Password manager with P2P Synchronisation

Technical Manual

Student Name: Dean Lynch

Student Number: 15359921

19/05/19

Table of Contents

1 Introduction

1.1 Overview

This is a unique application that combines password database management and synchronisation into a unified application. The application reads from a locally stored password database file, which can then be synchronised across a users devices using a peer-to-peer connection. This eliminates the need to expose the password database to third-party servers or cloud storage services, as well as saving the user the need to manually synchronise the database or manage their own self-hosted solution.

1.2 Glossary

- **Password manager:** A program used to store, manage and generate passwords.
- **Password database:** A file containing the users login information, but may also contain private information such as addresses, ID information etc.
- **Keepass:** KeePass is an open source password manager.
- **KDBX:** Keepass Database file format.

1.3 Initial Design Vs. Final Application

The application created achieves the minimum viable product outlined in the functional specification with the exception of two factor authentication.

The final iteration of the project before the deadline achieved the goal of creating a password manager that allows the user to create, edit, and manage password entries while also allowing the user to synchronise their password databases across devices using a secure peer-to-peer connection.

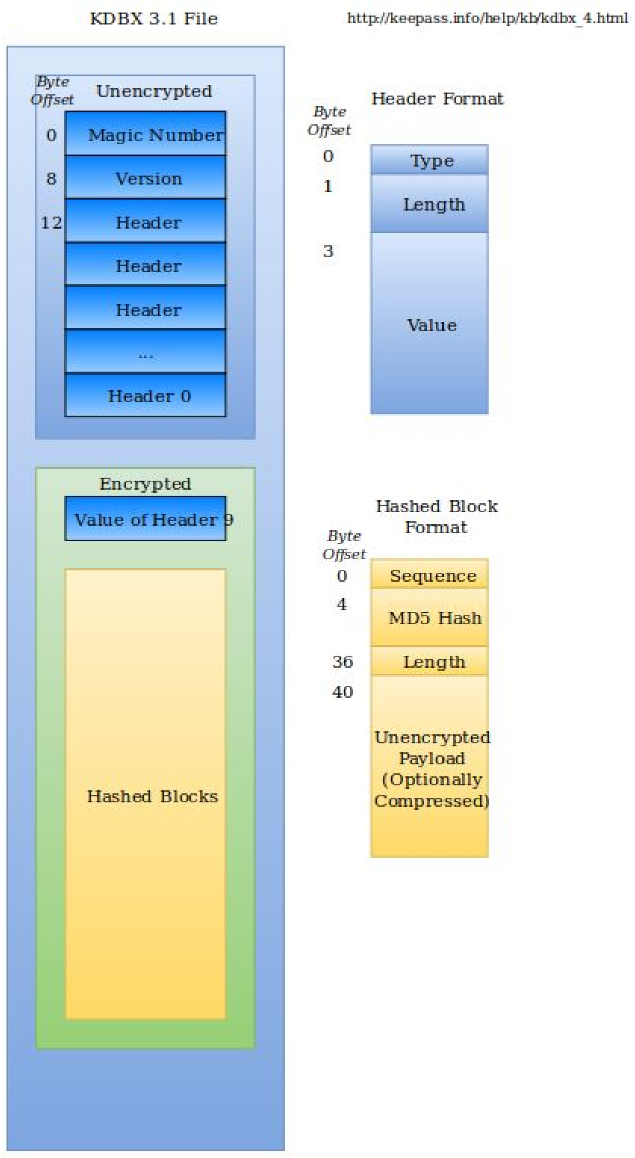
The two factor authentication feature, however was not included in this iteration of the application due to time constraints.

2 System Design & Architecture

2.1 Password Database Design

As writing a library to interact with KDBX databases would be a huge task and worthy of a full project in itself, I used an existing library called KeepassJava2. Further information about this library can be found at https://github.com/jorabin/KeePassJava2. This library is listed as an unofficial library on the official Keepass website, meaning that the creator of the Keepass and the KDBX file format has given this library their approval.

The diagram below shows the format of a KDBX 3.1 database file.



This project implements the Keepass KDBX 3.1 file format. KDBX 3.1 uses an AES key derivation function (AES-KDF) to encrypt the password database with a master password. Unfortunately this library does not yet support the KDBX 4 file format, so therefore I have not included support for this format in the final iteration of the project. The library also does not support changing the number of rounds for AES-KDF, so that is also not included in this application.

AES-KDF:
Advanced Encryption Standard (AES) is a symmetric encryption algorithm. The algorithm was developed by two Belgian cryptographer Joan Daemen and Vincent Rijmen. AES is designed to be efficient in both hardware and software, and supports a block length of 128 bits and key lengths of 128, 192, and 256 bits. The KDBX 3.1 file format uses a key length of 256. AES is trusted by the US government for securing sensitive but unclassified material, which speaks highly of its implementation. The same key is used to encrypt and decrypt the database, which in this case is the master password.

KDF stands for Key Derivation Function. Key derivation functions derive bytes suitable for cryptographic operations from passwords (such as a master password used for a password database) or other data sources using a pseudo-random function. This is to derive a key suitable for use as input to a encryption algorithm, which in this case is AES.

2.2 File Synchronisation Design

The file synchronisation feature of the application is implemented using the Java Secure Socket Extension (JSSE). This is a java implementation of the SSL and TLS protocols and provides data encryption, server and client authentication and data integrity,

SSL (Secure Socket Layer) enables a secure connection between the two devices, ensuring that the password database file cannot be compromised by another party with access to the network when the file is being transferred from one device to another.

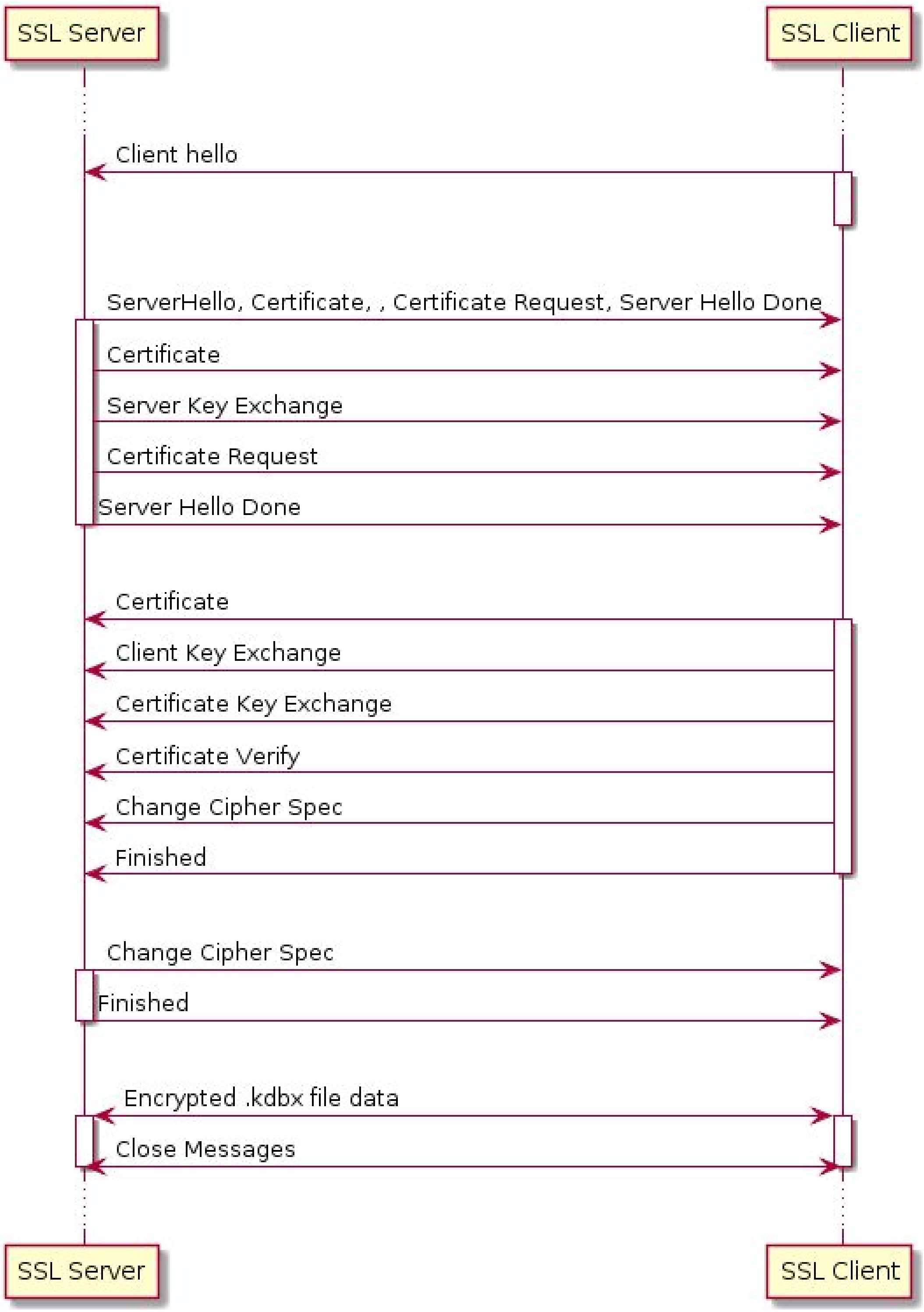
This ensures 3 main security principles:

1. Encryption: The data being transferred between the two devices (the password database file) is protected from other parties with access to the network.
2. Authentication: Ensure the device we connect to is the right one.
3. Data Integrity: Ensure the requested data is transferred without being corrupted in any way.

This application also adds extra checks to ensure that the file is not sent to an unwanted recipient. The file transfer process will not begin if the recipient's hostname does not match the one entered when starting the sending process. The recipient must know the exact name of the file and it is also not possible to request files with a format other than .kdbx.

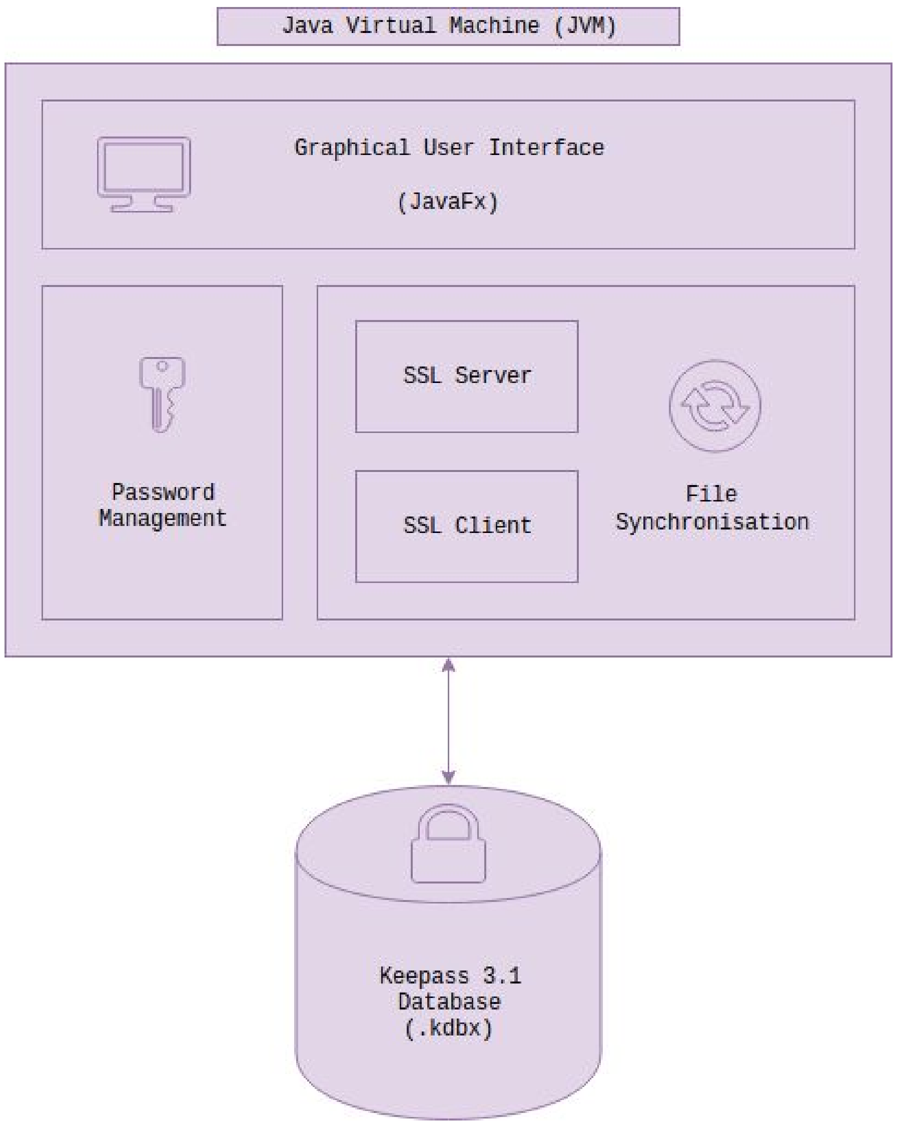
For added privacy and security, the error messages provided on the recipient devices side during the transfer process will not include any details as to why exactly the file transfer has failed. Detailed error messages are only displayed on the sending devices side.

SSL/TLS Handshake diagram for the application:



1. **Client hello:** The client sends the server information including the version of SSL and TLS that it supports.
2. **Server hello:** The server chooses the highest version of SSL and TLS that both the client and server support and sends this information to the client.
3. **Certificate:** The server sends the client a certificate chain.
4. **Certificate request:** The server sends the client a certificate request.
5. **Server key exchange:** The server sends the client a server key exchange message if the public key information from the Certificate is not sufficient for key exchange.
6. **Server hello done:** The server tells the client that it is finished with its initial negotiation messages.
7. **Certificate:** The client sends its certificate chain.
8. **Client key exchange:** The client generates information used to create a key to use for symmetric encryption.
9. **Certificate verify:** This message is sent by the client when the client presents a certificate. The client sends information that it digitally signs using a cryptographic hash function. When the server decrypts this information with the client's public key, the server is able to authenticate the client.
10. **Change cipher spec:** The client sends a message telling the server to change to encrypted mode.
11. **Finished:** The client tells the server that it is ready for secure data communication to begin.
12. **Change cipher spec:** The server sends a message telling the client to change to encrypted mode.
13. **Finished:** The server tells the client that it is ready for secure data communication to begin. This is the end of the SSL handshake.
14. **Encrypted .kdbx data:** The client and the server communicate using the symmetric encryption algorithm and the cryptographic hash function negotiated during the client hello and server hello, and using the secret key that the client sent to the server during the client key exchange. This is where the .kdbx file is transferred from the server to the client.
15. **Close Messages:** At the end of the connection, each side sends a close notify alert to inform the peer that the connection is closed.

2.3 System Architecture

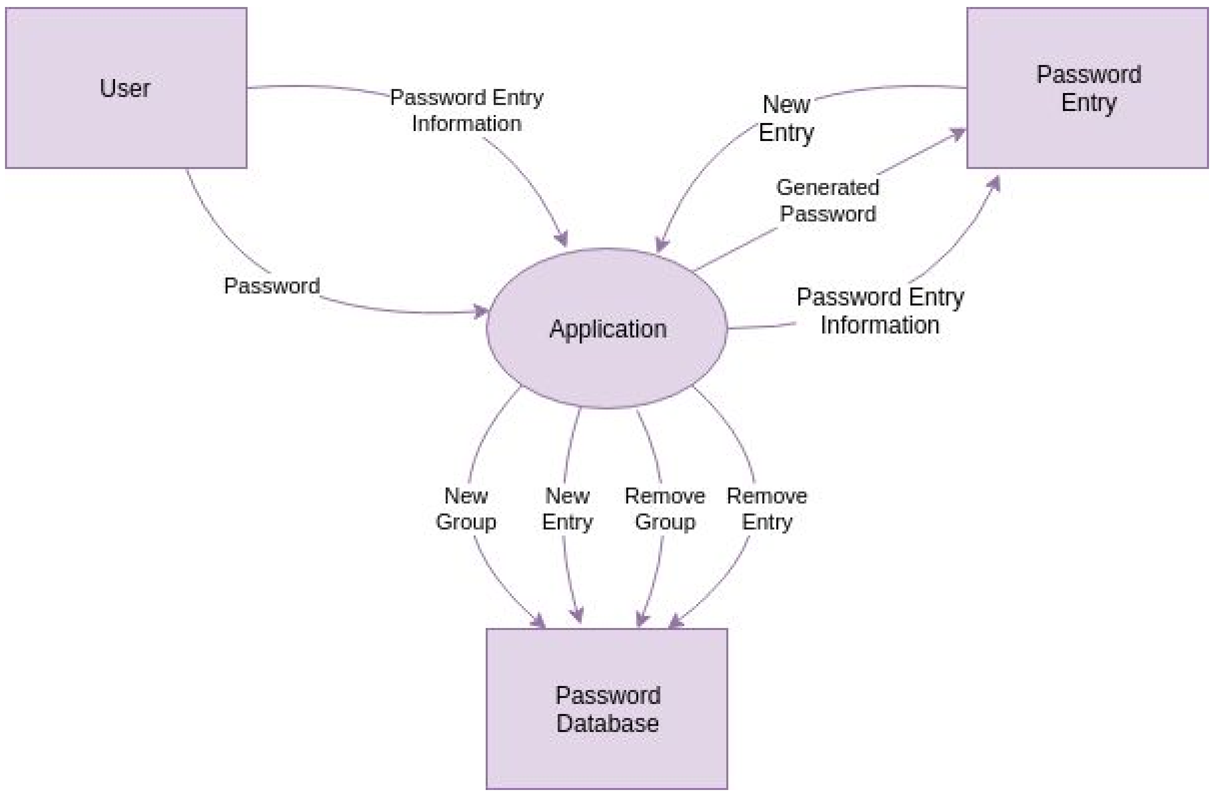


The above diagram shows a high level structure for the application. The application backend and frontend are tightly knit together as they are all written in Java.

The password management portion handles the encryption and decryption of the password database, as well as reading and writing to the database. The Synchronisation portion consists of two parts: a server and a client. The job of the server is to send .kdbx files directly to the recipient device using a secure SSL connection. The job of the client is to request and receive the file using the same secure SSL connection.

The GUI of the application is written entirely in Java using JavaFx. This mean that the GUI code can interact with the password management and synchronisation implementations without the need for an API layer between the GUI and the rest of the implementation.

2.4 Context Diagram



3 Design Decisions

3.1 Front end - JavaScript vs Java:

Despite having a good amount of experience with JavaScript and AngularJS based front-end implementations from working on my third year project and on my INTRA work placement, I decided to use a GUI implementation written entirely in Java.

I had a few ideas in mind when I decided this. Firstly I believe the lack of a REST API middle layer between the Java backend implementation and the frond end would reduce the number of potential entry points or security flaws. I also didn’t like the idea of exposing the backend or decrypted database file to a web browser connected to the internet. My only other real option would then be to use Electron instead of a web browser, but it is costly in performance and memory usage. Lastly, I wanted to challenge myself to learn something new, and I had never worked with a cross platform GUI implementation that wasn’t entirely written in JavaScript of some kind. So I chose to write it entirely in Java.

3.1 Swing vs JavaFx:

I has originally intended on writing the application using Swing, rather than JavaFx. I based this initial decision on the fact that major applications such as Intellij & PyCharm were written using Swing. Swing is also included in the default JDK, whereas JavaFx is not as of Java 8. After using Swing for a couple of days and feeling slightly unsatisfied with it, I decided to try out JavaFx. After using JavaFx for a couple of days I was much happier with it. I felt it was much easier to work with and provided a more modern looking GUI out of the box.

It’s a quite a steep curve to learn either Swing or JavaFx, but after using JavaFx over the course of the project I now feel quite comfortable using it. However I do slightly regret using JavaFx over Swing. Since it has been removed from the JDK, it’s not included in some distributions of Java 8. For this reason I decided to use Java 11, as it’s the latest LTS offering of Java, and the JavaFx integration is a bit more straightforward and better documented since the split. This has limited me to Java 11, which I understand can be a problem with compatibility on some devices (Mainly Ubuntu 18.04, which for some reason runs Java 10 labelled as Java 11, which caused me a lot of confusion).

3.3 Using KDBX Vs. my own password database format:

Creating my own database implementation with the time I had available and with the little experience I had would have been a massive task. It would have been too rushed and full of flaws. It would also be limited to only my application. For this reason I decided to use the tried and tested KDBX format. This is a format trusted by millions of users already. This decision was solidified by the fact that there is an existing Java library available called KeepassJava2 which is endorsed by Keepass which allowed me to easily integrate the database operations with the project.

3.4 Two factor authentication (2FA) exclusion:

As this was one of the only suggestions given to me in the project proposal stage, I would have really liked to include 2FA in the application, as it gives another layer of security to the password database and a greater piece of mind to the user. However, I simply did not have sufficient time to implement it, and rushing a security feature like this would be a very bad idea!

5. Problems & Resolutions

Merging the database when it is transferred from one database to another:

- **Problem:** When a database file was transferred from one device to another, I intended to merge both databases into one database file with the most up to date information. Unfortunately the KeepassJava2 library prevents me from copying entries from one database to another.
- **Resolution:** There was no real solution to this problem outside of forking the KeepassJava2 library and allowing this. In the end I had to settle with transfering the database to the recipient device without merging.

Password database format:

- **Problem:** Creating my own database format would be time consuming due to the design and security requirements
- **Resolution:** Use the an existing password database format: .kdbx. This format is already extremely popular and is one I use myself to store my personal passwords.

Unit testing Java sockets:

- **Problem:** I'd never worked with Java sockets before, and as a result had no idea how to go about writing unit tests for them.
- **Resolution:** I looked online but struggled to find consistent answers for my specific scenario. So I created a post on Stack Overflow with my queries. After a couple of weeks I received some good answers detailing how I could achieve this. Unfortunately by this point I'd already changed the implementation of the sockets and the information was no longer as useful.

4. Future Work

- Change original password
- Two factor authentication
- Open a new database without restarting the application
- Streamlined Group and Entry management such as 'drag and drop'
- Fully automated synchronisation
- KDF Iterations