUT_ERROR,
```

```
23            INPUT_LENGTH_ERROR,
24            INVALID_CHOICE_ERROR,
25            INVALID_INDEX_ERROR,
26            INVALID_INDEX_FORMAT_ERROR,
27            NO_VENUES_SELECTED_ERROR,
28            VENUE_ALREADY_SELECTED_ERROR,
29            EMAIL_AND_SUBJECT_BLANK_ERROR,
30            EMAIL_AND_SUBJECT_WRITE_ATTEMPTS_ERROR,
31            NO_VENUES_SELECTED_FOR_EMAIL_ERROR,
32            EMAIL_ERROR,
33            SUBJECT_LENGTH_ERROR,
34            EMAIL_MESSAGE_LENGTH_ERROR,
35            EMAIL_BLANK_ERROR,
36            SUBJECT_MESSAGE_ERROR,
37            EMAIL_WRITING_ATTEMPTS_ERROR,
38            TEMPLATE_PENDING_ERROR,
39            ATTACHMENT_PATH_ERROR,
40            ATTACHMENT_SIZE_ERROR,
41            SENDER_EMAIL_FORMAT_ERROR,
42            FILESYSTEM_ERROR,
43            CONFIG_OPEN_ERROR,
44            CONFIG_LOAD_ERROR,
45            CONFIG_OPEN_TO_WRITE_ERROR,
46            INVALID_CAPACITY_IN_CSV,
47            INVALID_DATA_IN_CSV,
48            LIBSODIUM_INIT_ERROR,
49            ENCRYPTION_ERROR,
50            DECRYPTION_ERROR,
51            EMAIL_PASSWORD_ENCRYPTION_ERROR,
52            EMAIL_PASSWORD_DECRYPTION_ERROR,
53            EMAIL_PASSWORD_ENCRYPTION_FORMAT_ERROR,
54            LIBCURL_ERROR,
55            SMTP_CONNECTION_ERROR,
56            SMTP_AUTH_ERROR
57        };
58
59        // Function to handle errors and display appropriate messages based on the type of error
60        void handleErrorAndReturn(ErrorType error);
61
62        // Overloaded version with extra information
63        void handleErrorAndReturn(ErrorType error, const std::string& extraInfo);
64
65        // Handles cURL errors and returns a boolean to indicate success/failure
66        bool handleCurlError(CURLcode res);
67    };
68
69    #endif // ERRORHANDLER_H
70
```

Explanation

- **Dependencies**: The header file includes dependencies related to cURL, input/output streams, and mutex handling for thread safety.

- **Forward Declarations**: Forward declarations for 'CsvReader' and 'ConfigManager' are present due to circular dependencies between 'fileutils.h' and 'errorhandler.h'.

- **Class Definition: ErrorHandler**:

    - **Mutex**: A static mutex 'outputMutex' is defined to ensure thread-safe output when handling errors.
    - **ErrorType Enumeration**: Represents different types of errors that the system can encounter. This provides a clear structure to identify and handle different error scenarios.
    - **handleErrorAndReturn()**: Handles various types of errors and displays appropriate error messages. There are two versions of this method: one accepts only the error type, while the overloaded version also takes additional information to provide a more detailed error message.
    - **handleCurlError()**: Checks for any cURL-related errors and provides a feedback mechanism.

10   test_venuesender.cpp

Source File

```
1      // This file is used for testing DO NOT DELETE
2
3      #define CATCH_CONFIG_RUNNER
4      // ... (various includes)
5
6      using namespace std;
7
8      class CinGuard {
9        // ... (definition of CinGuard class)
10     };
11
12     class CoutGuard {
13       // ... (definition of CoutGuard class)
14     };
15
16     // Test groups and test cases follow...
17
18     int main( int argc, char* argv[] ) {
19       // ... (main function for running tests)
20     }
21
```

Explanation

- **Purpose**: This source file is dedicated to unit testing. It is important to note that deleting or altering this file could impact the quality assurance process for the software.

- **Dependencies**: The file includes dependencies to header files associated with various components of the system. It also uses the Catch2 testing framework, as indicated by the `CATCH_CONFIG_RUNNER` preprocessor directive.

- **Utility Classes**: 'CinGuard' and 'CoutGuard' classes are utility classes that help in redirecting the standard input and output streams. These are especially useful in testing scenarios where you want to simulate user input or capture output for verification.

- **Test Groups**: Multiple test groups are present in this source file, each focusing on a specific component or functionality of the system. These groups include 'ConsoleUtils', 'CsvReader', 'ConfigManager', 'MenuManager', 'EmailManager', 'VenueFilter', 'VenueUtilities', and 'EncryptionManager'.

- **Test Cases**: Each test group contains various test cases, which are units of testing that ascertain the correctness of specific functionalities. For instance, the 'ConsoleUtils' test group has a test case for verifying the 'trim' function's behavior.

- **Main Function**: At the end of the source file, a main function is defined to run all the test cases using the Catch2 framework. This main function also handles cleanup tasks after all tests have been executed.

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.