Individual Final Project BLJ

Creating a Notes App using Rust

Name: Matteo Luciano Siro Cipriani

Submission Date: June 23, 2025

Company: Soreco AG

Contents

1   Versioning

Version	Date	Time	Updates	Author
v0.1.0	03.06.2025	13:06	Started Documentation	Matteo Cipriani
v0.1.1	03.06.2025	14:27	Started Introduction	Matteo Cipriani
v0.1.2	03.06.2025	15:08	Started Listing Sources	Matteo Cipriani
v0.2.0	03.06.2025	17:11	Revamped Components Cable to be of LaTeX-style	Matteo Cipriani
v0.3.0	04.06.2025	10:22	Added Decision Matrix	Matteo Cipriani
v0.3.1	04.06.2025	10:24	Added Dailies Section	Matteo Cipriani
v0.3.2	04.06.2025	11:02	Added Dailies 1 & 2	Matteo Cipriani
v0.4.0	04.06.2025	11:37	Described egui Testing	Matteo Cipriani
v0.4.1	10.06.2025	09:34	Added Daily 3	Matteo Cipriani
v0.5.0	10.06.2025	11:18	Rough Description of Security Features	Matteo Cipriani
v0.5.1	10.06.2025	13:17	Refactored Descriptions	Matteo Cipriani
v0.6.0	10.06.2025	13:35	Initialized Glossary	Matteo Cipriani
v0.6.1	10.06.2025	14:52	Entered First Entries into Glossary	Matteo Cipriani
v0.7.0	10.06.2025	15:22	Began Gantt for Planned Schedule	Matteo Cipriani
v0.7.1	10.06.2025	16:40	Changed Gantt from SVG to PNG	Matteo Cipriani
v0.7.2	11.06.2025	09:01	Added Daily 4	Matteo Cipriani
v0.7.3	11.06.2025	16:35	Added Daily 5	Matteo Cipriani
v0.8.0	16.06.2025	13:51	Redesigned Glossary	Matteo Cipriani
v0.8.1	16.06.2025	16:24	Added Daily 6	Matteo Cipriani
v0.8.2	17.06.2025	14:29	Extended Glossary with new words	Matteo Cipriani
v0.8.3	17.06.2025	15:31	Redesigned versioning table	Matteo Cipriani
v0.8.4	18.06.2025	08:12	Added Daily 7	Matteo Cipriani
v0.8.5	18.06.2025	17:46	Added Daily 8	Matteo Cipriani
v0.8.6	20.06.2025	17:27	Added Daily 9	Matteo Cipriani
v0.9.0	23.06.2025	15:18	Added extensive description of encryption process	Matteo Cipriani

2 Introduction

2.1 Task Definition

At the end of the year, each apprentice who worked in the ZLI is required to do a project of their own choosing. They have to plan, execute and document an appropriate project over the span of 4 weeks, while working Monday - Wednesday (or Wednesday - Friday, depending on their school days). With this project, the apprentices can demonstrate what they have learned from the coaches during the last year, as all competences required to fulfill the project have been topics during this past year, some have been used very frequently, while others have only been discussed during 1 week.

2.2 Project Description

I chose to create a Notes App using Rust. I initially wanted to make a To-Do App, but as I have already done a To-Do App using Dart & Flutter as my Sportferienprojekt, I chose to go with something different. I want to try to write this project purely in Rust, to see how much of the language I have learned during the last year, and I can definitely learn new things from this project too. Because Rust is quite famous for being a really safe programming language, I want to try and implement one or two ways to encrypt and store the data safely.

2.3 Known Risks

I know that creating an application purely in Rust might be difficult, especially because Rust isn't really made to design, but to work. To implement a GUI, you have to use crates, which are known to sometimes be even more difficult than the standard Rust syntax itself. And Rust itself has a pretty steep learning curve too. Managing lifetimes, references, and borrowing can be complex, especially with dynamically changing data like note content. On top of that, Rust's error system (e.g., `Result` and `Option`) is safe but verbose, requiring you to explicitly handle many cases.

3 Planning

3.1 Schedule
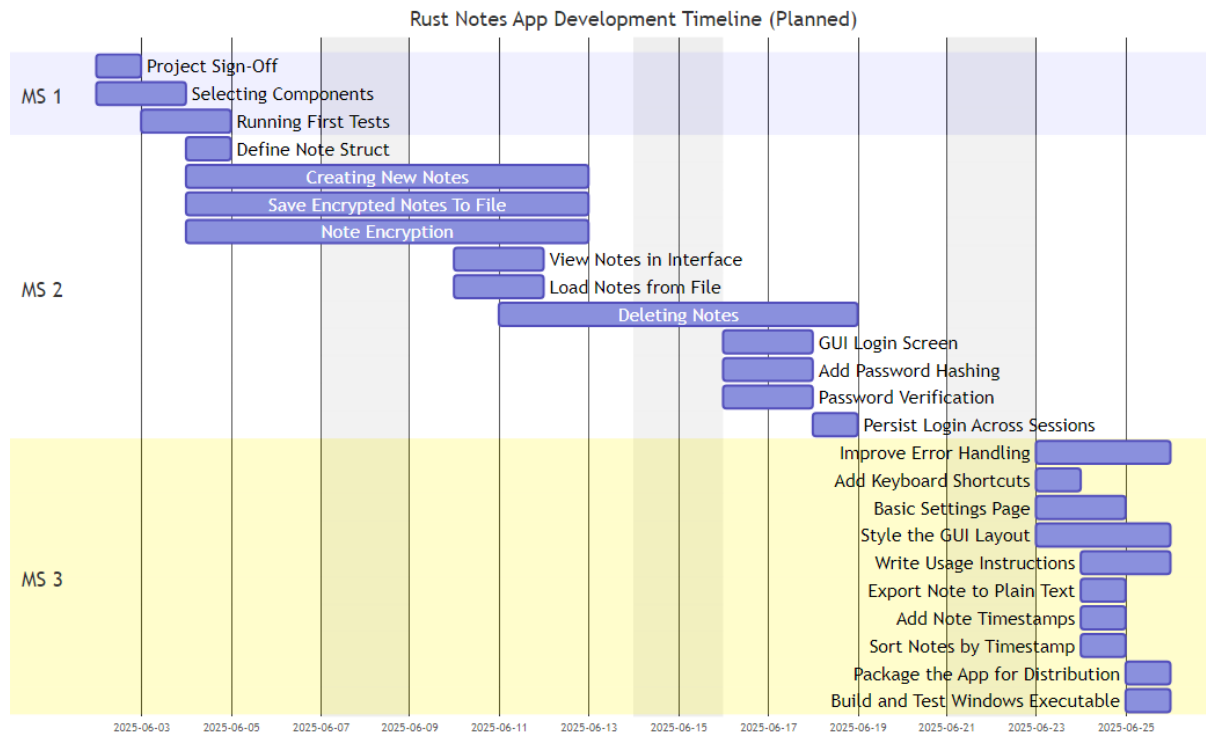
Figure 1: Schedule (Planned)

3.2 Decision Matrix

3.2.1 Basic Functionality Components (MS 2)

Feature	Recommended Crates	Implementation Suggestions
Password Verification	`argon2`, `bcrypt`, or `pbkdf2`	Use Argon2id for password hashing - it's modern and secure. Store the hash in a config file using `confy` or `config`. For verification, use the crate's verify function to compare entered password against stored hash.
GUI Framework	`egui`, `iced`, or `druid`	`egui` is lightweight and easy to use for beginners. `iced` offers a more Elm-like architecture. Both have good documentation and active communities.
Note Struct	`serde`, `chrono`, `uuid`	Use `chrono` for timestamps, `uuid` for unique IDs, and `serde` with the `derive` feature for serialization. Consider implementing `Display` and `Debug` traits.

Feature	Recommended Crates	Implementation Suggestions
Note Encryption	`aes-gcm`, `chacha20poly1305`, `orion`	ChaCha20-Poly1305 is modern and fast on all platforms. Use `ring` or `orion` for key derivation from password (PBKDF2 or Argon2). Store a random salt with each note.
File Storage	`serde_json`, `bincode`, `postcard`	`bincode` for efficient binary serialization or `serde_json` for human-readable storage. Use `directories-next` to find appropriate app data directory.
Loading Notes	Same as above + anyhow	Use `anyhow` or `thiserror` for error handling during file operations. Consider lazy loading for large note collections.
Note Viewing	GUI framework	Implement a split view with note list on left and content on right. Use virtual list if you expect many notes.
Deleting Notes	GUI framework + confirm_dialog	Use context menus from your GUI framework. Consider soft deletion (marking as deleted) before permanent removal.
Session Persistence	`keyring`, `directories-next`	Store an encrypted token in the system keyring or in a hidden file in the app directory. Use `directories-next` to find appropriate locations.

3.2.2 Finalization & Polish Components (MS 3)

Feature	Recommended Crates	Implementation Suggestions
Error Handling	`thiserror`, anyhow, `log`	Define custom error types with `thiserror`. Use `log` with `env_logger` or `fern` for logging. Show user-friendly messages in UI while logging details.
GUI Styling	GUI framework theming	Most Rust GUI frameworks support theming. Use system colors or implement dark/light mode toggle. Consider accessibility (contrast, font sizes).
Timestamps	`chrono`, `time`	Store UTC timestamps internally, convert to local time for display. Format with `chrono`'s formatting options.
Note Sorting	Standard library	Use Rust's built-in sorting with custom comparators. Consider allowing multiple sort options (recent, alphabetical).
Plain Text Export	`std::fs`, GUI file dialog	Use native file dialogs from your GUI framework. Consider supporting multiple formats (txt, md, html).
Settings Page	GUI framework, `confy`	Store settings with `confy` which handles serialization. Create a modal dialog or separate tab for settings.
Windows Executable	`cargo-wix`, `cargo-bundle`	Use GitHub Actions for CI/CD to automate builds. Test on Windows VM before release.

Feature	Recommended Crates	Implementation Suggestions
App Packaging	`cargo-wix`, `cargo-bundle`	Create an installer with `cargo-wix` for Windows. Include a nice icon and proper metadata.
Documentation	`mdbook`	Write user docs in Markdown, possibly generate with `mdbook`. Include screenshots and keyboard shortcuts.
Keyboard Shortcuts	GUI framework	Most GUI frameworks have built-in shortcut handling. Map to actions using a configuration struct.

4 Main Content

4.1 Procedure and steps

4.1.1 Testing with `egui`

After reading up on a bit of the documentation, I tried to copy a simple "tutorial" app that is just a input field for a name and a slider for the age. Once I was done copying all the code and successfully ran the program for the first time, I tried to figure out how I can access variables that I referenced when initiating the app's components. I also played around with function calls, and where it is best to place them.

4.1.2 Encrypting and saving

During my experiments with `egui` and other crates that I'd use later in the project, I initially went for an approach that encrypts the notes in an unusual way:

1. Key Derivation: A random 16-byte salt is generated. Using Argon2id, a secure 32-byte key is derived from the user's password and the salt.
2. Encryption: A random 12-byte nonce is created. The plaintext data is encrypted using AES-256-GCM with the derived key and nonce, producing ciphertext and an authentication tag.
3. Metadata Attachment The salt, Argon2 password hash (as UTF-8), nonce, and ciphertext are bundled together along with structured metadata (e.g., as a JSON `EncryptedData` object).
4. Obfuscation and Finalization
   - A fake 'SQLite format 3' header is prepended.
   - A Unix timestamp (8 bytes) and 48 bytes of random padding are appended.
   - A SHA-256 checksum of the entire content is added for integrity.

In the actual project, the encryption process looks a little more like this:

1. Hardware Fingerprinting   The system creates a stable hardware fingerprint using:

```rust
// Components used for hardware fingerprinting
let mut components = Vec::new();
components.push(format!("user:{}", username));         // Username
components.push(format!("home:{}", home_dir));         // Home directory
components.push(format!("os:{}", env::consts::OS));    // Operating system
components.push(format!("arch:{}", env::consts::ARCH)); // Architecture
components.push(format!("computer:{}", computer_name)); // Computer name
```

2. Password-Based Key Derivation (Argon2id)

```rust
// Production-grade Argon2 parameters
let memory_cost = 131072; // 128 MB memory usage
let iterations = 3;       // 3 iterations
let parallelism = 4;      // 4 parallel threads

// Key derivation takes ~5-10 seconds intentionally
let params = argon2::Params::new(memory_cost, iterations, parallelism, Some(32));
let argon2 = Argon2::new(argon2::Algorithm::Argon2id, argon2::Version::V0x13, params);
```

3. Hardware-Bound Salt Generation

```rust
fn generate_hardware_salt(&self) -> [u8; 32] {
    let hardware_hash = self.generate_stable_hardware_fingerprint();
    let hash_bytes = hardware_hash.to_le_bytes();
```

```
    // Create deterministic salt from hardware fingerprint
    for i in 0..32 {
        let factor = (i as u8).wrapping_mul(17);
        salt[i] = hash_bytes[i % 8] ^ factor ^ 0xAA;
    }
}
```

Encryption Implementation

1. Cipher: ChaCha20Poly1305 AEAD

- **Algorithm**: ChaCha20 stream cipher + Poly1305 MAC
- **Key Size**: 256 bits (32 bytes)
- **Nonce Size**: 96 bits (12 bytes)
- **Authentication**: Built-in message authentication

2. Encryption Process

```rust
pub fn encrypt(&self, data: &[u8]) -> Result<Vec<u8>> {
    let cipher = self.cipher.as_ref().ok_or("Cipher not initialized")?;

    // Generate random nonce for each encryption
    let nonce = ChaCha20Poly1305::generate_nonce(&mut OsRng);

    // Encrypt data with authentication
    let ciphertext = cipher.encrypt(&nonce, data)?;

    // Prepend nonce to ciphertext
    let mut result = Vec::new();
    result.extend_from_slice(&nonce);        // First 12 bytes: nonce
    result.extend_from_slice(&ciphertext); // Remaining: encrypted data + auth tag
    Ok(result)
}
```

3. Decryption Process

```rust
pub fn decrypt(&self, data: &[u8]) -> Result<Vec<u8>> {
    if data.len() < 12 {
        return Err(anyhow!("Invalid encrypted data"));
    }

    // Split nonce and ciphertext
    let (nonce_bytes, ciphertext) = data.split_at(12);
    let nonce = Nonce::from_slice(nonce_bytes);

    // Decrypt and verify authentication
    let plaintext = cipher.decrypt(nonce, ciphertext)?;
    Ok(plaintext)
}
```

Data Flow

Encrypted File → Read from Disk → ChaCha20Poly1305 Decryption → JSON Deserialization → Note Object
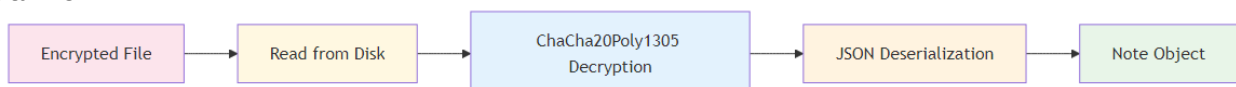
Figure 2: Authentication & Key Initialization Process

Figure 3: Note Storage Process

Security Features

1. Hardware Binding

```rust
// Security check on each login
let (current_hash, current_components) = self.generate_stable_hardware_fingerprint()?;

if metadata.hardware_fingerprint_hash != current_hash {
    // Detect hardware changes
    return Err(anyhow!("Hardware fingerprint changed"));
}
```
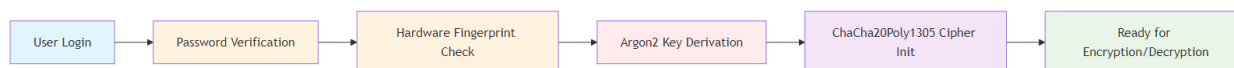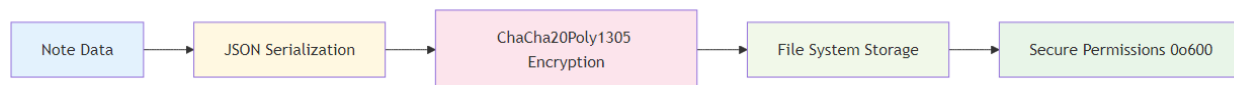
2. Security Metadata

```rust
struct SecurityMetadata {
    version: u32,                          // Encryption version
    created_timestamp: u64,                // Account creation time
    hardware_fingerprint_hash: u64,        // Hardware binding hash
    hardware_components: Vec<String>,      // Detailed hardware info
}
```

3. File Security

```rust
#[cfg(unix)]
fn secure_file_permissions(&self, file_path: &Path) -> Result<()> {
    let mut perms = fs::metadata(file_path)?.permissions();
    perms.set_mode(0o600); // Owner read/write only
    fs::set_permissions(file_path, perms)?;
}
```

4. User Isolation

```
~/.config/secure_notes/
├── users.json                # User database (hashed passwords)
└── users/
    ├── user1-uuid/
    │   ├── notes.enc          # Encrypted notes
    │   ├── auth.hash          # Password verification hash
    │   └── security.meta      # Security metadata
    └── user2-uuid/
        ├── notes.enc
        ├── auth.hash
        └── security.meta
```

Security Properties

Property	Implementation	Benefit
Confidentiality	ChaCha20 encryption	Data unreadable without key
Integrity	Poly1305 authentication	Detects tampering
Authentication	Argon2 password hashing	Prevents unauthorized access
Hardware Binding	Fingerprint-based salt	Prevents key extraction

Property	Implementation	Benefit
Forward Secrecy	Random nonces per encryption	Past data safe if key compromised
User Isolation	Per-user encryption keys	No cross-user data access

Key Security Parameters

```
// Argon2id Configuration (Production Grade)
memory_cost: 131072,    // 128 MB - Prevents memory-hard attacks
iterations: 3,          // 3 rounds - Balances security/performance
parallelism: 4,         // 4 threads - Utilizes modern CPUs
output_length: 32,      // 256-bit key - Industry standard

// ChaCha20Poly1305 Configuration
key_size: 32,           // 256-bit key
nonce_size: 12,         // 96-bit nonce (never reused)
auth_tag_size: 16,      // 128-bit authentication tag
```

Critical Security Checks

Hardware Change Detection

```rust
fn is_critical_hardware_change(&self, stored: &[String], current: &[String]) -> bool {
    // Only fail on critical component changes
    let stored_critical: Vec<_> = stored.iter()
        .filter(|c| c.starts_with("user:") || c.starts_with("os:") || c.starts_with("arch:"))
        .collect();

    let current_critical: Vec<_> = current.iter()
        .filter(|c| c.starts_with("user:") || c.starts_with("os:") || c.starts_with("arch:"))
        .collect();

    stored_critical != current_critical
}
```

This encryption system provides **military-grade security** while maintaining usability through automatic key management and hardware binding, ensuring that encrypted notes remain secure even if the application files are compromised.

4.2 Function Descriptions

4.2.1 `CryptoManager::initialize_for_user()`

- **Reference:** Code Snippet No. 4

- **Location:** `crypto.rs:44-154`

- **Description:** This is the **security cornerstone** of the entire application. It implements a sophisticated multi-layered security system:

    - **Hardware Binding:** Creates a unique fingerprint based on username, home directory, OS, architecture, and computer name. This prevents easy data portability between machines.
    - **Dual Authentication:** Stores both a password hash for verification AND derives an encryption key from the password.
    - **Backward Compatibility:** Handles metadata format upgrades gracefully.

– **Performance Monitoring:** Times the entire process and provides detailed logging.
– **Critical vs Non-Critical Changes:** Distinguishes between hardware changes that should block access (username/OS changes) vs those that shouldn't (computer name changes).

The function essentially creates a "vault" that's locked with both the user's password AND the specific hardware it was created on.

4.2.2 `NotesApp::start_authentication()`

- **Reference:** Code Snippet No. 5

- **Location:** `app.rs:108-174`

- **Description:** This function implements **asynchronous authentication** to prevent UI freezing during CPU-intensive operations. Key features:

  – **Thread Safety:** Uses message passing instead of shared memory to communicate between threads.
  – **Dual Flow Handling:** Manages both registration (create user → authenticate → initialize crypto) and login (authenticate → initialize crypto) in a single function.
  – **Error Propagation:** Comprehensive error handling with detailed error messages for debugging.
  – **Performance Tracking:** Records start time for performance monitoring.
  – **UI Responsiveness:** Keeps the UI thread free while expensive crypto operations run in background.

The registration flow is particularly complex as it requires three sequential operations, each of which can fail independently.

4.2.3 `CryptoManager::encrypt() & decrypt()`

- **Reference:** Code Snippet No. 6

- **Location:** `crypto.rs:280-306`

- **Description:** These functions provide **authenticated encryption** using ChaCha20Poly1305, which is considered state-of-the-art:

  – **ChaCha20Poly1305:** Combines ChaCha20 stream cipher with Poly1305 MAC for both confidentiality and authenticity.
  – **Nonce Management:** Each encryption uses a fresh random nonce, preventing replay attacks and ensuring semantic security.
  – **Data Format:** Encrypted data format is `[12-byte nonce][variable-length ciphertext+tag]`.
  – **Error Handling:** Validates input lengths and provides meaningful error messages.
  – **Performance:** ChaCha20 is faster than AES on systems without hardware AES acceleration.

The authenticated encryption prevents both eavesdropping AND tampering - if someone modifies the encrypted data, decryption will fail.

4.2.4 `StorageManager::save_user_notes() & load_user_notes()`

- **Reference:** Code Snippet No. 7

- **Location:** `storage.rs:18-67`

- **Description:** These functions implement **encrypted data persistence** with user isolation:

  – **Data Flow:** Notes HashMap → JSON → UTF-8 bytes → Encrypted bytes → File (and reverse for loading).
  – **User Isolation:** Each user gets their own directory under `users/{user_id}/`.

- **File Permissions:** On Unix systems, sets 0o600 (owner read/write only) for additional security.
- **Graceful Handling:** Returns empty HashMap if no notes file exists (new user scenario).
- **Error Propagation:** Each step can fail independently with specific error messages.

The serialization chain ensures that all note data (including metadata like timestamps) is preserved across application restarts.

4.2.5 `UserManager::create_user()`

- **Reference:** Code Snippet No.8

- **Location:** `user.rs:65-107`

- **Description:** This function implements **comprehensive user registration** with extensive validation:

  - **Input Sanitization:** Checks for empty, too short, too long, and invalid characters in usernames.
  - **Security Constraints:** Enforces password length limits (6-128 characters) to prevent both weak passwords and potential DoS attacks.
  - **Uniqueness Enforcement:** Case-insensitive username checking prevents confusion.
  - **Character Restrictions:** Only allows alphanumeric characters, underscores, and hyphens in usernames.
  - **Atomic Operations:** Either the entire user creation succeeds, or it fails completely (no partial state).

The validation is particularly thorough - it prevents common security issues like SQL injection (though not applicable here), ensures usernames are filesystem-safe, and enforces reasonable security policies.

4.2.6 `UserManager::authenticate()`

- **Reference:** Code Snippet No. 9

- **Location:** `user.rs:108-118`

- **Description:** This function implements **secure authentication** with several security best practices:

  - **Generic Error Messages:** Returns "Invalid username or password" for both non-existent users and wrong passwords, preventing username enumeration attacks.
  - **Secure Password Verification:** Delegates to `User::verify_password()` which uses Argon2 for secure hash comparison.
  - **Timing Attack Resistance:** Argon2 verification takes consistent time regardless of password correctness.
  - **User Object Return:** Returns a complete User object on success, providing all necessary user data for the session.

The function is deliberately simple but secure - it doesn't leak information about whether a username exists or not.

4.2.7 `User::verify_password()`

- **Reference:** Code Snippet No. 10

- **Location:** `user.rs:35-43`

- **Description:** This function provides **cryptographically secure password verification**:

  - **Argon2 Verification:** Uses the Argon2 password hashing function, winner of the Password Hashing Competition.
  - **Salt Handling:** The stored hash includes the salt, so no separate salt management is needed.
  - **Timing Attack Resistance:** Argon2 takes consistent time regardless of password correctness.

– **Error Handling:** Distinguishes between parsing errors (corrupted data) and verification failures (wrong password).

Argon2 is specifically designed to be memory-hard and resistant to both GPU and ASIC attacks, making it the gold standard for password hashing.

4.2.8 `NotesApp::check_authentication_result()`

- **Reference:** Code Snippet No. 11
- **Location:** `app.rs:175-226`
- **Description:** This function **bridges asynchronous authentication with the UI thread**:

  – **Non-blocking Polling:** Uses `try_recv()` to check for results without blocking the UI.
  – **Complete State Transition:** On success, performs all necessary setup: loads notes, migrates legacy data, runs security audit, updates UI state.
  – **Performance Monitoring:** Tracks and reports authentication duration.
  – **Cleanup:** Properly cleans up authentication state and clears sensitive input fields.
  – **Error Handling:** Handles both explicit errors and channel disconnection scenarios.

This function is called every UI frame, making it the bridge between the background authentication thread and the main application state.

4.2.9 `Note::relative_time()`

- **Reference:** Code Snippet No. 12
- **Location:** `note.rs:50-85`
- **Description:** This function provides **human-friendly time display** with intelligent granularity:

  – **Timezone Handling:** Converts UTC storage time to Swiss local time for accurate relative calculations.
  – **Granular Time Ranges:** Different descriptions for seconds, minutes, hours, days, and weeks.
  – **Singular/Plural Handling:** Proper grammar for "1 minute ago" vs "5 minutes ago".
  – **Fallback to Absolute:** For very old notes (>4 weeks), shows actual date instead of "X weeks ago".
  – **User Experience:** Makes it easy to quickly understand when notes were last modified.

This kind of relative time display is crucial for user experience - it's much easier to understand "2 hours ago" than "2024-06-23 14:30:15".

4.2.10 `Note::new()`

- **Reference:** Code Snippet No. 13
- **Location:** `note.rs:17-26`
- **Description:** This function creates **new note instances** with proper initialization:

  – **Unique Identification:** Uses UUID v4 for globally unique note IDs, preventing conflicts even across different users.
  – **Timestamp Management:** Sets both created_at and modified_at to the same UTC timestamp initially.
  – **Empty Content:** Starts with empty content, allowing users to immediately begin typing.
  – **UTC Storage:** Stores timestamps in UTC to avoid timezone confusion, converting to local time only for display.

The UUID ensures that even if two users create notes simultaneously, there will be no ID conflicts.

4.2.11 `NotesApp::create_new_note()`

- **Reference:** Code Snippet No. 14

- **Location:** `app.rs:295-306`

- **Description:** This function handles **note creation with user experience considerations**:

  - **Input Sanitization:** Handles empty titles gracefully by providing a default "Untitled Note".
  - **Immediate Selection:** Automatically selects the newly created note, allowing immediate editing.
  - **Persistent Storage:** Immediately saves the new note to prevent data loss.
  - **State Management:** Updates both the notes collection and the UI selection state.

The function ensures that users can never create a note without a title, preventing UI confusion.

4.2.12 `NotesApp::auto_save_if_needed()`

- **Reference:** Code Snippet No. 15

- **Location:** `app.rs:318-323`

- **Description:** This function implements **intelligent auto-save functionality**:

  - **Debouncing:** Waits for a 2-second delay after the last change before saving, preventing excessive disk I/O.
  - **Performance Optimization:** Avoids saving on every keystroke, which would be inefficient with encryption.
  - **Data Safety:** Ensures that changes are persisted regularly without user intervention.
  - **Timer Reset:** Updates the last save time after each save to restart the delay period.

This is called every UI frame, but only actually saves when the delay threshold is met.

4.2.13 `CryptoManager::generate_stable_hardware_fingerprint()`

- **Reference:** Code Snippet No. 16

- **Location:** `crypto.rs:156-195`

- **Description:** This function creates a **hardware-bound security fingerprint**:

  - **Cross-Platform Compatibility:** Uses different environment variables for Windows vs Unix systems.
  - **Stability Prioritization:** Chooses components that rarely change (username, OS) over volatile ones (IP address, running processes).

- **Deterministic Hashing:** Sorts components to ensure consistent hash generation across runs.

  - **Graceful Degradation:** Provides fallback values if environment variables aren't available.
  - **Security Through Binding:** Makes encrypted data difficult to transfer between machines.

This fingerprint prevents someone from copying encrypted files to another machine and accessing them, even with the correct password.

4.2.14 `StorageManager::migrate_legacy_data_if_needed()`

- **Reference:** Code Snippet No. 17

- **Location:** `storage.rs:85-108`

- **Description:** This function handles **backward compatibility and data migration**:

  - **Legacy Detection:** Checks for old single-user data files from previous versions.
  - **Safe Migration:** Moves data to new user-specific structure without data loss.

      – **Backup Creation:** Renames old files instead of deleting them, providing a safety net.
      – **Conditional Operation:** Only performs migration if legacy data actually exists.
      – **User Feedback:** Provides clear logging about what migration actions were taken.

This ensures that users upgrading from single-user to multi-user versions don't lose their existing notes.

4.2.15 `NotesApp::render_auth_dialog()`

- **Reference:** Code Snippet No. 18

- **Location:** `auth.rs:15-130`

- **Description:** This function renders the **complete authentication interface** with sophisticated UX features:

      – **Dual Mode Interface:** Seamlessly switches between login and registration modes with conditional UI elements.
      – **Real-time Validation:** Shows validation errors as users type, preventing submission of invalid data.
      – **Progress Feedback:** During authentication, shows spinner, elapsed time, and escalating warnings for long operations.
      – **Keyboard Navigation:** Supports Enter key submission and proper tab order.
      – **Responsive Layout:** Calculates text widths and centers elements properly across different screen sizes.
      – **User Feedback:** Shows current user count and time for context.
      – **Error Handling:** Displays authentication errors with appropriate color coding.

The function handles the complex state transitions between idle, validating, authenticating, and error states while maintaining a clean user experience.

4.3 Test plan

Tests etc

5   Dailies

5.1   Day 1: 02.06.2025

On the first day, I first ran my idea by Reto really quickly, just to confirm wether I could do a project like this. After all, I didn't want to plan my project for the next few hours just for it to get turned down by him. After his confirmation, I began planning my project, utilizing GitHub Projects, just like I've already done for my other project that I realized while in the ZLI. Shortly after the lunch break, I presented my idea in a more detailed manner - every milestone and a few distinct goals that I wanted to reach. After getting the project signed off by Reto, I was able to begin my research on what I would actually need / use for this app. Very quickly, I found out about `egui` and `eframe`, which are 2 incredibly well documented crates that make it manageable to create a GUI for your app. As for safety, I chose to go with `argon2` for the password verification, while I decided to try out `chacha20poly1305` combined with `ring` or `orion` for the note encryption itself. I actually already got to test a bit with egui, where I tried to copy a basic application with name and age, that shows you how `egui` works and what is to expect when working with it.

5.2   Day 2: 03.06.2025

Day 2 was mainly focussed on the documentation, as I knew from my last project, that it would get incredibly difficult to write a good documentation just during the last week, as you forgot a lot of things already. I didn't want to create a documentation with Word, as I had quite a few problems with it the last time I tried it, so I did some research into Markdown-Documentations enhanced with LaTeX and found out, that it is actually a viable alternative to create your documentation with. While the installation of all the things I needed (or I needed to update) took quite some time, I think that I'll get that time back by not having to wrestle with the layout on each page every time I try to insert a picture. In the afternoon, I first began by describing the project's guidelines, my project description and the risks I knew about before beginning my project. I then added the list of sources that I have already used, which there were more of than I first thought.

5.3   Day 3: 04.06.2025

On day 3, I started implementing on implementing my basic features. I started by first designing a lightweight design for my app, with a simple GUI layout, so that the app wouldn't be hard to use. While I had setup the GUI fairly quickly, the saving / encryption process wouldn't be that fast. It took me way longer than expected to combine all my security features with each other, so that'd it actually be encrypted the way I described it in my project setup. But - after some trial and error, lots of documentation read and some help by v0, I got it to work. It now stores all data in the user's configuration directory. For windows, this would be `%APPDATA%\secure_notes\`, where it creates 3 files:

- `auth.hash`, which stores the password hash for authentication
- ~~`safety.meta`~~ `security.meta`, which contains security metadata (hardware fingerprints, timestamps)

  Changed `safety.meta` to `security.meta` for more accurate file names

- `notes.enc`, which is the encrypted note data itself

By implementing an encryption key which is bound to hardware characteristics like username, operating system, computer name, etc..., this creates a hardware fingerprint that makes the encrypted data only accessible on the same machine that it was encrypted on.

5.4   Day 4: 10.06.2025

The fourth day was also mainly focussed on documenting. I documented a few of my security features and how they work, and created the first entries into the glossary. I also created a Gantt diagram that shows how I initially planned my project to develop, to which I will add a Gantt of how it actually developed over the days once the project is finished. The app itself did not change much this day, as I had some catching up to do with my documentation. I only changed the UI a bit and also decreased the security of the app a little

bit - to a point where it still is theoretically safe for production, but the user doesn't have to wait 26 seconds every time they want to log in.

5.5   Day 5: 11.06.2025

The fifth day was overshadowed by a simple, yet embarrassing error that I made. When I was re-reading my code that I wrote over the past week, I noticed that I set the filename for security metadata to `safety.meta`, which I decided to change to `security.meta`, as that was a more fitting name for me. I also noticed that I was using `allocate_ui_at_rect()` to display my notes in the statusbar on the left, which I tried to refactor to `allocate_new_ui()` as `allocate_ui_at_rect()` is now deprecated and been replaced by `allocate_new_ui()`. When I ran the project for the first time again, the login just didn't work anymore. I got an error that my password was not matching, even though I entered the correct password. It took me an embarrassing amount of time to realize that it wasn't the new `allocate_new_ui()` approach that I took that was causing the error - which wouldn't have made any sense anyways since that only gets loaded once the login has already been completed - but the name change from `safety.meta` to `security.meta` that caused the error during login, as it searched for a file that didn't even exist. This is what caused the authentication error. I thusly had to delete all the files that have been created in `%APPDATA%\secure_notes\`, so that the program would think that it's a first time setup again, so I can create a new master password.

5.6   Day 6: 16.06.2025

Day 6 was shaped by some more documenting. Early in the morning, Reto reminded us that each of our repositories needed a README file, which I didn't yet create. So I had to work on that before I could do anything else. It took me a bit longer than expected because I wanted to create a clean and easy-to-read README file, that can be understood by anyone. I also added some parts where I'm not sure if I want to keep them in over longer periods of time, as they could grow to be untrue / not implemented.

5.7   Day 7: 17.06.2025

Day 7 was - unfortunately - quite an unproductive day. I tried my best at implementing user logins and getting the login to persist for *x* amount of time. However, I had some problems with storing all the user data and getting the right one to decrypt their password, and with showing them the right notes, that I had to call that mission off quite quickly. I then looked back at my documentation, and added some more words to my glossary. I then also went over my GitHub Roadmap again, for which I marked off the issues that I have now completed.

5.8   Day 8: 18.06.2025

On the eighth day I was working from home. I was able to implement User Login / Registration after dealing with it for some time. Then I had to fix a problem where *all* notes would be shown to every user once they've logged in. I fixed it by modifying how notes are stored. Now every user has their own folder with the 3 files in it. This means that the size of the storage goes up slightly, but it was the only way that came to mind fairly quickly. Deleting a note is now also stored in a context menu that pops up on right click, and not just always available as a button.

5.9   Day 9: 20.06.2025

Day 9 was an additional day that I got to work on my project on, as school got cancelled. I redesigned a few things in the app, although you can't really see that much of it because it is mainly just minor adjustments in the layout. I want to add a few different color schemes at some point, but I don't know how hard that might be to implement. I also split my code even further, as my `main.rs` file got pretty long. Now I use 9 files instead of the old 3. Weirdly enough, the decision on *how* to split the files itself was more complicated than the splitting of the files itself.

6 Appendix

6.1 Sources

- docs.rs
    - **–** Basic Documentation for all of the crates used
    - **–** Further linking to official websites / GitHub repositories with official examples / code snippets
    - **–** Structs, Enums, Functions, Models & Type Aliases for each crate (if available)
- GitHub
    - **–** Extensive Documentation about crates
    - **–** Function snippets
    - **–** Implementation Methods
- Rust's official website
    - **–** Basic Questions about Rust's functionality
    - **–** Further linking to community boards
- THE Rust Book
    - **–** Basic Introduction to Rust
    - **–** Easy explanations for some more complicated topics of Rust
- Rust By Example
    - **–** Examples for some crucial features
        - ∗ Examples are editable & executable
    - **–** Good "playground"
- The Cargo Book
    - **–** Guide through Rust's package manager
    - **–** Easy point to access Features, Commands and general infos about cargo

6.2 Glossary

6.2.1 Checksum

A small-sized hash or value used to verify the integrity of data. It ensures that data has not been tampered with or corrupted during storage or transmission.

6.2.2 CI/CD

Stands for Continuous Integration and Continuous Deployment/Delivery. In the context of a Rust secure notes app, CI/CD automates testing, building, and deploying updates to ensure code reliability and fast delivery.

6.2.3 Ciphertext

The encrypted form of data that cannot be read without decryption. In the app, notes are converted to ciphertext using an encryption key before being stored on disk.

6.2.4 Hash

A deterministic output of a hash function, producing a fixed-size value from arbitrary input. Hashes are used for verifying integrity, storing passwords securely, and comparing data without revealing the original input.

6.2.5 Key Derivation

A cryptographic process that generates a strong encryption key from a password or passphrase. Typically used with algorithms like PBKDF2, Argon2, or scrypt to protect against brute-force attacks.

6.2.6 Lightweight

Describes a program or library with minimal resource usage (e.g., memory, CPU). A lightweight secure notes app in Rust would be fast, efficient, and suitable for low-power or embedded environments.

6.2.7 Nonce

A "number used once" in cryptography to ensure that encryption results are unique each time. Used in encryption schemes like AES-GCM to prevent replay attacks and ensure data security.

6.2.8 Password Hashing

The process of converting a password into a fixed-size string (hash) using a cryptographic hash function. In a secure notes app, this is used to securely store and verify user passwords without keeping them in plain text.

6.2.9 Salt

A random value added to passwords before hashing to ensure unique hashes for identical passwords. This prevents precomputed hash attacks (e.g., rainbow tables).

6.2.10 Serialization

The process of converting data structures (e.g., Rust structs) into a format that can be stored or transmitted, such as JSON, TOML, or binary. Used in the app to save and load notes securely.

6.2.11 System Keyring

A secure storage mechanism provided by the operating system for storing secrets such as passwords or keys. The secure notes app can optionally use the system keyring to store encryption keys safely.

6.3   Code Snippets

```rust
fn main() -> Result<(), eframe::Error> {
    let options = eframe::NativeOptions {
        viewport: egui::ViewportBuilder::default()
            .with_min_inner_size([650.0, 465.0]) // Set minimum window size
            .with_title("Secure Notes") // Set app bar title
            .with_maximized(true) // Start in fullscreen
            .with_decorations(true), // Show the decorations (Minimize, Close, App Title)
        ..Default::default()
    };

    eframe::run_native(
        "Secure Notes",
        options,
        Box::new(|_cc| Ok(Box::new(NotesApp::new()))),
    )
}
```

Snippet 1: Initiating 'eframe'

```rust
egui::CentralPanel::default().show(ctx, |ui| {
    ui.heading("Encrypted Text Application");

    ui.horizontal(|ui| {
        let name_label = ui.label("Your name: ");
        let response = ui
            .text_edit_multiline(&mut self.name)
            .labelled_by(name_label.id);

        if response.changed() {
            if let Err(e) = self.save_encrypted_data() {
                eprintln!("Failed to save data: {}", e);
            }
        }
    });

    ui.add(egui::Slider::new(&mut self.age, 0..=120).text("years old"));
    ui.label(format!(
        "Hello {}, you are {} years old!", self.name, self.age
    ));
```

Snippet 2: First tests with variables

```rust
struct MyApp {
    name: String,
    age: u32,
}

impl MyApp {
    fn new() -> Self {
        let mut app = Self {
            name: String::new(),
            age: 18,
        };
    }

    fn increment(&mut self) {
        if self.age < 120 {
            self.age += 1;
        }
    }

    fn decrement(&mut self) {
        if self.age > 0 {
            self.age -= 1;
        }
    }
}

egui::CentralPanel::default().show(ctx, |ui| {
    ui.add(egui::Slider::new(&mut self.age, 0..=120).text("years old"));

    ui.horizontal(|ui| {
        if ui.button("Increment").clicked() {
            self.increment(); // Increment function call
        }

        if ui.button("Decrement").clicked() {
            self.decrement(); // Decrement function call
        }
    });
})
```

Snippet 3: First playtesting with functions

```rust
pub fn initialize_for_user(&mut self, user_id: &str, password: &str) -> Result<()> {
    println!("Starting crypto initialization for user: {}", user_id);
    let start_time = std::time::Instant::now();

    // Create user-specific config directory
    let mut user_config_path = self.config_path.clone();
    user_config_path.push("users");
    user_config_path.push(user_id);

    if !user_config_path.exists() {
        fs::create_dir_all(&user_config_path)?;
    }

    let key_file = user_config_path.join("auth.hash");
    let metadata_file = user_config_path.join("security.meta");

    let key = if key_file.exists() && metadata_file.exists() {
        println!("Loading existing user configuration...");

        // Load existing setup
        let stored_hash = fs::read_to_string(&key_file)?;
        let parsed_hash = PasswordHash::new(&stored_hash)
            .map_err(|e| anyhow!("Failed to parse password hash: {}", e))?;

        println!("Verifying password...");
        // Verify password (this should be fast with default Argon2)
        Argon2::default()
            .verify_password(password.as_bytes(), &parsed_hash)
            .map_err(|e| anyhow!("Password verification failed: {}", e))?;

        println!("Loading metadata...");
        // Load metadata
        let metadata_content = fs::read_to_string(&metadata_file)?;
        let mut metadata: SecurityMetadata = serde_json::from_str(&metadata_content)
            .map_err(|e| anyhow!("Failed to parse security metadata: {}", e))?;

        // Handle backward compatibility - if hardware_components is empty, regenerate it
        if metadata.hardware_components.is_empty() {
            println!("Upgrading old metadata format...");
            let (current_hash, current_components) =
                self.generate_stable_hardware_fingerprint()?;

            // Update the metadata with current components
            metadata.hardware_components = current_components;
            metadata.hardware_fingerprint_hash = current_hash;

            // Save updated metadata
            fs::write(&metadata_file, serde_json::to_string_pretty(&metadata)?)?;
            println!("Metadata upgraded successfully");
        } else {
            println!("Checking hardware fingerprint...");
            // Get current hardware components
            let (current_hash, current_components) =
                self.generate_stable_hardware_fingerprint()?;
```

```rust
        // Check if hardware fingerprint matches
        if metadata.hardware_fingerprint_hash != current_hash {
            // Try to identify what changed
            let mut changed_components = Vec::new();
            for (i, (stored, current)) in metadata
                .hardware_components
                .iter()
                .zip(current_components.iter())
                .enumerate()
            {
                if stored != current {
                    changed_components
                        .push(format!("Component {}: '{}' -> '{}'", i, stored, current));
                }
            }

            if !changed_components.is_empty() {
                println!("Hardware changes detected:");
                for change in &changed_components {
                    println!("  {}", change);
                }

                // For now, let's be more lenient and only fail if critical components change
                if self.is_critical_hardware_change(
                    &metadata.hardware_components,
                    &current_components,
                ) {
                    return Err(anyhow!(
                        "Critical hardware components changed: {}",
                        changed_components.join(", ")
                    ));
                } else {
                    println!("Non-critical hardware changes detected, allowing access...");
                    // Update the stored fingerprint
                    metadata.hardware_fingerprint_hash = current_hash;
                    metadata.hardware_components = current_components;

                    // Save updated metadata
                    fs::write(&metadata_file, serde_json::to_string_pretty(&metadata)?)?;
                }
            }
        } else {
            println!("Hardware fingerprint matches!");
        }
    }

    self.security_metadata = Some(metadata);

    println!("Deriving encryption key...");
    // Use standard security key derivation
    self.derive_secure_key(password)
} else {
    println!("First time setup for user...");
```

```rust
        let current_time = std::time::SystemTime::now()
            .duration_since(std::time::UNIX_EPOCH)
            .unwrap()
            .as_secs();

        let (hardware_hash, hardware_components) =
            self.generate_stable_hardware_fingerprint()?;

        println!("Initial hardware components: {:?}", hardware_components);
        println!("Initial hardware hash: {}", hardware_hash);

        let metadata = SecurityMetadata {
            version: 1,
            created_timestamp: current_time,
            hardware_fingerprint_hash: hardware_hash,
            hardware_components,
        };

        let key = self.derive_secure_key(password);

        println!("Storing password hash...");
        // Store password hash
        let verification_salt = SaltString::generate(&mut OsRng);
        let argon2 = Argon2::default();
        let password_hash = argon2
            .hash_password(password.as_bytes(), &verification_salt)
            .map_err(|e| anyhow!("Failed to hash password: {}", e))?;

        fs::write(&key_file, password_hash.to_string())?;
        fs::write(&metadata_file, serde_json::to_string_pretty(&metadata)?)?;

        self.secure_file_permissions(&key_file)?;
        self.secure_file_permissions(&metadata_file)?;

        self.security_metadata = Some(metadata);
        key
    };

    self.cipher = Some(ChaCha20Poly1305::new(&key));

    let elapsed = start_time.elapsed();
    println!(
        "Crypto initialization completed in {:.2}s",
        elapsed.as_secs_f64()
    );

    Ok(())
}
```

Snippet 4: Create User Locked Vault

```rust
pub fn start_authentication(
    &mut self,
    username: String,
    password: String,
    is_registration: bool,
) {
    if self.is_authenticating {
        return; // Already authenticating
    }

    self.is_authenticating = true;
    self.authentication_error = None;
    self.auth_start_time = Some(std::time::Instant::now());

    let (sender, receiver) = mpsc::channel();
    self.auth_receiver = Some(receiver);

    let user_manager = self.user_manager.clone();

    // Spawn background thread for authentication
    thread::spawn(move || {
        println!("Starting authentication in background thread...");

        if let Some(mut user_manager) = user_manager {
            let result = if is_registration {
                // Registration flow
                match user_manager.create_user(username.clone(), &password) {
                    Ok(_) => {
                        println!("User created successfully, now authenticating...");
                        // After successful registration, authenticate the user
                        match user_manager.authenticate(&username, &password) {
                            Ok(user) => {
                                let mut crypto_manager = CryptoManager::new();
                                match crypto_manager.initialize_for_user(&user.id, &password) {
                                    Ok(_) => {
                                        println!("Registration and authentication successful!");
                                        AuthResult::Success(crypto_manager, user)
                                    }
                                    Err(e) => {
                                        println!("Crypto initialization failed: {}", e);
                                        AuthResult::Error(format!(
                                            "Crypto initialization failed: {}",
                                            e
                                        ))
                                    }
                                }
                            }
                            Err(e) => {
                                println!("Authentication after registration failed: {}", e);
                                AuthResult::Error(format!(
                                    "Authentication after registration failed: {}",
                                    e
                                ))
                            }
```

```rust
                }
            }
            Err(e) => {
                println!("Registration failed: {}", e);
                AuthResult::Error(format!("Registration failed: {}", e))
            }
        }
    } else {
        // Login flow
        match user_manager.authenticate(&username, &password) {
            Ok(user) => {
                println!("User authenticated, initializing crypto...");
                let mut crypto_manager = CryptoManager::new();
                match crypto_manager.initialize_for_user(&user.id, &password) {
                    Ok(_) => {
                        println!("Login successful!");
                        AuthResult::Success(crypto_manager, user)
                    }
                    Err(e) => {
                        println!("Crypto initialization failed: {}", e);
                        AuthResult::Error(format!("Authentication failed: {}", e))
                    }
                }
            }
            Err(e) => {
                println!("Login failed: {}", e);
                AuthResult::Error(format!("Login failed: {}", e))
            }
        }
    };

    if let Err(_) = sender.send(result) {
        println!("Failed to send authentication result - UI may have closed");
    }
    } else {
        let _ = sender.send(AuthResult::Error("User manager not available".to_string()));
    }
});
}
```

Snippet 5: Authenticating the user

```rust
pub fn encrypt(&self, data: &[u8]) -> Result<Vec<u8>> {
    let cipher = self
        .cipher
        .as_ref()
        .ok_or_else(|| anyhow!("Cipher not initialized"))?;
    let nonce = ChaCha20Poly1305::generate_nonce(&mut OsRng);
    let ciphertext = cipher
        .encrypt(&nonce, data)
        .map_err(|e| anyhow!("Encryption failed: {}", e))?;

    let mut result = Vec::new();
    result.extend_from_slice(&nonce);
    result.extend_from_slice(&ciphertext);
    Ok(result)
}

pub fn decrypt(&self, data: &[u8]) -> Result<Vec<u8>> {
    let cipher = self
        .cipher
        .as_ref()
        .ok_or_else(|| anyhow!("Cipher not initialized"))?;

    if data.len() < 12 {
        return Err(anyhow!("Invalid encrypted data"));
    }

    let (nonce_bytes, ciphertext) = data.split_at(12);
    let nonce = Nonce::from_slice(nonce_bytes);
    let plaintext = cipher
        .decrypt(nonce, ciphertext)
        .map_err(|e| anyhow!("Decryption failed: {}", e))?;
    Ok(plaintext)
}
```

Snippet 6: Encrypting & Decrypting the Notes

```rust
pub fn save_user_notes(
    &self,
    user_id: &str,
    notes: &HashMap<String, Note>,
    crypto: &CryptoManager,
) -> Result<()> {
    let json_data = serde_json::to_string(notes)?;
    let encrypted_data = crypto.encrypt(json_data.as_bytes())?;

    // Create user-specific directory
    let user_dir = self.data_dir.join("users").join(user_id);
    fs::create_dir_all(&user_dir)?;

    let notes_file = user_dir.join("notes.enc");
    fs::write(&notes_file, encrypted_data)?;

    // Set secure file permissions on Unix systems
    #[cfg(unix)]
    {
        use std::os::unix::fs::PermissionsExt;
        let mut perms = fs::metadata(&notes_file)?.permissions();
        perms.set_mode(0o600); // Read/write for owner only
        fs::set_permissions(&notes_file, perms)?;
    }

    println!("Saved {} notes for user {}", notes.len(), user_id);
    Ok(())
}

pub fn load_user_notes(
    &self,
    user_id: &str,
    crypto: &CryptoManager,
) -> Result<HashMap<String, Note>> {
    let notes_file = self.data_dir.join("users").join(user_id).join("notes.enc");

    if !notes_file.exists() {
        println!(
            "No notes file found for user {}, starting with empty notes",
            user_id
        );
        return Ok(HashMap::new());
    }

    let encrypted_data = fs::read(&notes_file)?;
    let decrypted_data = crypto.decrypt(&encrypted_data)?;
    let json_str = String::from_utf8(decrypted_data)?;
    let notes: HashMap<String, Note> = serde_json::from_str(&json_str)?;

    println!("Loaded {} notes for user {}", notes.len(), user_id);
    Ok(notes)
}
```

Snippet 7: Save & Load User Notes

```rust
pub fn create_user(&mut self, username: String, password: &str) -> Result<()> {
    // Validate input
    if username.trim().is_empty() {
        return Err(anyhow!("Username cannot be empty"));
    }

    if username.len() < 3 {
        return Err(anyhow!("Username must be at least 3 characters long"));
    }

    if username.len() > 50 {
        return Err(anyhow!("Username must be less than 50 characters"));
    }

    // Check for invalid characters
    if !username
        .chars()
        .all(|c| c.is_alphanumeric() || c == '_' || c == '-')
    {
        return Err(anyhow!(
            "Username can only contain letters, numbers, underscores, and hyphens"
        ));
    }

    if password.len() < 6 {
        return Err(anyhow!("Password must be at least 6 characters long"));
    }

    if password.len() > 128 {
        return Err(anyhow!("Password must be less than 128 characters"));
    }

    // Check if user already exists (case-insensitive)
    let username_lower = username.to_lowercase();
    if self
        .users
        .keys()
        .any(|k| k.to_lowercase() == username_lower)
    {
        return Err(anyhow!("Username already exists"));
    }

    // Create the user
    let user = User::new(username.clone(), password)?;
    self.users.insert(username, user);
    self.save_users()?;

    println!("Successfully created user account");
    Ok(())
}
```

Snippet 8: Create New Users

```rust
pub fn authenticate(&self, username: &str, password: &str) -> Result<User> {
    let user = self
        .users
        .get(username)
        .ok_or_else(|| anyhow!("Invalid username or password"))?;

    if user.verify_password(password)? {
        println!("User {} authenticated successfully", username);
        Ok(user.clone())
    } else {
        Err(anyhow!("Invalid username or password"))
    }
}
```

Snippet 9: Authenticate username & password

```rust
pub fn verify_password(&self, password: &str) -> Result<bool> {
    let parsed_hash = PasswordHash::new(&self.password_hash)
        .map_err(|e| anyhow!("Failed to parse password hash: {}", e))?;

    match Argon2::default().verify_password(password.as_bytes(), &parsed_hash) {
        Ok(_) => Ok(true),
        Err(_) => Ok(false),
    }
}
```

Snippet 10: Cryptographically safe password verification

```rust
pub fn check_authentication_result(&mut self) {
    if let Some(receiver) = &self.auth_receiver {
        match receiver.try_recv() {
            Ok(AuthResult::Success(crypto_manager, user)) => {
                if let Some(start_time) = self.auth_start_time {
                    println!(
                        "Authentication completed in {:.2}s",
                        start_time.elapsed().as_secs_f64()
                    );
                }

                self.crypto_manager = Some(crypto_manager);
                self.current_user = Some(user);
                self.load_notes();
                self.migrate_legacy_data_if_needed();

                // Perform security audit
                if let Some(ref crypto) = self.crypto_manager {
                    if let Ok(warnings) = crypto.security_audit() {
                        self.security_warnings = warnings;
                    }
                }

                self.is_authenticated = true;
                self.show_auth_dialog = false;
                self.is_authenticating = false;
                self.auth_receiver = None;
                self.auth_start_time = None;

                // Clear input fields
                self.username_input.clear();
                self.password_input.clear();
                self.confirm_password_input.clear();
            }
            Ok(AuthResult::Error(error)) => {
                self.authentication_error = Some(error);
                self.is_authenticating = false;
                self.auth_receiver = None;
                self.auth_start_time = None;
            }
            Err(mpsc::TryRecvError::Empty) => {
                // Still waiting for result
            }
            Err(mpsc::TryRecvError::Disconnected) => {
                self.authentication_error = Some("Authentication process failed".to_string());
                self.is_authenticating = false;
                self.auth_receiver = None;
                self.auth_start_time = None;
            }
        }
    }
}
```

Snippet 11: Asynchronous authentication with the UI thread

```rust
pub fn relative_time(&self) -> String {
    let now = Utc::now().with_timezone(&Zurich);
    let modified = self.modified_at_local();
    let duration = now.signed_duration_since(modified);

    if duration.num_seconds() < 60 {
        "Just now".to_string()
    } else if duration.num_minutes() < 60 {
        let minutes = duration.num_minutes();
        if minutes == 1 {
            "1 minute ago".to_string()
        } else {
            format!("{} minutes ago", minutes)
        }
    } else if duration.num_hours() < 24 {
        let hours = duration.num_hours();
        if hours == 1 {
            "1 hour ago".to_string()
        } else {
            format!("{} hours ago", hours)
        }
    } else if duration.num_days() < 7 {
        let days = duration.num_days();
        if days == 1 {
            "Yesterday".to_string()
        } else {
            format!("{} days ago", days)
        }
    } else if duration.num_weeks() < 4 {
        let weeks = duration.num_weeks();
        if weeks == 1 {
            "1 week ago".to_string()
        } else {
            format!("{} weeks ago", weeks)
        }
    } else {
        // For older notes, show the actual date
        self.format_modified_time()
    }
}
```

Snippet 12: Display relative time

```rust
pub fn new(title: String) -> Self {
    let now = Utc::now();
    Self {
        id: Uuid::new_v4().to_string(),
        title,
        content: String::new(),
        created_at: now,
        modified_at: now,
    }
}
```

Snippet 13: Create new Notes

```rust
pub fn create_new_note(&mut self, title: String) {
    let final_title = if title.trim().is_empty() {
        "Untitled Note".to_string()
    } else {
        title
    };

    let note = Note::new(final_title);
    let note_id = note.id.clone();
    self.notes.insert(note_id.clone(), note);
    self.selected_note_id = Some(note_id);
    self.save_notes();
}
```

Snippet 14: Note creation with user experience considerations

```rust
pub fn auto_save_if_needed(&mut self) {
    if self.last_save_time.elapsed() >= self.auto_save_delay {
        self.save_notes();
        self.last_save_time = std::time::Instant::now();
    }
}
```

Snippet 15: Auto-save every 2 seconds

```rust
fn generate_stable_hardware_fingerprint(&self) -> Result<(u64, Vec<String>)> {
    println!("Generating stable hardware fingerprint...");

    // Use only the most stable components
    let mut components = Vec::new();

    // 1. Username (very stable)
    let username = env::var("USER")
        .or_else(|_| env::var("USERNAME"))
        .unwrap_or_else(|_| "unknown_user".to_string());
    components.push(format!("user:{}", username));

    // 2. Home directory (stable, but can change)
    if let Ok(home) = env::var("HOME").or_else(|_| env::var("USERPROFILE")) {
        components.push(format!("home:{}", home));
    } else {
        components.push("home:unknown".to_string());
    }

    // 3. OS and Architecture (very stable)
    components.push(format!("os:{}", env::consts::OS));
    components.push(format!("arch:{}", env::consts::ARCH));

    // 4. Computer name (can change but usually stable)
    let computer_name = env::var("COMPUTERNAME")
        .or_else(|_| env::var("HOSTNAME"))
        .or_else(|_| env::var("NAME"))
        .unwrap_or_else(|_| "unknown_computer".to_string());
    components.push(format!("computer:{}", computer_name));

    // Sort components for consistency
    components.sort();

    // Generate hash
    let combined = components.join("||");
    let mut hasher = DefaultHasher::new();
    combined.hash(&mut hasher);
    let hash = hasher.finish();

    println!("Hardware fingerprint components: {:?}", components);
    println!("Generated hash: {}", hash);

    Ok((hash, components))
}
```

Snippet 16: Generate Hardware Fingerprint

```rust
pub fn migrate_legacy_data_if_needed(&self, user_id: &str, crypto: &CryptoManager) -> Result<()>
    let legacy_file = self.data_dir.join("notes.enc");

    if legacy_file.exists() {
        println!("Found legacy notes file, migrating to user-specific storage...");

        // Load legacy notes
        let legacy_notes = self.load_notes(crypto)?;

        if !legacy_notes.is_empty() {
            // Save to user-specific location
            self.save_user_notes(user_id, &legacy_notes, crypto)?;

            // Backup the legacy file instead of deleting it
            let backup_file = self.data_dir.join("notes.enc.backup");
            fs::rename(&legacy_file, &backup_file)?;

            println!(
                "Migrated {} notes to user-specific storage",
                legacy_notes.len()
            );
            println!("Legacy file backed up as notes.enc.backup");
        }
    }

    Ok(())
}
```

Snippet 17: Migrate old files (if available)

```rust
pub fn render_auth_dialog(&mut self, ctx: &egui::Context) {
    egui::CentralPanel::default().show(ctx, |ui| {
        ui.vertical_centered(|ui| {
            ui.add_space(75.0);
            ui.heading("Secure Notes");
            ui.add_space(20.0);

            if self.is_authenticating {
                ui.label("Processing... Please wait");
                ui.spinner();

                // Show elapsed time
                if let Some(start_time) = self.auth_start_time {
                    let elapsed = start_time.elapsed().as_secs_f64();
                    ui.label(format!("Elapsed: {:.1}s", elapsed));

                    // Show warning if taking too long
                    if elapsed > 10.0 {
                        ui.colored_label(
                            egui::Color32::YELLOW,
                            "This is taking longer than expected...",
                        );
                    }

                    if elapsed > 30.0 {
                        ui.colored_label(
                            egui::Color32::RED,
                            "Something may be wrong. Try restarting the application.",
                        );
                        if ui.button("Cancel Authentication").clicked() {
                            self.is_authenticating = false;
                            self.auth_receiver = None;
                            self.auth_start_time = None;
                        }
                    }
                }

                // Request repaint to update timer
                ctx.request_repaint_after(std::time::Duration::from_millis(100));
            } else {
                let screen_width = ui.available_width();

                // Mode selection - calculate actual widget width and center it
                ui.horizontal(|ui| {
                    // Calculate actual text widths
                    let login_text_size = ui
                        .fonts(|f| {
                            f.layout_no_wrap(
                                "Login".to_string(),
                                egui::FontId::default(),
                                egui::Color32::WHITE,
                            )
                        })
                        .size();
```

```rust
        let register_text_size = ui
            .fonts(|f| {
                f.layout_no_wrap(
                    "Register".to_string(),
                    egui::FontId::default(),
                    egui::Color32::WHITE,
                )
            })
            .size();

        let spacing = ui.spacing().item_spacing.x;
        let total_width = login_text_size.x + register_text_size.x + spacing;

        let padding = (screen_width - total_width) / 2.0;

        ui.add_space(padding.max(0.0));
        ui.selectable_value(&mut self.auth_mode, AuthMode::Login, "Login");
        ui.selectable_value(&mut self.auth_mode, AuthMode::Register, "Register");
    });

    ui.add_space(20.0);

    // Username input
    ui.label("Username:");
    ui.add(
        egui::TextEdit::singleline(&mut self.username_input).desired_width(200.0),
    );

    ui.add_space(10.0);

    // Password input
    ui.label("Password:");
    let password_response = ui.add(
        egui::TextEdit::singleline(&mut self.password_input)
            .password(true)
            .desired_width(200.0),
    );

    // Confirm password for registration
    if self.auth_mode == AuthMode::Register {
        ui.add_space(10.0);
        ui.label("Confirm Password:");
        ui.add(
            egui::TextEdit::singleline(&mut self.confirm_password_input)
                .password(true)
                .desired_width(200.0),
        );
    }

    ui.add_space(20.0);

    // Submit button
    let button_text = match self.auth_mode {
```

```rust
            AuthMode::Login => "Login",
            AuthMode::Register => "Register",
        };

        let can_submit = !self.username_input.trim().is_empty()
            && !self.password_input.is_empty()
            && self.password_input.len() >= 6
            && (self.auth_mode == AuthMode::Login
                || self.password_input == self.confirm_password_input);

        if ui
            .add_enabled(can_submit, egui::Button::new(button_text))
            .clicked()
            || (password_response.lost_focus()
                && ui.input(|i| i.key_pressed(egui::Key::Enter))
                && can_submit)
        {
            if self.auth_mode == AuthMode::Register
                && self.password_input != self.confirm_password_input
            {
                self.authentication_error = Some("Passwords do not match".to_string());
            } else if self.password_input.len() < 6 {
                self.authentication_error =
                    Some("Password must be at least 6 characters long".to_string());
            } else {
                let username = self.username_input.clone();
                let password = self.password_input.clone();
                let is_registration = self.auth_mode == AuthMode::Register;
                self.start_authentication(username, password, is_registration);
            }
        }

        // Show validation errors
        if self.auth_mode == AuthMode::Register
            && !self.password_input.is_empty()
            && !self.confirm_password_input.is_empty()
            && self.password_input != self.confirm_password_input
        {
            ui.add_space(10.0);
            ui.colored_label(egui::Color32::YELLOW, "Passwords do not match");
        }

        if !self.password_input.is_empty() && self.password_input.len() < 6 {
            ui.add_space(10.0);
            ui.colored_label(
                egui::Color32::YELLOW,
                "Password must be at least 6 characters",
            );
        }

        // Show authentication error
        if let Some(error) = &self.authentication_error {
            ui.add_space(10.0);
            ui.colored_label(egui::Color32::RED, error);
```

```rust
            }

            // Show user count for context
            if let Some(ref user_manager) = self.user_manager {
                let screen_height = ui.available_height();
                ui.add_space(screen_height - 45.0);
                ui.separator();
                ui.small(format!(
                    "Registered users: {}",
                    user_manager.get_user_count()
                ));
                ui.small(format!("Current time: {}", self.get_current_time()));
            }
        }
    });
});
}
```

Snippet 18: Render Authentication GUI