Below is a comprehensive reference document that compiles all the responses related to your query about creating a secure file-sharing application. Since Grok does not yet support resuming conversations, this markdown-formatted file captures the entire discussion for easy reference.

**Secure File Sharing Project Reference**

This document compiles the guidance provided for creating a secure file-sharing application between two users, User A and User B, who are not on the same local network. It includes an explanation of the scenario, insights into how SCP works, necessary Rust dependencies, steps to establish a TCP connection, a detailed 14-week timeline for project implementation, and tips for success. Use this as a standalone guide for your project.

**Table of Contents**

1. [Scenario Description](#scenario-description)
2. [Understanding SCP](#understanding-scp)
3. [Rust Dependencies](#rust-dependencies)
4. [Establishing TCP Connection](#establishing-tcp-connection)
5. [Project Timeline](#project-timeline)
6. [Tips for Success](#tips-for-success)

**Scenario Description**

The goal is to create a program that allows User A to securely share a file with User B over the internet, ensuring privacy through encryption and a user-friendly interface. The program should:

* Allow User A to start a local server with a graphical user interface (GUI) for uploading the file.
* Generate a URL or connection information that User A can share with User B for downloading the file.
* Encrypt the file and its transmission using a password agreed upon by both users.
* Enable User B to start the program, input the connection information, and download and decrypt the file securely.

This setup draws inspiration from SCP (Secure Copy Protocol) but enhances it with a GUI and additional encryption features for accessibility and security.

**Understanding SCP**

The **Secure Copy Protocol (SCP)** is a network protocol for securely transferring files between hosts, built on top of the **Secure Shell (SSH)** protocol. Here’s how SCP works, which provides context for your project:

1. **Connection Establishment via SSH**:
   * SCP establishes a secure connection over **TCP port 22** using SSH.
   * For example, if User A runs scp file.txt userB@hostB:/path, the SCP client on User A’s machine initiates an SSH connection to User B’s machine (hostB).
   * Authentication is handled via passwords or public-key cryptography, ensuring only authorized users can connect.
2. **File Transfer Process**:
   * After authentication, the SCP client communicates with the SCP process on the remote server through the SSH daemon.
   * SCP operates in two modes:
     + **Source Mode**: The client sends the file from User A’s machine to User B’s machine.
     + **Sink Mode**: The client requests a file from the server to be written locally.
   * Example: In source mode, SCP sends a command like C0644 1024 file.txt (permissions, size, filename) followed by the file’s data, which the server writes to the specified path.
3. **Encryption and Security**:
   * SSH encrypts all data (commands and file contents) during transmission, ensuring confidentiality and integrity.

While SCP requires SSH access to the recipient’s machine, your project aims to simplify this by using a temporary server and a public URL, avoiding the need for direct SSH access.

**Rust Dependencies**

To build your secure file-sharing program in Rust, you’ll need libraries (crates) for server functionality, encryption, and client-side operations. Here’s a list of recommended dependencies:

* **rocket or actix-web**:
  + **Purpose**: Create an HTTPS server on User A’s machine with a web-based GUI for file uploads.
  + **Why**: Popular Rust web frameworks with support for HTTP/HTTPS and file handling.
* **rustls**:
  + **Purpose**: Enable TLS (Transport Layer Security) for secure data transmission.
  + **Why**: A Rust-native TLS library for encrypted connections (complements rocket or actix-web).
* **aes-gcm**:
  + **Purpose**: Encrypt the file using AES in Galois/Counter Mode.
  + **Why**: A fast, secure symmetric encryption algorithm for file privacy.
* **pbkdf2 or argon2**:
  + **Purpose**: Derive a strong encryption key from the user-agreed password.
  + **Why**: Strengthens weak passwords; argon2 is more modern and resistant to attacks.
* **reqwest**:
  + **Purpose**: Enable User B’s program to download the encrypted file as an HTTP client.
  + **Why**: A user-friendly Rust HTTP client with HTTPS support.
* **tokio**:
  + **Purpose**: Handle asynchronous operations for the server and client.
  + **Why**: Powers non-blocking I/O in Rust frameworks and reqwest.
* **serde and serde\_json** (optional):
  + **Purpose**: Serialize/deserialize data like metadata or configuration.
  + **Why**: Useful for structured data exchange if needed.

For exposing User A’s local server to the internet, use **ngrok** (a tunnel service). You can run it as a subprocess via tokio::process for reliability. The GUI will be web-based (HTML/CSS/JavaScript), leveraging the server framework, keeping the setup simple.

**Establishing TCP Connection**

User A’s temporary server must establish a secure TCP connection with User B’s client over the internet. Here’s how it works:

1. **Start an HTTPS Server**:
   * The server binds to a local TCP port (e.g., 8000) on User A’s machine and listens for connections.
   * TLS (via rustls or framework support) ensures encrypted communication, creating an HTTPS server.
2. **Expose the Server to the Internet**:
   * Use **ngrok** to create a public URL (e.g., https://abc123.ngrok.io) that routes traffic to the local port.
   * The program launches ngrok and retrieves the URL for User A to share with User B.
3. **Handle File Upload and Encryption**:
   * User A accesses the web GUI (e.g., localhost:8000), uploads the file, and enters the password.
   * The server:
     + Derives an encryption key using pbkdf2 or argon2.
     + Encrypts the file with aes-gcm and stores it temporarily.
     + Assigns a unique URL path (e.g., /download/xyz789).
4. **Provide Connection Information**:
   * The server generates a full URL (e.g., https://abc123.ngrok.io/download/xyz789) for User A to share with User B via a secure channel (e.g., email).
5. **Establish the TCP Connection**:
   * User B enters the URL in their program, which uses reqwest to connect.
   * Ngrok routes the request to User A’s server, which accepts the TCP connection on the bound port.
   * A **TLS handshake** establishes a secure HTTPS connection.
6. **Serve the File**:
   * User B’s program sends an HTTP GET request to the URL.
   * The server streams the encrypted file over the connection.
   * User B’s program downloads the file, prompts for the password, and decrypts it locally.

**Key Considerations:**

* **Security**: HTTPS encrypts transmission, and file-level encryption adds extra protection.
* **Stability**: TCP ensures reliable delivery; frameworks can handle large files with chunked transfers.
* **Accessibility**: Ngrok bypasses NAT/firewall issues, making the server reachable.

**Project Timeline**

Since you’re new to Rust and lack experience with React, this 14-week timeline helps you learn and build the project step-by-step. It assumes consistent weekly effort.

**Weeks 1-2: Learn Rust Basics**

* **Tasks**:
  + Study Rust syntax, ownership, borrowing, and I/O.
  + Write small programs (e.g., calculator, file reader).
* **Resources**:
  + [The Rust Book](https://doc.rust-lang.org/book/)
  + [Rust by Example](https://doc.rust-lang.org/rust-by-example/)
* **Goal**: Gain enough Rust knowledge to start a server.

**Weeks 3-4: Set Up a Basic HTTP Server in Rust**

* **Tasks**:
  + Use rocket or actix-web to create a server.
  + Serve a static HTML page (e.g., “Hello, World!”).
* **Resources**:
  + [Rocket Docs](https://rocket.rs/v0.5/guide/)
  + [Actix Docs](https://actix.rs/docs/)
* **Goal**: Run a server locally (e.g., http://localhost:8000).

**Weeks 5-6: Implement File Upload and Encryption**

* **Tasks**:
  + Add a POST endpoint for file uploads.
  + Encrypt files with aes-gcm using a password-derived key (pbkdf2 or argon2).
* **Resources**:
  + [aes-gcm](https://crates.io/crates/aes-gcm)
  + [pbkdf2](https://crates.io/crates/pbkdf2) or [argon2](https://crates.io/crates/argon2)
* **Goal**: Upload and encrypt a file via the server.

**Weeks 7-8: Implement File Download and Unique URLs**

* **Tasks**:
  + Generate unique IDs for files (e.g., /download/xyz789).
  + Serve encrypted files via a GET endpoint.
* **Goal**: Enable file sharing with unique links.

**Weeks 9-10: Integrate ngrok for Public Access**

* **Tasks**:
  + Install ngrok and start it from Rust (e.g., via tokio::process).
  + Test public access with the ngrok URL.
* **Resources**:
  + [ngrok Docs](https://ngrok.com/docs)
* **Goal**: Make the server accessible online.

**Weeks 11-12: Develop the Client-Side GUI**

* **Tasks**:
  + Build an HTML page with upload/download forms.
  + Use JavaScript and the [Web Crypto API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API) for client-side decryption.
* **Resources**:
  + [MDN Web Docs](https://developer.mozilla.org/)
* **Goal**: Create a functional web GUI.

**Weeks 13-14: Testing, Security, and Refinement**

* **Tasks**:
  + Test end-to-end file sharing.
  + Ensure encryption/decryption works securely.
  + Handle errors and refine the app.
* **Resources**:
  + [OWASP Secure Coding](https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/)
* **Goal**: Deliver a secure, polished file-sharing tool.

**Tips for Success**

* **Start Small**: Build incrementally (e.g., start with a simple server).
* **Use Git**: Track progress with version control (e.g., GitHub).
* **Get Help**: Seek advice on [Rust Subreddit](https://www.reddit.com/r/rust/) or [Stack Overflow](https://stackoverflow.com/).
* **Focus on Security**: Review your code with an experienced developer before deployment.

This document provides everything you need to develop your secure file-sharing application. Save it as a .md file (e.g., file-sharing-reference.md) for easy access and reference throughout your project!