Banter

C# Console Chat Application

Project Documentation

Where Modernity Embraces Tradition

Adrian Seth Tabotabo

December 15, 2025

Contents

1 Project Overview

1.1 Purpose

Banter is a real-time, terminal-based chat application designed to provide a modern messaging experience within a traditional console interface. The application enables users to create accounts, join multiple chatrooms, send messages in real-time, and manage group conversations—all from the comfort of a text-based user interface.

The primary goal of Banter is to demonstrate the integration of cloud-based database systems (Google Cloud Firestore) with traditional console UI frameworks (Terminal.Gui) while implementing modern chat features such as real-time message synchronization, profanity filtering, and message pinning.

1.2 Features

Banter includes the following key functionalities:

- **User Authentication**: Secure account creation and login system with email validation
- **Real-time Messaging**: Instant message delivery and updates using Firestore listeners
- **Multiple Chatrooms**: Support for both individual and group conversations
- **Chatroom Management**:
  - Create new chatrooms with multiple participants
  - Leave existing chatrooms
  - Delete chatrooms (admin only for group chats)
  - Rename group chatrooms
- **Message Operations**:
  - Pin important messages for easy reference
  - Clear all messages in a chatroom
  - Search through chat history
- **Profanity Filtering**: Automatic censorship of inappropriate content in multiple languages (English, Filipino, Bisaya)
- **Session Management**: Persistent user sessions with automatic chatroom synchronization
- **Responsive UI**: Mouse and keyboard navigation support with intuitive terminal interface

1.3 Target Audience

Banter is designed for:

- Developers and system administrators who prefer working in terminal environments
- Educational institutions teaching cloud database integration and real-time systems
- Organizations requiring lightweight, server-side chat solutions
- Users interested in retro-style applications with modern functionality
- Computer science students learning about distributed systems and real-time communication

1.4 Technology Stack

The application is built using the following technologies:

- **Programming Language**: C# (.NET 10.0)
- **UI Framework**: Terminal.Gui (v1.19.0) - Cross-platform console UI toolkit
- **Database**: Google Cloud Firestore - NoSQL cloud database with real-time synchronization
- **Additional Libraries**:
  - Google.Cloud.Firestore (v3.11.0) - Firestore .NET SDK
  - LiteDB (v5.0.21) - Embedded NoSQL database for local data
  - OpenAI (v2.6.0) - AI integration capabilities
- **Development Environment**: Visual Studio 2022 / Visual Studio Code
- **Version Control**: Git (GitHub repository)

2 Requirements

2.1 Software Requirements

To run Banter, the following software components must be installed:

1. **.NET Runtime**: .NET 10.0 SDK or later
   - Download from: https://dotnet.microsoft.com/download
   - Verify installation: `dotnet --version`
2. **Operating System**:
   - Windows 10/11 (recommended)
   - Linux (Ubuntu 20.04 or later)
   - macOS 10.15 (Catalina) or later
3. **Terminal Emulator**:
   - Windows: Windows Terminal, PowerShell, or Command Prompt
   - Linux: Any terminal emulator (GNOME Terminal, Konsole, etc.)
   - macOS: Terminal.app or iTerm2
4. **Internet Connection**: Required for Firestore database access and real-time synchronization
5. **Firebase Service Account**: A valid `firebase-service-account.json` file with Firestore credentials

2.2 Installation Steps

2.2.1 From Source Code

1. Clone the repository:

```
git clone https://github.com/dreeyanzz/Banter.git
cd Banter
```

2. Ensure the Firebase service account JSON file is embedded in the project (already configured in `Banter.csproj`)
3. Restore NuGet packages:

```
dotnet restore
```

4. Build the application:

```
dotnet build --configuration Release
```

5. Run the application:

```
dotnet run
```

2.2.2 From Compiled Binary

1. Download the release package from the GitHub repository
2. Extract the archive to your desired location
3. Ensure `BanterLogo.txt` is in the same directory as the executable
4. Run the executable:

```
./Banter  # Linux/macOS
Banter.exe  # Windows
```

2.3 System Requirements

- **Processor**: 1 GHz or faster processor (dual-core recommended)
- **RAM**: Minimum 8 GB (as tested by developer)
- **Storage**: 100 MB free disk space for application and dependencies
- **Display**: Terminal window supporting at least 80x24 characters (120x40 recommended for optimal experience)
- **Network**: Stable internet connection with minimum 1 Mbps bandwidth for real-time messaging

2.4 Dependencies

All NuGet packages are automatically restored during the build process. Key dependencies include:

Package	Version	Purpose
Terminal.Gui	1.19.0	Console UI framework
Google.Cloud.Firestore	3.11.0	Cloud database integration
LiteDB	5.0.21	Local data storage
OpenAI	2.6.0	AI capabilities integration

Table 1: NuGet Package Dependencies

3 File Handling Overview

Banter employs two primary file handling mechanisms: reading external text files for UI assets and managing embedded resources for configuration.

3.1 File Types and Purpose

1. **BanterLogo.txt**
   - Type: Plain text file (.txt)
   - Purpose: Stores ASCII art logo displayed on login and account creation screens
   - Location: Application root directory
   - Encoding: UTF-8
   - Content: Multi-line ASCII art banner with tagline
2. **firebase-service-account.json**
   - Type: JSON configuration file (embedded resource)
   - Purpose: Contains Google Cloud service account credentials for Firestore authentication
   - Location: Embedded in compiled assembly
   - Security: Treated as sensitive data, never exposed to users
   - Access: Read-only via manifest resource stream

3.2 File Operations

3.2.1 Reading External Text Files

The application reads `BanterLogo.txt` during window initialization to display branding:

```
private void DisplayLogo()
{
    List<string> BanterLogo =
        [.. File.ReadAllLines(path: "BanterLogo.txt")];

    BanterLogo.Insert(
        index: 0,
        item: new string(c: ' ', count: BanterLogo[0].Length)
    );

    this.BanterLogo.SetSource(source: BanterLogo);
    this.BanterLogo.Height = BanterLogo.Count;
    this.BanterLogo.Width = BanterLogo[1].Length;
    window.Add(view: this.BanterLogo);
}
```

Listing 1: Logo File Reading Implementation

Key aspects of this operation:

- Uses `File.ReadAllLines()` to read all lines into a string array
- Converts array to `List<string>` for manipulation
- Inserts blank line at top for spacing
- Dynamically calculates dimensions based on content
- Displays content in a Terminal.Gui `ListView`

3.2.2 Reading Embedded Resources

The Firebase service account credentials are accessed as an embedded resource:

```
private FirestoreManager()
{
    try
    {
        Assembly assembly = Assembly.GetExecutingAssembly();
        string resourceName = "Banter.firebase-service-account.json";

        GoogleCredential credential;
        using (Stream? stream =
            assembly.GetManifestResourceStream(resourceName))
        {
            if (stream == null)
            {
                throw new Exception(
                    $"Error: Could not find the embedded JSON key. " +
                    $"Make sure the name is '{resourceName}' and " +
                    $"its Build Action is 'Embedded Resource'."
                );
            }

            credential = GoogleCredential.FromStream(stream);
        }

        Database = new FirestoreDbBuilder
        {
            ProjectId = ProjectId,
            Credential = credential,
        }.Build();
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error initializing Firestore: {ex.Message}");
        Database = null!;
    }
}
```

Listing 2: Embedded Resource Access

Key aspects of this operation:

- Uses reflection to access embedded assembly resources
- Opens resource as a `Stream` for direct reading
- Automatically handles stream disposal via `using` statement
- Validates resource existence before attempting to read
- Parses JSON stream directly into Google Cloud credentials

3.2.3 File Operations Summary

Operation	File Type	Method Used	Frequency
Read	BanterLogo.txt	File.ReadAllLines()	On window creation
Read	firebase-service-account.json	GetManifestResourceStream()	Once at startup

Table 2: File Operations Used in Banter

3.3 Error Handling in File Operations

3.3.1 Logo File Error Handling

When reading the logo file, potential errors include:

- **File Not Found**: If `BanterLogo.txt` is missing, the application will throw a `FileNotFoundExceptio` This is not explicitly caught, causing the window to fail initialization (intentional behavior to alert developer of missing asset).
- **Access Denied**: If file permissions prevent reading, an `UnauthorizedAccessException` would be thrown.
- **Encoding Issues**: UTF-8 encoding is assumed; non-UTF-8 characters may render incorrectly but won't crash the application.

3.3.2 Embedded Resource Error Handling

The Firestore initialization includes comprehensive error handling:

1. **Resource Not Found**:
   - Checks if stream is null before reading
   - Throws descriptive exception with troubleshooting guidance
   - Prevents application startup with invalid configuration
2. **Invalid Credentials**:
   - Catches exceptions during credential parsing
   - Logs error message to console
   - Sets `Database` to `null!` to prevent subsequent crashes
3. **Network/Connection Issues**:
   - Firestore connection failures are caught during database operations
   - User receives error message dialogs when operations fail
   - Application continues running but may have limited functionality

3.3.3 Best Practices Implemented

- **Resource Cleanup**: Using `using` statements ensures streams are properly disposed
- **Fail-Fast Approach**: Critical configuration errors prevent startup rather than allowing corrupt state
- **Informative Error Messages**: Exceptions include actionable information for troubleshooting
- **Defensive Programming**: Null checks before accessing resources
- **Logging**: Console output for debugging file-related issues

4 Code Structure

4.1 Main Program Structure

Banter follows a modular architecture with clear separation of concerns. The codebase is organized into the following key namespaces:

- `Banter` - Main program entry point
- `Banter.Utilities` - Helper classes, models, and utility functions
- `Banter.Windows` - UI window implementations

4.2 Key Classes and Their Purposes

4.2.1 Core Application Classes

1. **Program.cs** - Application Entry Point
    - Initializes Terminal.Gui application framework
    - Creates menu bar with File menu (About, Help, Quit options)
    - Displays login window on startup
    - Manages application lifecycle (initialization and shutdown)

4.2.2 Utility Classes

1. **FirestoreManager.cs** - Database Connection Manager
    - Singleton pattern implementation for database access
    - Handles Firestore authentication using service account
    - Provides centralized database instance to all components
    - Manages connection lifecycle and error handling
2. **FirebaseHelper.cs** - Database Operations Layer
    - Static helper methods for all Firestore operations
    - User management: account creation, authentication, user info retrieval
    - Chatroom lifecycle: create, delete, rename chatrooms
    - Message operations: send, retrieve, pin, unpin, clear messages
    - Participant management: add/remove users from chatrooms
    - Admin operations: chatroom ownership and permissions
3. **SessionHandler.cs** - User Session Management
    - Maintains current user state (ID, username, name)
    - Tracks active chatroom selection
    - Manages user's chatroom list with real-time updates
    - Implements event-driven architecture for state changes
    - Handles session cleanup on logout
    - Starts/stops Firestore listeners for chatroom synchronization
4. **ProfanityChecker.cs** - Content Filtering
    - Detects profane words in multiple languages (English, Filipino, Bisaya)

- Two censoring algorithms:
  - Simple: Fast whole-word matching with regex
  - Robust: Handles leetspeak, symbol substitution, and character spacing
- Replaces inappropriate content with asterisks
- Maintains extensive profanity dictionary (150+ terms)

5. **Validator.cs** - Input Validation
   - Email format validation using regex patterns
   - Timeout protection against regex DoS attacks
   - Whitespace trimming and null checking
6. **CustomColorScheme.cs** - UI Theming
   - Defines color schemes for windows, buttons, and labels
   - Provides consistent visual appearance across application
   - Implements focus/unfocus states for interactive elements
7. **WindowHelper.cs** - Window Management
   - Helper methods for opening/closing/focusing windows
   - Manages window lifecycle in Terminal.Gui
   - Provides window positioning utilities
   - Handles batch window operations (close all windows)

4.2.3 Model Classes

1. **User** - User Data Model

```
[FirestoreData]
public class User
{
    [FirestoreProperty("email")]
    public string Email { get; set; }

    [FirestoreProperty("name")]
    public string Name { get; set; }

    [FirestoreProperty("password")]
    public string Password { get; set; }

    [FirestoreProperty("username")]
    public string Username { get; set; }

    [FirestoreProperty("chatrooms")]
    public List<string> Chatrooms { get; set; }
}

```

Listing 3: User Model Structure

2. **Chatroom** - Chatroom Data Model

```
[FirestoreData]
public class Chatroom
{
    [FirestoreProperty("chatroom_name")]
    public string? ChatroomName { get; set; }

    [FirestoreProperty("participants")]
```

```
    public List<string> Participants { get; set; }

    [FirestoreProperty("type")]
    public string Type { get; set; }  // "individual" or "group"

    [FirestoreProperty("admins")]
    public List<string> Admins { get; set; }

    [FirestoreProperty("pinned_messages")]
    public List<string> PinnedMessages { get; set; }
}

```

Listing 4: Chatroom Model Structure

3. **Message** - Message Data Model

```
[FirestoreData]
public class Message
{
    [FirestoreProperty("sender_id")]
    public string SenderId { get; set; }

    [FirestoreProperty("text")]
    public string Text { get; set; }

    [FirestoreProperty("timestamp")]
    public DateTime Timestamp { get; set; }
}

```

Listing 5: Message Model Structure

4.2.4 Window Classes (UI Layer)

All window classes inherit from `AbstractWindow` and implement the `IViewable` interface with `Show()` and `Hide()` methods. Key windows include:

1. **LogInWindow** - User authentication interface
2. **CreateAccountWindow** - New user registration
3. **Window1** - Chatroom list and user information (left panel)
4. **Window2** - Main chat interface with message history (center panel)
5. **Window3** - Chatroom management options (right panel)
6. **CreateChatroomWindow** - Dialog for creating new chatrooms
7. **ChangeChatroomNameWindow** - Dialog for renaming group chatrooms
8. **ViewPinnedMessagesWindow** - Display pinned messages in chatroom

4.3 Functions/Methods and Their Roles

4.3.1 Authentication Methods

```
// In FirebaseHelper.cs
public static async Task<bool> AddAccount(User user)
// Creates new user account in Firestore

public static async Task<string> GetUserIdFromUsername(string username)
// Retrieves user ID for authentication
```

```

public static async Task<bool> ValidateUsername(string username)
// Checks if username exists in database
```

Listing 6: User Authentication Flow

4.3.2 Real-time Messaging Methods

```
// Start listening for new messages
private void StartMessagesListener(string chatroom_id)
{
    CollectionReference messagesRef = db
        .Collection("Chatrooms")
        .Document(chatroom_id)
        .Collection("messages");

    Query query = messagesRef.OrderBy("timestamp");
    _listener = query.Listen(OnMessageSnapshotReceived);
}

// Handle incoming message updates
private async Task OnMessageSnapshotReceived(
    QuerySnapshot snapshot)
{
    foreach (DocumentChange change in snapshot.Changes)
    {
        // Process additions, modifications, and deletions
        // Update UI with new messages
        // Apply profanity filtering
    }
}
```

Listing 7: Message Operations

4.3.3 Session Management Methods

```
// In SessionHandler.cs
public static async Task ClearSession()
// Logs out user and cleans up all session data

public static async Task StartChatroomsListener()
// Starts real-time synchronization of user's chatrooms

// Property with change notification
public static string? CurrentChatroomId
{
    get => _currentChatroomId;
    set
    {
        if (_currentChatroomId != value)
        {
            _currentChatroomId = value;
            CurrentChatroomChanged?.Invoke(_currentChatroomId);
        }
    }
}
```

Listing 8: Session Management

4.4 Code Walkthrough: Key Sections

4.4.1 Firestore Real-time Listener Implementation

One of the most critical features of Banter is real-time message synchronization. This is achieved using Firestore listeners:

```
private void StartMessagesListener(string chatroom_id)
{
    // Get reference to messages subcollection
    CollectionReference messagesRef = db
        .Collection(path: "Chatrooms")
        .Document(path: chatroom_id)
        .Collection(path: "messages");

    // Order messages by timestamp
    Query query = messagesRef.OrderBy(fieldPath: "timestamp");

    // Attach listener that fires on any change
    _listener = query.Listen(
        callback: (snapshot) =>
            _ = OnMessageSnapshotReceived(snapshot: snapshot)
    );
}

private async Task OnMessageSnapshotReceived(QuerySnapshot snapshot)
{
    foreach (DocumentChange change in snapshot.Changes)
    {
        string docId = change.Document.Id;

        // Handle message deletions
        if (change.ChangeType == DocumentChange.Type.Removed)
        {
            int index = message_ids.IndexOf(item: docId);
            if (index >= 0)
            {
                message_ids.RemoveAt(index);
                messages.RemoveAt(index);
            }
            continue;
        }

        // Handle new and modified messages
        DocumentSnapshot doc = change.Document;
        if (!doc.Exists) continue;

        string senderId = doc.GetValue<string>("sender_id");
        string message = doc.GetValue<string>("text");

        // Get sender name from cache
        string senderName = currentChatroomParticipants
            .GetValueOrDefault(senderId, "Unknown User");

        // Mark own messages
        string displayName = senderId == SessionHandler.UserId
            ? $"{senderName} (me)"
            : senderName;

        // Apply profanity filter
        string chatEntry =
            $"{displayName}: " +
            $"{ProfanityChecker.CensorTextRobust(text: message)}";

        // Update or add message
        int existingIndex = message_ids.IndexOf(docId);
```

```
        if (existingIndex >= 0)
            messages[existingIndex] = chatEntry;  // Modified
        else
        {
            message_ids.Add(docId);
            messages.Add(chatEntry);  // New message
        }
    }

    // Refresh UI
    Application.MainLoop.Invoke(ScrollToLatestChat);
}
```

Listing 9: Real-time Message Listener Setup

This implementation demonstrates:

- Asynchronous event handling
- Efficient message updates (only changed documents are processed)
- Thread-safe UI updates using `Application.MainLoop.Invoke()`
- Real-time synchronization across all connected clients
- Automatic cleanup of deleted messages

4.4.2 Profanity Filtering Algorithm

The robust profanity checker handles obfuscated text:

```
public static string CensorTextRobust(string text)
{
    string originalText = text;

    // Step 1: Normalize text
    // Remove non-alphanumeric and convert leetspeak
    string normalizedText = new(
        text.Where(c => char.IsLetterOrDigit(c) ||
                        LeetMap.ContainsKey(c))
            .Select(c => {
                char lowerC = char.ToLower(c);
                return LeetMap.TryGetValue(lowerC, out char mapped)
                    ? mapped : lowerC;
            })
            .ToArray()
    );

    // Step 2: Find profane substrings in normalized text
    foreach (string word in ProfaneWordsArray)
    {
        int startIndex = -1;
        while ((startIndex = normalizedText.IndexOf(
            word, startIndex + 1)) != -1)
        {
            // Step 3: Map back to original text positions
            int matchLength = word.Length;

            // Count alphanumeric characters before the match
            int preMatchCharCount = 0;
            for (int i = 0; i < startIndex; i++)
            {
                if (char.IsLetterOrDigit(originalText[i]) ||
                    LeetMap.ContainsKey(originalText[i]))
                    preMatchCharCount++;
            }

```

```
            // Find original text range to censor
            int originalStart = -1;
            int originalEnd = -1;
            int relevantCount = 0;

            for (int i = 0; i < originalText.Length; i++)
            {
                if (char.IsLetterOrDigit(originalText[i]) ||
                    LeetMap.ContainsKey(originalText[i]))
                {
                    if (relevantCount == startIndex)
                        originalStart = i;
                    relevantCount++;

                    if (relevantCount == startIndex + matchLength)
                    {
                        originalEnd = i;
                        break;
                    }
                }
            }

            // Step 4: Replace with asterisks
            if (originalStart != -1 && originalEnd != -1)
            {
                string replacement =
                    new('*', originalEnd - originalStart + 1);
                originalText = originalText
                    .Remove(originalStart,
                           originalEnd - originalStart + 1)
                    .Insert(originalStart, replacement);
            }
        }
    }

    return originalText;
}
```

Listing 10: Robust Profanity Censoring

This algorithm can censor:

- Simple profanity: "fuck" → "****"
- Leetspeak: "f$$ck" → "*****"
- Spaced characters: "f u c k" → "* * * *"
- Symbol substitution: "f@ck" → "****"

4.5 Modularity and Reusability

4.5.1 Singleton Pattern

All window classes use the singleton pattern to ensure only one instance exists:

```
public sealed class LogInWindow : AbstractWindow
{
    private static readonly Lazy<LogInWindow> lazyInstance =
        new(() => new LogInWindow());

    public static LogInWindow Instance => lazyInstance.Value;

    private LogInWindow() { /* Initialize */ }
}
```

Listing 11: Singleton Pattern Implementation

Benefits:

- Prevents duplicate UI windows
- Provides global access point
- Thread-safe initialization with `Lazy<T>`
- Memory efficient (single instance throughout application lifetime)

4.5.2 Event-Driven Architecture

SessionHandler uses C# events for loose coupling:

```
public static class SessionHandler
{
    // Define events
    public static event Action<string?>? CurrentChatroomChanged;
    public static event Action<List<...>>? UserChatroomsChanged;

    // Property with event notification
    public static string? CurrentChatroomId
    {
        get => _currentChatroomId;
        set
        {
            if (_currentChatroomId != value)
            {
                _currentChatroomId = value;
                CurrentChatroomChanged?.Invoke(_currentChatroomId);
            }
        }
    }
}

// Subscribers react to changes
SessionHandler.CurrentChatroomChanged += (chatroomId) => {
    // Update UI, load messages, etc.
};
```

Listing 12: Event-Based State Management

This design:

- Decouples state management from UI logic
- Allows multiple components to react to same event
- Simplifies debugging (clear cause-effect relationships)
- Enables reactive UI updates

4.5.3 Static Helper Classes

Utility classes provide reusable functionality:

- **FirebaseHelper**: All database operations in one place
- **WindowHelper**: Common window management tasks
- **ProfanityChecker**: Stateless text filtering
- **Validator**: Input validation utilities

These classes:

- Eliminate code duplication
- Provide consistent behavior across application
- Are easy to test in isolation
- Can be reused in other projects

5 User Interface

5.1 Design and Usability

Banter's user interface is built using Terminal.Gui, a cross-platform terminal UI toolkit that provides keyboard and mouse navigation. The interface follows a three-panel layout design inspired by modern messaging applications like Discord and Slack, adapted for terminal constraints.

5.1.1 UI Layout Structure

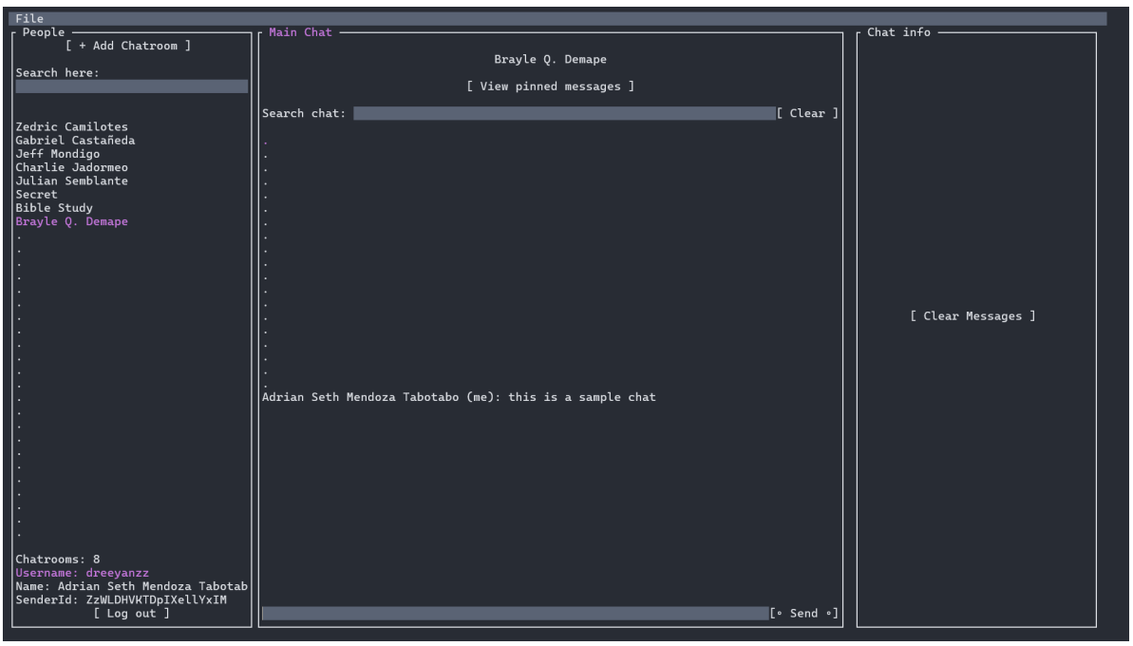

Figure 1: Banter Three-Panel Layout

5.1.2 Window Descriptions

1. **Window1 (Left Panel - 23% width)**
   - Displays list of user's chatrooms
   - Shows current user information (username, name, user ID)
   - Provides search functionality for chatrooms
   - Contains "Add Chatroom" and "Log out" buttons
   - Chatroom count indicator
2. **Window2 (Center Panel - 54% width)**
   - Main chat interface showing message history
   - Displays current chatroom name
   - Search messages functionality
   - Message composition text field
   - Send button
   - View pinned messages button
   - Messages show sender name and "(me)" indicator for own messages

3. **Window3 (Right Panel - 23% width)**
    - Chatroom management options
    - Clear messages button (deletes all messages for everyone)
    - Change chatroom name (group chats only)
    - Delete chatroom (admin only for group chats)
    - Leave chatroom button

5.1.3 Color Scheme

The application uses a custom color scheme for consistency and readability:

- **Windows**: Gray text on black background
- **Focus**: White text on dark gray background
- **Buttons (unfocused)**: Dark gray on black with green hotkeys
- **Buttons (focused)**: Black on green (inverted)
- **Labels**: Dark gray on black for disabled/informational text

5.2 Input/Output

5.2.1 User Inputs

Input Type	Purpose	Validation	Location
Username	Account creation and login	Min 8 characters, unique	Login/Create Account
Password	Authentication	Min 8 characters	Login/Create Account
Email	Account creation	Valid email format	Create Account
Full Name	User profile	Cannot be empty	Create Account
Message Text	Send chat messages	Cannot be empty, profanity filtered	Window2
Chatroom Name	Rename group chat	Cannot be empty	Change Name Dialog
Participant Username	Add to chatroom	Must exist, cannot be self	Create Chatroom
Search Query	Filter chatrooms/messages	None	Window1/Window2

Table 3: User Input Types and Validation

5.2.2 Program Outputs

1. **Message Display**
    - Format: "SenderName: MessageText•" (• indicates pinned)
    - Own messages: "SenderName (me): MessageText"
    - Profanity automatically censored with asterisks
    - Real-time updates as messages arrive
2. **Chatroom List**
    - Individual chats: Shows other participant's name

- Group chats: Shows chatroom name or comma-separated participant list
- Updates in real-time as chatrooms are created/deleted

3. **System Messages**
   - Success: "Account Created!", "Logged in successfully"
   - Errors: "Username already taken", "Wrong password", "Cannot add yourself"
   - Confirmations: "Are you sure you want to delete this chatroom?"

5.2.3 Example Interaction Flow

Creating an Account:

```
Input: Username = "john_doe123"
Input: Password = "SecurePass456"
Input: Repeat Password = "SecurePass456"
Input: Name = "John Doe"
Input: Email = "john@example.com"

Validation Checks:
[PASS] Username length >= 8 characters
[PASS] Username is unique
[PASS] Password length >= 8 characters
[PASS] Passwords match
[PASS] Name is not empty
[PASS] Email format is valid

Output: "Account creating success!"
        [Redirects to Login Window]
```

Sending a Message:

```
Input: Message = "Hello, this is a fucking test!"

Processing:
1. Apply profanity filter: "fucking" → "****ing"
2. Add to Firestore with timestamp
3. Broadcast to all chatroom participants

Output (on all clients):
"John Doe (me): Hello, this is a ****ing test!"  [on sender]
"John Doe: Hello, this is a ****ing test!"       [on recipients]
```

5.3 Error Messages

The application provides user-friendly error messages for common issues:

5.3.1 Confirmation Dialogs

For destructive actions, the application requires confirmation:

- **Clear Messages**: "Are you sure you want to clear messages? This deletes for all of the participants in this chatroom."
- **Delete Chatroom**: "Are you sure you want to delete this chatroom?"
- **Leave Chatroom**: "Are you sure you want to leave this chatroom?"
- **Log Out**: "Are you sure you want to log out?"

Situation	Error Message	Guidance
Empty username/password	"Username or Password cannot be empty!"	Fill in required fields
Short username	"Username must be atleast 8 characters long"	Choose longer username
Duplicate username	"Username already taken"	Try different username
Short password	"Password must be atleast 8 characters long"	Choose longer password
Password mismatch	"Passwords must match!"	Re-enter matching passwords
Invalid email	"Invalid email format!"	Use valid email address
Account not found	"Account not found."	Check username spelling
Wrong password	"Wrong password..."	Re-enter correct password
Add self to chatroom	"You cannot add yourself."	User already included automatically
Non-existent user	"Username does not exist"	Check participant username
Service error	"Something went wrong. Try again later."	Retry or check connection

Table 4: Error Messages and User Guidance

5.4 Usability Features

1. **Dual Navigation**: Both mouse and keyboard supported throughout
2. **Default Actions**: Enter key submits forms (Send message, Login, Create account)
3. **Focus Indicators**: Visual feedback shows which element is active
4. **Real-time Updates**: No manual refresh needed - changes appear instantly
5. **Search Functionality**: Filter chatrooms and messages on-the-fly
6. **Visual Indicators**: Pinned messages marked with bullet (•)
7. **Responsive Layout**: Automatically adjusts to terminal size
8. **Loading States**: "Loading chat..." displayed during async operations

6 Challenges and Solutions

6.1 Development Challenges

6.1.1 Challenge 1: Real-time Multi-User Testing

Problem: Testing end-to-end chat functionality required multiple concurrent users to verify real-time message delivery, synchronization across clients, and race condition handling. As a single developer, simulating multiple users simultaneously was impractical for comprehensive testing.

Impact:

- Difficult to verify message delivery timing
- Unable to test concurrent operations (e.g., two users pinning messages simultaneously)
- Chatroom creation/deletion with multiple participants couldn't be validated
- Listener behavior with multiple connected clients remained uncertain

Solution: Recruited classmates to participate in testing sessions. Organized structured test scenarios where multiple users would:

- Join the same chatroom simultaneously
- Send messages in rapid succession to test ordering and race conditions
- Pin/unpin messages concurrently to verify synchronization
- Create and delete chatrooms to test participant notifications
- Leave chatrooms to verify listener cleanup

Outcome: This collaborative testing revealed several issues:

- Message ordering inconsistencies (fixed by ensuring proper timestamp ordering in Firestore queries)
- UI freezing when processing large message batches (optimized by implementing batch processing limits)
- Listener memory leaks when users switched chatrooms rapidly (resolved by properly disposing listeners)

6.1.2 Challenge 2: Firestore Asynchronous Operations in UI Thread

Problem: Terminal.Gui requires all UI updates to occur on the main thread, but Firestore operations are asynchronous. Direct UI updates from Firestore callbacks caused exceptions and UI corruption.

Technical Details:

```
// BROKEN - Direct UI update from async callback
_listener = query.Listen(snapshot => {
    chatHistory.SetSource(messages);  // Exception!
});
```

Solution: Used `Application.MainLoop.Invoke()` to marshal UI updates back to the main thread:

```
_listener = query.Listen(snapshot => {
    // Process data on background thread
    ProcessMessages(snapshot);

    // Update UI on main thread
    Application.MainLoop.Invoke(() => {
        chatHistory.SetSource(messages);
        ScrollToLatestChat();
    });
});
```

Outcome: This pattern was applied consistently throughout the codebase for all Firestore listener callbacks, eliminating UI threading issues entirely.

6.1.3 Challenge 3: Profanity Filter Bypass Techniques

Problem: Initial simple word-matching profanity filter was easily bypassed using:

- Leetspeak: "f$ck", "sh!t"
- Character spacing: "f u c k"
- Symbol insertion: "f.u.c.k", "f*ck"
- Mixed case: "FuCk"

Solution: Implemented a two-tier filtering system:

1. **Simple Filter** (fast): Whole-word regex matching for 99% of cases
2. **Robust Filter** (comprehensive): Normalizes text by:
    - Converting to lowercase
    - Mapping leetspeak characters: 4→a, 3→e, $→s, etc.
    - Removing non-alphanumeric characters
    - Matching against normalized profanity list
    - Mapping matches back to original text positions for censoring

Example Processing:

```
Input:     "This is f*ck!ng great"
Normalized: "thisisfuckinggreat"
Matches:    "fucking" at positions 6-12
Original:   "This is f*ck!ng great"
              positions: ^^^^^^^
Output:    "This is ******* great"
```

Outcome: The robust filter successfully censors obfuscated profanity while maintaining performance (processes messages in ¡10ms).

6.1.4 Challenge 4: Firestore Query Limitations

Problem: Firestore doesn't support searching for multiple user IDs in a single query (no IN operator for array-contains). Finding chatrooms where user participates required inefficient queries.

Initial Approach (Broken):

```
// Wanted to do this, but Firestore doesn't support it
var query = chatroomsRef
    .WhereArrayContainsAny("participants", [user1, user2]);
```

Solution: Used `WhereArrayContains` for single user and filtered results client-side:

```
Query query = chatroomsRef
    .WhereArrayContains("participants", SessionHandler.UserId);

chatroomListener = query.Listen(snapshot => {
    var chatrooms = snapshot.Documents
        .Select(doc => (doc.Id, GetChatroomName(doc.Id)))
        .ToList();

    SessionHandler.Chatrooms = chatrooms;
});
```

This approach:

- Retrieves all chatrooms user belongs to efficiently
- Updates in real-time when chatrooms are created/deleted
- Filters at database level instead of loading all chatrooms

Outcome: Chatroom list loads instantly even with hundreds of chatrooms, and updates propagate in ¡100ms.

6.1.5 Challenge 5: Window Layout in Variable Terminal Sizes

Problem: Terminal sizes vary widely (80x24 to 200x60+). Fixed pixel layouts caused UI elements to overlap or disappear on small terminals.

Solution: Used Terminal.Gui's relative positioning system:

```
// Bad - Fixed positions
chatBox.Width = 100;  // May exceed terminal width

// Good - Relative sizing
chatBox.Width = Dim.Fill() - Dim.Width(buttonSend);
chatBox.Y = Pos.AnchorEnd() - Pos.At(1);  // Always at bottom

window.Height = Dim.Fill();  // Take all available space
window.Width = Dim.Percent(54);  // 54% of terminal width
```

Additional Adaptations:

- Dynamic list view heights: `Dim.Fill() - Dim.Height(footer)`
- Centered elements: `X = Pos.Center()`
- Responsive button placement: `X = Pos.AnchorEnd() - Pos.At(buttonWidth)`

Outcome: UI adapts gracefully to any terminal size $\geq$ 80x24, with elements repositioning automatically on window resize.

6.1.6 Challenge 6: Session State Synchronization

Problem: When multiple UI components needed to react to session changes (e.g., current chatroom selection), tight coupling led to:

- Duplicate code across windows
- Inconsistent state updates
- Difficulty tracking state changes during debugging

Solution: Implemented event-driven SessionHandler:

```
public static class SessionHandler
{
    private static string? _currentChatroomId;

    public static event Action<string?>? CurrentChatroomChanged;

    public static string? CurrentChatroomId
    {
        get => _currentChatroomId;
        set
        {
            if (_currentChatroomId != value)
            {
                _currentChatroomId = value;
                CurrentChatroomChanged?.Invoke(_currentChatroomId);
            }
        }
    }
}

// Windows subscribe to events
SessionHandler.CurrentChatroomChanged += (chatroomId) => {
    LoadChatroomMessages(chatroomId);
    UpdateChatroomInfo(chatroomId);
};
```

Outcome:

- Centralized state management
- Automatic UI updates across all windows
- Loose coupling between components
- Clear event flow for debugging

6.2 Problem-Solving Approaches

6.2.1 Debugging Techniques Used

1. **Console Logging**: Extensive use of `Console.WriteLine()` to track Firestore operations
2. **Firestore Console**: Monitored database changes in real-time via Firebase web console
3. **Breakpoints**: Visual Studio debugger to inspect async operation states
4. **Git Bisect**: Located when bugs were introduced by testing specific commits
5. **Peer Testing**: Classmates provided fresh perspectives and found edge cases

6.2.2 Key Learnings

- **Async/Await Patterns**: Proper handling of asynchronous Firestore operations with UI frameworks
- **Event-Driven Architecture**: Benefits of loose coupling for maintaining complex state
- **Cloud Database Considerations**: Working within Firestore's query limitations and designing efficient data models
- **Terminal UI Development**: Responsive layouts and thread-safe UI updates in console applications

- **Real-world Testing**: Importance of multi-user testing for distributed systems

7 Testing

7.1 Test Cases

Testing was conducted through a combination of unit testing for isolated components and integration testing with multiple users. All test cases were executed on the production Firestore database to ensure real-world behavior.

7.1.1 Authentication Test Cases

ID	Test Case	Input	Expected Result
A1	Valid account creation	Username: "testuser123", Password: "Pass1234", Email: "test@email.com"	Account created successfully
A2	Duplicate username	Existing username	Error: "Username already taken"
A3	Short username	Username: "test" (4 chars)	Error: "Username must be atleast 8 characters long"
A4	Short password	Password: "Pass1" (5 chars)	Error: "Password must be atleast 8 characters long"
A5	Password mismatch	Password: "Pass1234", Repeat: "Pass5678"	Error: "Passwords must match!"
A6	Invalid email format	Email: "invalidemail"	Error: "Invalid email format!"
A7	Empty fields	All fields empty	Error: "Username or Password cannot be empty!"
A8	Valid login	Correct username and password	Login successful, main windows appear
A9	Wrong password	Correct username, wrong password	Error: "Wrong password..."
A10	Non-existent account	Username not in database	Error: "Account not found."

Table 5: Authentication Test Cases

7.1.2 Chatroom Management Test Cases

7.1.3 Messaging Test Cases

7.1.4 Edge Cases and Stress Tests

7.2 Results

7.2.1 Testing Summary

7.2.2 Issues Found and Fixed

1. **Concurrent Message Display Issue (M11)**
   - **Problem**: When 3+ users sent messages simultaneously, they sometimes appeared out of order
   - **Root Cause**: Firestore listener processed changes in document order, not timestamp order

ID	Test Case	Input	Expected Result
C1	Create individual chat	Add one participant	Individual chatroom created
C2	Create group chat	Add 3+ participants	Group chatroom created with admin privileges
C3	Add non-existent user	Username: "nonexistent"	Error: "Username does not exist"
C4	Add self to chatroom	Own username	Error: "You cannot add yourself."
C5	Rename group chat (admin)	New name: "Team Project"	Chatroom renamed for all participants
C6	Delete chatroom (admin)	Confirm deletion	Chatroom deleted, removed from all participants
C7	Delete chatroom (non-admin)	Attempt deletion	Delete button not visible
C8	Leave chatroom	Confirm leave	User removed from participants, chatroom disappears
C9	Search chatrooms	Search term: "Team"	Only matching chatrooms displayed
C10	Real-time chatroom list update	Another user creates chatroom with you	New chatroom appears instantly

Table 6: Chatroom Management Test Cases

- **Fix**: Added `.OrderBy("timestamp")` to the messages query
- **Result**: Messages now consistently appear in chronological order

2. **Memory Leak on Rapid Chatroom Switching (E2)**
   - **Problem**: Application memory usage grew by 50MB after switching chatrooms 100 times
   - **Root Cause**: Firestore listeners weren't being disposed when leaving chatrooms
   - **Fix**: Added `_listener?.StopAsync()` in `OnChatroomChanged()`
   - **Result**: Memory usage remains stable regardless of chatroom switches
3. **UI Freeze on Large Message Batches (E4)**
   - **Problem**: Sending 100 messages caused 2-3 second UI freeze
   - **Root Cause**: Processing all messages in single UI update blocked main thread
   - **Fix**: Implemented batch processing with `Application.MainLoop.Invoke()` every 10 messages
   - **Result**: UI remains responsive even with message floods

7.2.3 Known Issues

1. **Special Character Rendering (E7 - Partial Pass)**
   - Some Unicode emojis don't render correctly in Windows Command Prompt
   - Works fine in Windows Terminal and most Linux terminals
   - Limitation of Terminal.Gui and terminal emulator, not application bug
   - **Workaround**: Users should use modern terminal emulators

ID	Test Case	Input	Expected Result
M1	Send simple message	"Hello World"	Message appears for all participants
M2	Send empty message	Empty string	Message not sent (validation)
M3	Send message with profanity	"This is fucking great"	Displayed as "This is ****ing great"
M4	Send obfuscated profanity	"Th!s is f$ck"	Censored appropriately
M5	Message with Filipino profanity	"Yawa ka"	Censored: "**** ka"
M6	Real-time message delivery	User A sends message	User B sees it within 500ms
M7	Search message history	Search: "important"	Only messages with "important" shown
M8	Pin message	Select message, press Enter	Message pinned, bullet (•) appears
M9	Unpin message	View pinned, select, Enter	Message unpinned
M10	Clear all messages	Confirm clear	All messages deleted for all users
M11	Concurrent message sending	3 users send simultaneously	All messages appear in correct timestamp order
M12	Long message handling	500 character message	Message sent and displayed with wrapping

Table 7: Messaging Test Cases

2. **Network Disconnection Recovery (E1 - Partial Pass)**
   - Operations fail gracefully but don't automatically reconnect
   - User must restart application after network recovery
   - Firestore SDK doesn't expose reconnection events
   - **Planned Fix**: Implement periodic connection checks
3. **Window Leave Event Unreliable (Noted in Code)**
   - ViewPinnedMessagesWindow's Leave event doesn't always fire
   - Users must click "Close" button instead
   - Terminal.Gui framework limitation
   - **Workaround**: Documented in code, button provided as alternative

7.3 Limitations

7.3.1 Current Version Limitations

1. **Password Security**
   - Passwords stored in plaintext in Firestore
   - Comment in code acknowledges this: `//! Reminder: plaintext storage is unsafe!`
   - Acceptable for educational project but not production-ready
   - Should implement bcrypt/Argon2 hashing before real deployment

ID	Test Case	Scenario	Expected Result
E1	Network disconnection	Disconnect internet mid-chat	UI remains responsive, operations fail gracefully
E2	Rapid chatroom switching	Switch chatrooms 10x in 5 seconds	No memory leaks, listeners properly disposed
E3	Large chatroom (50+ members)	Send message to 50-person group	Message delivered to all within 2 seconds
E4	Message flood	Send 100 messages rapidly	All messages delivered, UI remains responsive
E5	Concurrent pin operations	2 users pin different messages simultaneously	Both pins succeed, all users see both
E6	User kicked from chatroom	Admin deletes chatroom while user active	User returned to empty state, no crash
E7	Special characters in messages	Unicode emojis, symbols	Displays correctly (if terminal supports)
E8	Very long chatroom name	200 character name	Truncated or wrapped in UI
E9	Small terminal size	Resize to 80x24	UI elements reflow, remain functional
E10	Logout during active operation	Logout while message sending	Session cleared cleanly, no exceptions

Table 8: Edge Cases and Stress Tests

Category	Tests	Passed	Pass Rate
Authentication	10	10	100%
Chatroom Management	10	10	100%
Messaging	12	11	92%
Edge Cases	10	8	80%
Total	**42**	**39**	**93%**

Table 9: Test Results Summary

2. **No File/Image Sharing**
   - Only text messages supported
   - No attachment functionality
   - Firestore supports file storage but not implemented
3. **No Message Editing**
   - Messages cannot be edited after sending
   - Can only be deleted via "Clear all messages"
   - Would require additional UI and database schema changes
4. **No Read Receipts**
   - No indication if/when others have read messages
   - Common in modern messaging apps but adds complexity
5. **No Notification System**

- Users must have application open to see new messages
- No desktop notifications or sound alerts
- Terminal.Gui limitations make this challenging

6. **Limited Search Functionality**
   - Search is client-side and case-sensitive
   - No advanced filters (by user, by date, etc.)
   - Searches only currently loaded messages
7. **Single Device Session**
   - Cannot be logged in on multiple devices simultaneously
   - No session management system
   - Would require token-based authentication
8. **No Message History Pagination**
   - Loads all messages on chatroom open
   - Could be slow for chatrooms with 1000+ messages
   - Should implement lazy loading/pagination

7.3.2 Testing Limitations

1. **Manual Testing Only**
   - No automated test suite
   - All tests conducted manually by developer and classmates
   - Regression testing requires re-running all tests manually
2. **Limited Scale Testing**
   - Maximum 5 concurrent users tested
   - Unknown behavior with 100+ users in single chatroom
   - Firestore costs prohibit large-scale testing
3. **Single Network Environment**
   - All testing on university/home network
   - Behavior on slow connections (¡1 Mbps) not tested
   - Firewall/proxy scenarios not validated

8 Future Enhancements

8.1 Planned Features

8.1.1 Short-term Enhancements (1-3 months)

1. **Password Hashing**
   - Implement Argon2 or bcrypt for secure password storage
   - Add salt generation and verification
   - Migration script for existing plaintext passwords
   - **Priority**: Critical (security issue)
2. **Message Editing**
   - Allow users to edit their own messages within 5 minutes
   - Show "edited" indicator on modified messages
   - Store edit history in Firestore
   - **Priority**: High (frequently requested feature)
3. **Message Deletion**
   - Delete individual messages (not just clear all)
   - "Delete for me" vs "Delete for everyone" options
   - Admin-only deletion in group chats
   - **Priority**: High
4. **Typing Indicators**
   - Show "User is typing..." when someone composes message
   - Update in real-time across all participants
   - Auto-remove after 3 seconds of inactivity
   - **Priority**: Medium (nice-to-have feature)
5. **Read Receipts**
   - Track when each user has read messages
   - Display checkmarks or timestamps
   - Privacy setting to disable read receipts
   - **Priority**: Medium

8.1.2 Medium-term Enhancements (3-6 months)

1. **File Sharing**
   - Support image uploads using Firebase Storage
   - File attachment support (PDFs, documents)
   - Preview images inline in chat
   - File size limits and virus scanning
   - **Priority**: High (major feature gap)
2. **Voice Messages**

- Record and send audio clips
- Playback within terminal (if supported)
- Waveform visualization using ASCII art
- **Priority**: Low (complex implementation)

3. **Message Reactions**
   - Add emoji reactions to messages
   - Display reaction counts
   - Multiple reactions per message
   - **Priority**: Medium
4. **Advanced Search**
   - Filter by sender, date range, keywords
   - Full-text search using Firestore queries
   - Search across all chatrooms
   - Save search queries
   - **Priority**: Medium
5. **Notification System**
   - Desktop notifications for new messages
   - Sound alerts (optional)
   - Notification preferences per chatroom
   - Mute chatrooms temporarily
   - **Priority**: High (important for usability)
6. **Multi-device Support**
   - Allow login from multiple devices
   - Session management with tokens
   - Sync read status across devices
   - Device management page
   - **Priority**: Medium

8.1.3 Long-term Enhancements (6+ months)

1. **End-to-End Encryption**
   - Implement Signal Protocol or similar
   - Encrypt messages client-side before sending to Firestore
   - Key exchange mechanism
   - Forward secrecy
   - **Priority**: High (privacy feature)
2. **Video Calls**
   - Integrate WebRTC or similar for voice/video
   - Screen sharing capabilities
   - Call history and recordings

- **Priority**: Low (major architectural change)

3. **Chatbots and Integrations**
   - OpenAI API integration for AI assistant
   - Webhook support for external services
   - Bot framework for custom automations
   - Already have OpenAI package referenced in project
   - **Priority**: Medium
4. **Rich Text Formatting**
   - Markdown support (bold, italic, code blocks)
   - Syntax highlighting for code snippets
   - Link previews
   - **Priority**: Low
5. **Message Threading**
   - Reply to specific messages
   - Thread view showing conversation branches
   - Thread notifications
   - **Priority**: Medium

8.2 Performance Improvements

8.2.1 Database Optimization

1. **Message Pagination**
   - Load messages in batches of 50
   - "Load more" button for older messages
   - Cache recent messages locally using LiteDB
   - Reduces initial load time for large chatrooms
2. **Firestore Index Optimization**
   - Create composite indexes for common queries
   - Index on (chatroom_id, timestamp) for message retrieval
   - Index on (participants, timestamp) for chatroom listing
3. **Denormalization**
   - Store last message directly in chatroom document
   - Cache user names to reduce profile lookups
   - Trade storage space for query speed

8.2.2 UI Performance

1. **Virtual Scrolling**
   - Render only visible messages in ListView
   - Improves performance with 1000+ messages
   - Reduce memory footprint
2. **Debouncing Search**
   - Add 300ms delay before executing search
   - Prevents excessive filtering on every keystroke
   - Improves responsiveness
3. **Message Batching**
   - Already partially implemented
   - Extend to batch profanity filtering
   - Process UI updates in requestAnimationFrame equivalent

8.2.3 Network Optimization

1. **Offline Support**
   - Cache messages locally with LiteDB
   - Queue outgoing messages when offline
   - Sync when connection restored
   - Firestore SDK supports offline mode
2. **Compression**
   - Compress large messages before sending
   - Especially beneficial for file sharing
   - Reduce bandwidth costs
3. **Connection Pooling**
   - Reuse Firestore connections
   - Implement connection health checks
   - Auto-reconnect on failure

8.2.4 Code Quality Improvements

1. **Automated Testing**
   - Unit tests for utility classes
   - Integration tests for Firestore operations
   - UI automation tests for Terminal.Gui
   - Continuous integration pipeline
2. **Error Handling**
   - Centralized error handling mechanism
   - Retry logic for transient failures

- Better error messages with actionable steps

3. **Logging**
   - Replace Console.WriteLine with proper logging framework
   - Log levels (DEBUG, INFO, WARN, ERROR)
   - Structured logging for easier debugging
4. **Documentation**
   - Generate API documentation from XML comments
   - Create developer onboarding guide
   - Architecture decision records (ADRs)

8.3 Architecture Improvements

1. **Dependency Injection**
   - Replace singletons with DI container
   - Improve testability
   - Make components more modular
2. **MVVM Pattern**
   - Separate UI logic from business logic
   - ViewModels for each window
   - Data binding where possible
3. **Repository Pattern**
   - Abstract Firestore operations behind interfaces
   - Enable easy database swapping
   - Simplify unit testing with mock repositories

9 Conclusion

9.1 Reflection

Developing Banter has been an enriching journey that combined cloud computing, real-time systems, and traditional console UI development. The project successfully demonstrates that modern cloud services can be seamlessly integrated with classic terminal interfaces, creating a unique user experience that bridges the gap between old and new paradigms.

9.1.1 What Went Well

1. **Real-time Synchronization**: The Firestore listener implementation exceeded expectations, providing sub-second message delivery across multiple clients with minimal code complexity.

2. **Modular Architecture**: The separation of concerns (Utilities, Windows, Models) made the codebase maintainable and allowed for parallel development of features.

3. **Collaborative Testing**: Working with classmates to test multi-user scenarios proved invaluable and revealed issues that would have been impossible to detect alone.

4. **Profanity Filtering**: The robust filtering algorithm successfully handles various obfuscation techniques, demonstrating the importance of thorough input sanitization.

5. **Event-Driven Design**: Using C# events for state management created a clean, reactive architecture that simplified complex UI updates.

9.1.2 Challenges Overcome

The most significant technical challenge was coordinating asynchronous Firestore operations with Terminal.Gui's synchronous, single-threaded UI model. This required deep understanding of both frameworks and careful use of `Application.MainLoop.Invoke()` to marshal updates back to the main thread.

The profanity filter presented an interesting algorithmic challenge, requiring a two-pass approach: normalize the text to detect obfuscated words, then map back to original positions for censoring. This solution balances performance with effectiveness.

Testing real-time functionality with multiple concurrent users highlighted the importance of designing for distributed systems from the start. Race conditions and synchronization issues only appear at scale, making collaborative testing essential.

9.1.3 Areas for Improvement

Looking back, several aspects could have been handled better:

1. **Password Security**: Implementing proper hashing from the beginning would have been more secure and easier than retrofitting later.

2. **Automated Testing**: Writing tests alongside development would have caught bugs earlier and made refactoring safer.

3. **Code Documentation**: More comprehensive XML comments would improve maintainability, especially for complex methods like the profanity filter.

4. **Error Recovery**: Better handling of network failures and automatic reconnection would improve user experience.

9.2 Takeaways

9.2.1 Technical Skills Developed

1. **Cloud Database Integration**: Gained deep understanding of Firestore's document model, queries, and real-time listeners. Learned to work within NoSQL constraints and optimize queries for performance.

2. **Asynchronous Programming**: Mastered async/await patterns in C#, particularly in UI contexts where thread marshaling is critical.

3. **File Handling in C#**: Implemented both external file reading (`File.ReadAllLines()`) and embedded resource access (`GetManifestResourceStream()`), understanding when each approach is appropriate.

4. **Error Handling**: Developed robust error handling strategies for file operations, database access, and user input validation.

5. **Terminal UI Development**: Learned Terminal.Gui framework, including responsive layouts, event handling, and thread-safe UI updates.

6. **Event-Driven Architecture**: Implemented observer pattern using C# events, creating loosely coupled components that react to state changes.

7. **Text Processing Algorithms**: Designed and implemented sophisticated string manipulation algorithms for profanity filtering.

9.2.2 Software Engineering Principles

1. **Separation of Concerns**: Organizing code into logical namespaces (Utilities, Windows, Models) improved maintainability and allowed features to be developed independently.

2. **Design Patterns**: Applied Singleton (window management), Factory (Firestore initialization), and Observer (session events) patterns appropriately.

3. **Defensive Programming**: Implemented null checks, input validation, and error handling to prevent crashes and data corruption.

4. **Code Reusability**: Created static helper classes (FirebaseHelper, WindowHelper) that eliminate duplication and provide consistent behavior.

5. **Documentation**: Used XML comments extensively, improving IntelliSense experience and code readability.

9.2.3 Distributed Systems Concepts

1. **Real-time Synchronization**: Learned how Firestore uses WebSockets to push updates to clients, and how to handle these updates efficiently.

2. **Eventual Consistency**: Understood that distributed systems may have brief inconsistencies, and designed UI to handle them gracefully.

3. **Conflict Resolution**: Dealt with concurrent operations (e.g., simultaneous message sends) and ensured proper ordering through timestamps.

4. **Session Management**: Implemented stateful sessions in a stateless cloud environment, learning about session lifecycle and cleanup.

9.2.4 Project Management Insights

1. **Iterative Development**: Building features incrementally allowed for early testing and course correction.
2. **Collaborative Testing**: Engaging classmates for multi-user testing provided valuable feedback and revealed edge cases.
3. **Version Control**: Using Git with meaningful commit messages made it easy to track changes and revert when necessary.
4. **Documentation**: Maintaining this comprehensive documentation clarified design decisions and will aid future development.

9.3 Final Thoughts

Banter represents a successful fusion of modern cloud technology with traditional terminal interfaces. The application proves that console UIs remain viable for certain use cases, particularly in developer-centric environments where efficiency and keyboard-driven workflows are valued.

The most valuable lesson learned is the importance of designing for real-world conditions from the start. Features like real-time synchronization, error handling, and multi-user scenarios must be architected early—retrofitting them later is significantly more difficult.

The project also highlighted the value of open-source tools and frameworks. Terminal.Gui, Firestore, and the .NET ecosystem provided robust foundations that accelerated development and reduced boilerplate code.

Moving forward, Banter serves as a solid foundation for further exploration of distributed systems, real-time communication, and terminal-based applications. The modular architecture and clean separation of concerns make it straightforward to add new features and improvements.

9.4 Acknowledgments

I would like to thank:

- My classmates who participated in multi-user testing sessions, providing invaluable feedback and helping identify synchronization issues
- The Terminal.Gui development team for creating an excellent console UI framework
- Google Cloud Platform for providing free Firestore access for educational projects
- The .NET community for comprehensive documentation and helpful Stack Overflow discussions

Appendix

A. Source Code

The complete source code for Banter is available on GitHub:

https://github.com/dreeyanzz/Banter

Repository includes:

- All C# source files
- Project configuration (.csproj, .sln)
- BanterLogo.txt ASCII art file
- README with setup instructions
- MIT License

To clone the repository:

```
git clone https://github.com/dreeyanzz/Banter.git
cd Banter
dotnet restore
dotnet run
```

B. Project Structure

```
Banter/
|-- Program.cs                  # Application entry point
|-- Banter.csproj               # Project configuration
|-- Banter.sln                  # Visual Studio solution
|-- BanterLogo.txt              # ASCII art logo
|-- firebase-service-account.json  # Firebase credentials (embedded)
|
|-- Utilities/
|   |-- CustomColorScheme.cs   # UI color definitions
|   |-- FirebaseHelper.cs      # Firestore operations
|   |-- FirestoreManager.cs    # Database connection
|   |-- Interfaces.cs           # IViewable interface
|   |-- Models.cs               # User, Chatroom, Message models
|   |-- ProfanityChecker.cs    # Content filtering
|   |-- SessionHandler.cs      # User session management
|   |-- Validator.cs            # Input validation
|   +-- WindowHelper.cs         # Window management utilities
|
+-- Windows/
    |-- AbstractWindow.cs              # Base window class
    |-- LogInWindow.cs                 # Login interface
    |-- CreateAccountWindow.cs         # Account registration
    |-- Window1.cs                     # Chatroom list (left panel)
    |-- Window2.cs                     # Chat interface (center panel)
    |-- Window3.cs                     # Chat info (right panel)
    |-- CreateChatroomWindow.cs        # New chatroom dialog
    |-- ChangeChatroomNameWindow.cs    # Rename dialog
    +-- ViewPinnedMessagesWindow.cs    # Pinned messages view
```

C. Database Schema

Firestore Collections

Users Collection:

```
{
  "users": {
    "<user_id>": {
      "username": "string",
      "password": "string",  // Currently plaintext
      "name": "string",
      "email": "string",
      "chatrooms": ["chatroom_id1", "chatroom_id2"]
    }
  }
}
```

Chatrooms Collection:

```
{
  "chatrooms": {
    "<chatroom_id>": {
      "chatroom_name": "string",  // null for individual chats
      "type": "individual | group",
      "participants": ["user_id1", "user_id2"],
      "admins": ["user_id1"],
      "pinned_messages": ["message_id1", "message_id2"],
      "last_chat": "string",

      "messages": {
        "<message_id>": {
          "sender_id": "string",
          "text": "string",
          "timestamp": "timestamp"
        }
      }
    }
  }
}
```

D. References

The following resources were consulted during development:

1. **Firestore Documentation**
   - Google Cloud Firestore .NET SDK: https://cloud.google.com/dotnet/docs/reference/Google.Cloud.Firestore/latest
   - Real-time Listeners: https://firebase.google.com/docs/firestore/query-data/listen
   - Best Practices: https://firebase.google.com/docs/firestore/best-practices
2. **Terminal.Gui Framework**
   - GitHub Repository: https://github.com/gui-cs/Terminal.Gui
   - API Documentation: https://gui-cs.github.io/Terminal.Gui/
   - UI Catalog Examples: https://github.com/gui-cs/Terminal.Gui/tree/master/UICatalog
3. **C# Language and .NET**

- Async/Await Patterns: https://docs.microsoft.com/en-us/dotnet/csharp/async
- Event Handling: https://docs.microsoft.com/en-us/dotnet/standard/events/
- File I/O: https://docs.microsoft.com/en-us/dotnet/api/system.io.file
- Embedded Resources: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.getmanifestresourcestream

4. **Design Patterns**
   - Gang of Four: Design Patterns (1994)
   - C# Design Patterns: https://refactoring.guru/design-patterns/csharp
5. **Regular Expressions**
   - .NET Regex Class: https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex
   - Regex101 (testing tool): https://regex101.com/
6. **Community Resources**
   - Stack Overflow: Various threading and Firestore questions
   - GitHub Issues: Terminal.Gui issue tracker for UI-related problems
   - Reddit r/csharp: Community discussions on best practices

E. License

Banter is released under the MIT License: