

KeePassJ per  
“Programmazione ad Oggetti”

Di Santi Giovanni, Ercolani Francesco, Conti Massimiliano, Cellot Davide

29 settembre 2020

# Indice

<b>1</b>	<b>Analisi</b>	<b>2</b>
1.1	Requisiti . . . . .	2
1.2	Analisi e modello del dominio . . . . .	3
<b>2</b>	<b>Design</b>	<b>4</b>
2.1	Architettura . . . . .	4
2.2	Design dettagliato . . . . .	4
<b>3</b>	<b>Sviluppo</b>	<b>21</b>
3.1	Testing automatizzato . . . . .	21
3.2	Metodologia di lavoro . . . . .	22
3.3	Note di sviluppo . . . . .	23
<b>4</b>	<b>Commenti finali</b>	<b>25</b>
4.1	Autovalutazione e lavori futuri . . . . .	25
4.2	Difficoltà incontrate e commenti per i docenti . . . . .	26
<b>A</b>	<b>Guida utente</b>	<b>28</b>

# Capitolo 1

## Analisi

### 1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un keepass per salvare in modo sicuro le password. Al momento i keepass desktop più usati sono keepass2 [5] e keepassxc [7]. TODO: scrivere descrizione più lunga.

#### **Requisiti funzionali**

- Il database su cui sono salvate le password deve essere opportunamente autenticato e cifrato, in modo da non permettere a terze parti di leggere o manipolare il contenuto delle password.
- Gestione delle entry (add, edit, delete) con la possibilità di suddividere le entry in vari gruppi.
- Funzioni per generare in modo sicuro password e controllarne la validità.
- Possibilità di importare in chiaro un database in formato XML, in modo simile alla funzionalità di un keepass originale.

#### **Requisiti non funzionali**

- Possibilità di cifrare archivi esterni.
- Funzioni di sort e find per ordinare e cercare entry specifiche.
- Funzione expire che avvisa quando una password è da cambiare (es. 2 anni).

- Sezione Statistics che mostra le statistiche relative al proprio database (Es. il numero di account salvati)

## 1.2 Analisi e modello del dominio

Nella creazione del database bisogna inserire una master password. Dopodiché è necessario configurare vari parametri, tra cui il cipher e la Key Derivation Function (KDF) da usare. La KDF ha vari parametri opzionali da utilizzare per generare la key in modo più sicuro (più rounds) o più veloce (più parallelismo). Lo pseudocodice seguente spiega come la **KDF** e il **cipher** sono collegati:

```
plaintext = "this is the plaintext"
key = KDF(password)
ciphertext = cipher.encrypt(key, plaintext)
assert plaintext == cipher.decrypt(key, ciphertext)
```

I vari parametri crittografici del database sono configurabili tramite **KDB-Header**, che vengono usati dalla classe **KDB** per effettuare l'encryption e la decryption di array di byte arbitrari. Per lavorare sul database in chiaro usiamo un'ulteriore classe che è **DataBase** che permette di manipolare le entry e i gruppi.

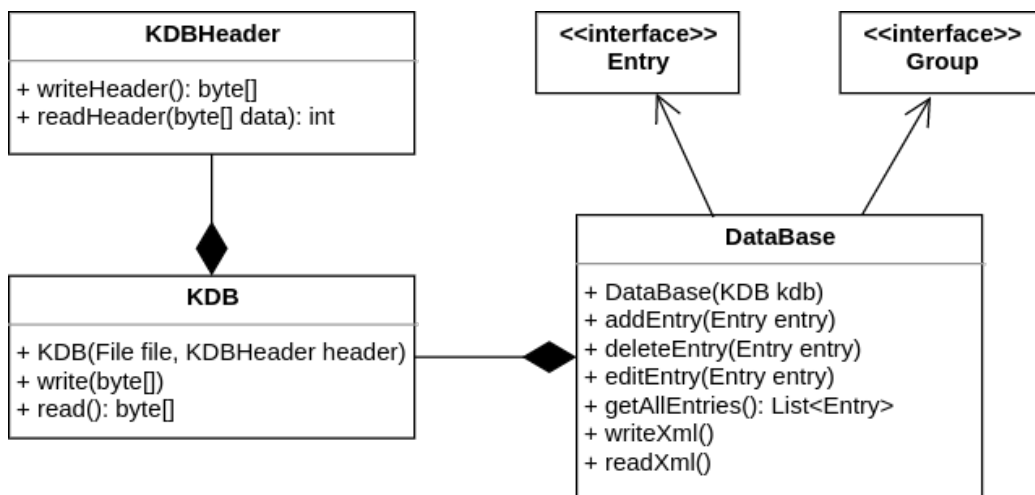


Figura 1.1: rappresentazione UML dell'analisi del progetto

# Capitolo 2

## Design

### 2.1 Architettura

Per la realizzazione di KeePassJ abbiamo scelto di utilizzare il pattern architetturale Model-View-Controller (MVC).

TODO: Inserire schema UML MVC.

### 2.2 Design dettagliato

#### Giovanni Di Santi

Il mio compito principale del progetto è stato quello di gestire la parte crittografica e definire la struttura dell'header del database. Prima di iniziare a modellare le varie parti mi sono documentato sulle tecnologie attuali dei vari keepass, il paper più citato è "On The Security of Password Manager Database Formats" [1]. Mi sono ispirato a come funziona il formato di keepass2 KDBX4. Il formato è composta da un *header* e un *body*. Nell'*header* sono definiti i vari cipher, kdf, e parametri per cifrare/decifrare il *body*. Il body decifrato in KDBX4 è formato da un XML, tuttavia questa parte è ha scelta di chi deve implementare il database in chiaro. Nel nostro caso Massimiliano ha deciso di usare XML perché ha una buona interoperabilità con Java.

La sezione del paper 4.6 evidenzia come il formato KDBX4 di keepass2 sia insicuro contro attacchi del tipo **MAL-CDBA**. Senza entrare nei dettagli, l'header del database non viene autenticato e quindi si aprono una serie di possibili attacchi teorici. Tuttavia il paper sopra citato non è coerente con il sito di keepass2 in cui si descrive KDBX4, infatti leggendo il sorgente di keepass2 si può notare che l'autenticazione viene eseguita. Probabilmente il paper essendo del 2012 non è aggiornato sui recenti cambiamenti.

## CryptoCipher

**CryptoCipher** è l'interfaccia che descrive i metodi necessari per effettuare l'encryption e la decryption di un array di **byte**.

Ogni implementazione disponibile di questa interfaccia è un AEAD Cipher [6] (Authenticated Encryption with Associated Data).

Ho scelto questo schema di encryption per rendere il database resistente ad attacchi del tipo **CCA** (Chosen Ciphertext Attack), cifrando il contenuto del database e autenticando sia il contenuto che l'header (Associated Data). Attualmente i cipher disponibili sono:

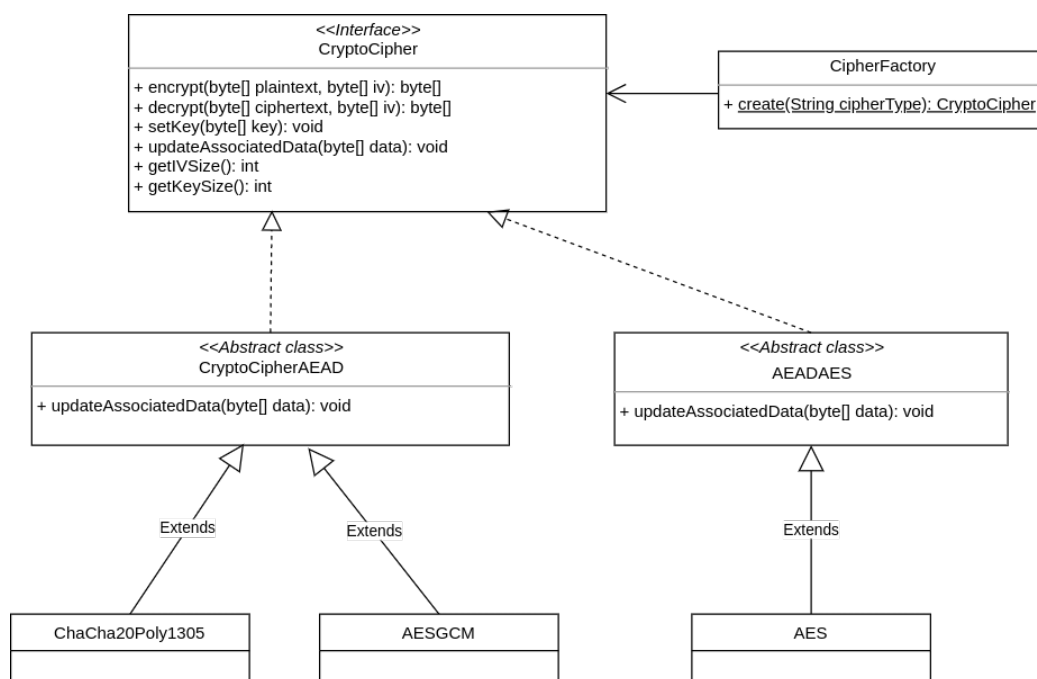


Figura 2.1: rappresentazione UML del pattern factory per creare un CryptoCipher

- ChaCha20-Poly1305 [3].
- AES-GCM [4].
- AES-256-CBC-HMAC-SHA-512 [2].

Esistono due **abstract class** diverse per implementare un **CryptoCipher**, poiché la costruzione di **AES** che sarebbe **AES-256-CBC-HMAC-SHA-512** è manuale, mentre **ChaCha20-Poly1305** e **AES-GCM** sono implementate

direttamente in openjdk11. La classe astratta **AEADAES**, permette di essere estesa per costruire altri cryptosystem come **AES-192-CBC-HMAC-SHA-384**, tuttavia ho deciso di estendere solo lo schema più sicuro. Nonostante i dati da cifrare e decifrare sono nella pratica degli `{Input,Output}Stream`, non ho usato le classi `CipherOutputStream` e `CipherInputStream` per:

- Rendere più semplice il suo utilizzo.
- Facilitare il testing delle varie implementazioni.

## KDF

**KDF** (Key Derivation Function) è l'interfaccia che descrive i metodi necessari e opzionali per generare una chiave simmetrica per cifrare/decifrare il database.

Gli algoritmi disponibili sono:

- Argon2.
- Scrypt.
- PBKDF2.

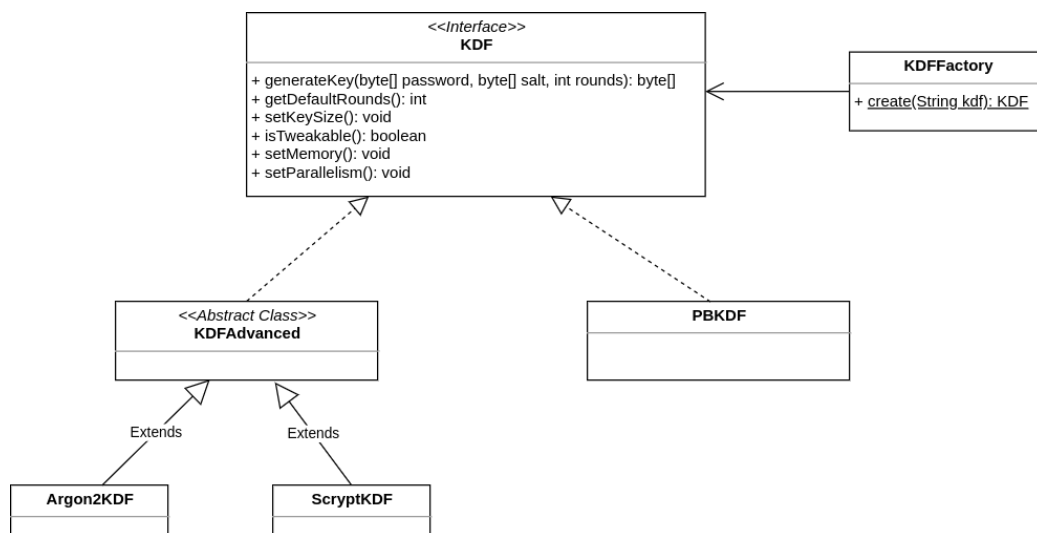


Figura 2.2: rappresentazione UML del pattern factory per creare un KDF

**Argon2** e **Scrypt** estendono **KDFAdvanced** poiché i loro algoritmi permettono di definire parametri extra come il parallelismo e la memoria

usata dal KDF. **PBKDF2** è un vecchio metodo per generare una chiave dalla password e l'unico parametro configurabile è il numero di round che usa internamente, per questo ho settato il campo **tweakable** a falso.

Per capire perché il pattern **Factory** è usato sia per creare **KDF** e **CryptoCipher** bisogna prima analizzare il parsing dell'header e la relativa encryption/decryption del database.

## KDB

Per progettare questa parte non ho usato le interfacce perché:

- Ho solo una implementazione disponibile.
- Sono più flessibile quando devo cambiare la signature di un metodo, senza dover usare un IDE o un LSP per il refactoring.
- Principio YAGNI e KISS.

**KDBHeader** è la classe che si occupa di:

- Parsare l'header (Lettura).
- Configurare i vari parametri (Scrittura).

**KDB** è la classe che tramite il **KDBHeader** si occupa di cifrare/decifrare dati arbitrari. Il pattern factory per **CryptoCipher** e **KDF** è utile quando in **KDB** si effettua l'operazione **encrypt** e **decrypt**. I metodi richiedono a **KDBHeader** il valore (String) del **Cipher** e del **KDF** che viene passato come parametro di `{Cipher,KDF}Factory.create()` per generare l'oggetto richiesto. I due metodi pubblici principali di **KDB** sono:

- **write**: che esegue l'encryption dell'array di byte in input e lo scrive sul file. Ad ogni write viene generato un nuovo IV, che viene poi usato per cifrare il plaintext. In questo modo si creano due versioni diverse dello stesso *body* e si rende lo schema crittografico sicuro contro vari attacchi di tipo **CCA**.
- **read**: che legge il file e lo decrypta. Questo metodo lancia un **IOException** se il file non esiste o un **AEADBadTagException** quando il file è corrotto. Il file può risultare corrotto o perché la password è sbagliata o perché è stato effettivamente manipolato. Non lancio tipi diversi di eccezioni (es. **BadPaddingException**) in base a vari tipi di errore, per evitare vari tipi di attacchi (praticamente difficili, ma teoricamente possibili) come il padding oracle.



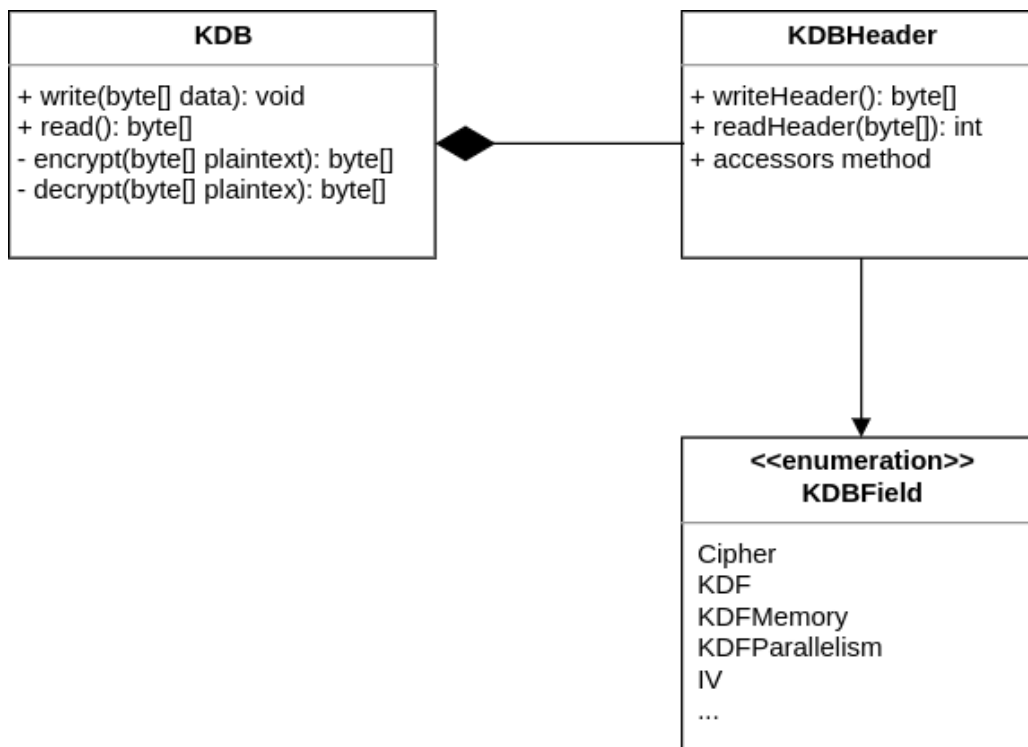


Figura 2.3: rappresentazione UML di KDB

## Francesco Ercolani

Il mio ruolo nel progetto KeePassJ è stato quello di sviluppo e gestione dell'interfaccia grafica del programma. L'interfaccia grafica è gestita attraverso controllers situati nel package **view.controllers**, ciascuno collegato al rispettivo file fxml presente nella directory **src/main/resources/view**. I controllers utilizzano classi nel package controller per gestire la logica dell'interfaccia.

### view.controllers

Come già citato in precedenza, il package **view.controllers** contiene tutti i controllers dei rispettivi file fxml; il mio compito è stato quello di creare sia i file fxml che di gestire la logica, con opportuni metodi e campi interni ai rispettivi controllers, di una parte di essi. La directory **src/main/resources/** è gestita nel seguente modo: all'interno della sottodirectory **/view**, è presente il file **MainMenu.fxml** e due ulteriori sottodirectory **/createnew** e **/database**. All'interno di **/createnew** sono presenti i file:

- **chooseNameDb.fxml**: è l'interfaccia per l'inserimento del nome e della descrizione del database.
- **chooseEncryptionSet.fxml**: è l'interfaccia per l'impostazione del metodo di encryption del database.
- **choosePassMenu.fxml**: è l'interfaccia per la scelta della password del database.

All'interno di /database sono presenti i file:

- **OpenDatabase.fxml**: è l'interfaccia per l'inserimento della password per l'apertura del database.
- **ManageMenu.fxml**: è l'interfaccia dove vengono visualizzati gli account registrati nel database corrente e dove si può scegliere di aggiungere un account o un gruppo.
- **AddEntry.fxml**: è l'interfaccia dove si aggiungono gli account che si vogliono gestire.
- **AddGroup.fxml**: è l'interfaccia dove si aggiungono i gruppi ai quali apparterranno gli account inseriti.

Il file MainMenu.fxml che si trova dentro la directory /view è l'interfaccia principale che viene caricata all'esecuzione del programma e contiene i due pulsanti principali: "Create new database" e "Open database" per appunto rispettivamente creare ed aprire un database. Oltre che della creazione dei file fxml citati, mi sono occupato della creazione di tutti i rispettivi controllers e della gestione della logica interna di essi eccetto per: **AddEntryController.java**, **AddNewGroupController.java** e **ManageMenuController.java** (compito delegato ad altri componenti del gruppo di progetto).

Dal controller di MainMenu.fxml **MainMenuController.java** partono due "linee" separate in riferimento alla creazione o all'apertura di un database. Nel caso si clicchi su "Create new database", la linea prosegue verso il controller dell'interfaccia per l'inserimento del nome e della descrizione del database (chooseNameDb.fxml) **ChooseNameDBController.java**. Dentro quest'ultimo sono di fondamentale importanza i campi per l'acquisizione dei dati inseriti dall'utente.

Cliccando su "Continue" la linea prosegue passando al controller dell'interfaccia adibita al settaggio dell'encryption del database **ChooseEncrSetController.java**, che a sua volta acquisirà anch'esso i dati inseriti dall'utente.

Continuando con la creazione si arriva all'ultima interfaccia della linea controllata dal controller **ChoosePassController.java**, il quale alla pressione del bottone "Confirm", attraverso l'oggetto di tipo KDBHeader, setta tutti i valori inseriti dall'utente.

La seconda linea invece porta all'interfaccia di inserimento della password per l'apertura del database selezionato in precedenza.

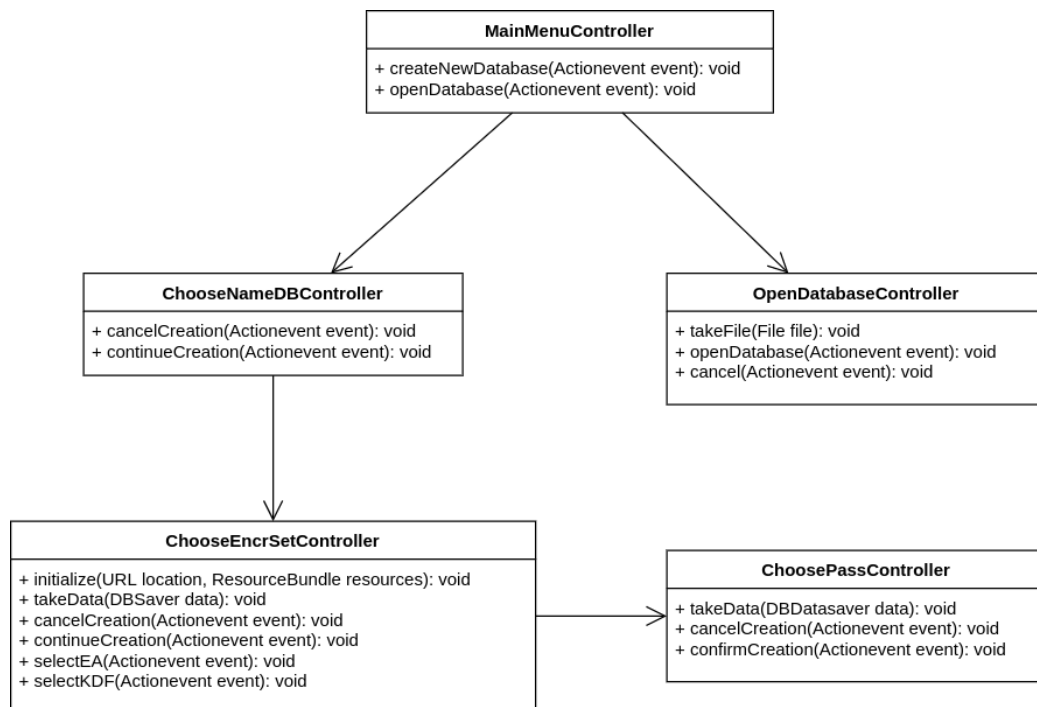


Figura 2.4: rappresentazione UML delle due linee che collegano i controllers

I controllers al loro interno hanno un metodo per ciascuna azione che l'utente compie nell'interfaccia.

Ciascun controller fa uso di classi presenti nel package **controller**. Per il loading dei file fxml si utilizza la classe **FxmlFilesLoaderImpl.java** che implementa la relativa interfaccia **FxmlFilesLoader.java**.

Questa classe ha diversi metodi per fare il caricamento dell'interfaccia in modi differenti: alcuni caricano il file con metodo standard, altri hanno parametri per il passaggio di dati da controller a controller, infatti alcuni controller come da **OpenDatabaseController.java** in poi, hanno bisogno dei dati inseriti precedentemente.

Per implementare questo passaggio ho creato una classe **DBDataSaverIm-**

**pl.java** che salva i dati inseriti e li passa, attraverso metodi di **FxmlFilesLoaderImpl**, al controller successivo.

È stato scelto questo approccio per due principali motivi:

- In questo modo la conferma e il settaggio dei dati avviene soltanto alla fine del procedimento di creazione.
- Il passaggio da controller a controller è più semplice in termini di trasferimento.

Ogni controller che richiede dati dal precedente ha quindi un metodo **takeData()** che prende in ingresso l'oggetto di tipo **DBDataSaver**. I controllers utilizzano inoltre la classe **FxmlSetterImpl.java** che implementa l'interfaccia **FxmlSetter.java**, per operazioni come:

- Settaggio di spinner o warning dialogs.
- Acquisizione dello stage corrente.

Il settaggio effettivo per la creazione avviene in **ChoosePassController.java**, alla pressione del pulsante "Confirm" dopo aver scelto il nome e la destinazione del file (deve avere necessariamente l'estensione **.kdbj**), attraverso le classi **KDBHeader** e **KDB**.

Se si sceglie di aprire un database esistente, si apre la schermata di inserimento della password controllata da **OpenDatabaseController.java** e, in caso di inserimento corretto, si apre la schermata di gestione degli account controllata da **ManageMenuController.java**.

Anche in questo caso il passaggio di dati tra controllers avviene attraverso l'utilizzo della classe **FxmlFilesLoader**.

#### **Vantaggi di questo approccio per il passaggio dei dati:**

- Semplicità di utilizzo: è facile passare i dati da un controller all'altro in quanto viene passato soltanto l'oggetto responsabile del raggruppamento di essi.
- Non c'è bisogno di settaggi intermedi sugli oggetti **KDBHeader** e **KDB** in quanto viene settato tutto alla conferma finale.

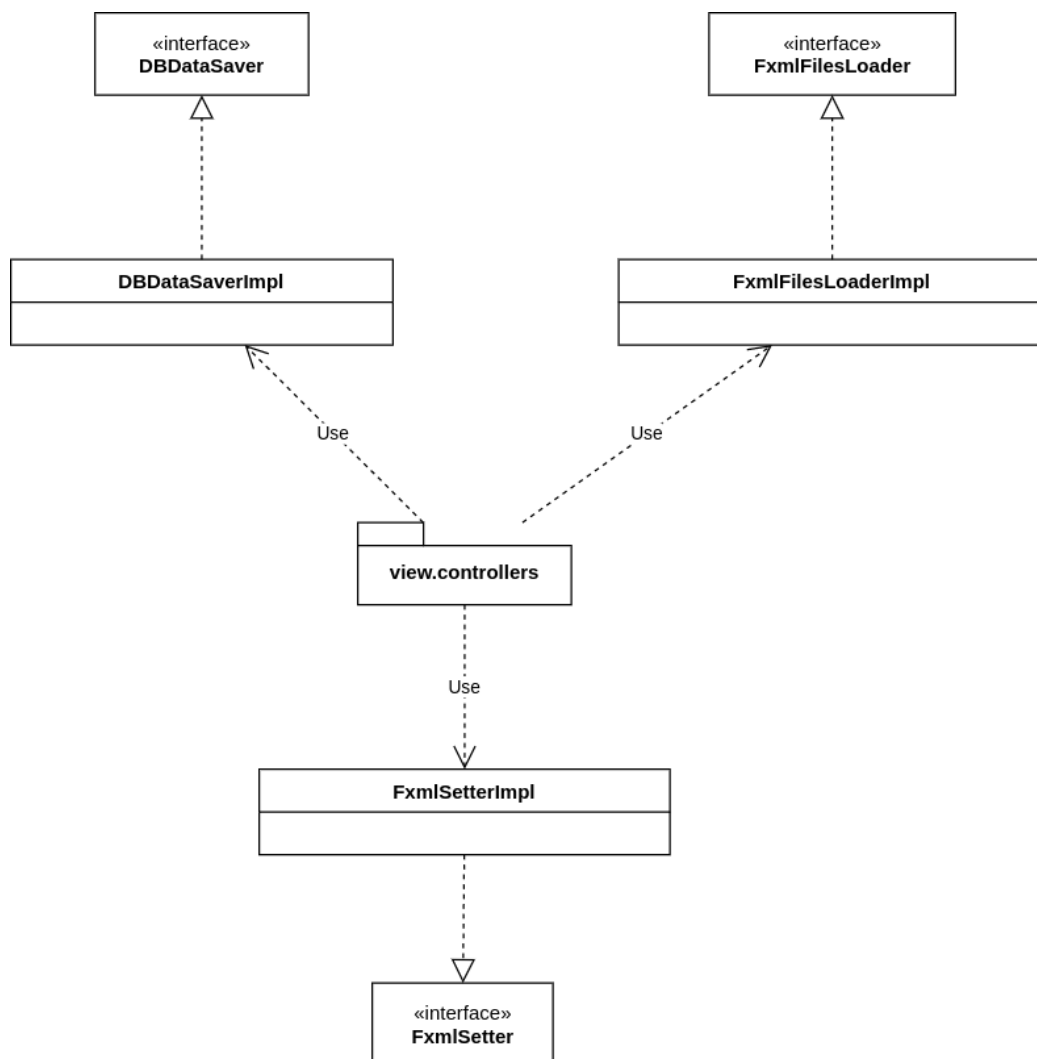


Figura 2.5: rappresentazione UML della relazione tra le classi nel package controller e i controllers in view

#### Svantaggi di questo approccio per il passaggio dei dati:

- Ridondanza di codice nella classe **FxmlFilesLoaderImpl**.
- Necessità di modifiche/aggiunte nella classe **FxmlFilesLoaderImpl** nel caso di cambi/aggiunte di funzionalità nell'applicazione.

## Esecuzione dell'applicazione

La classe **ViewImpl.java** che implementa l'interfaccia **View.java** nel package **view**, è incaricata del loading della prima scena del programma. La classe **Main.java**, situata all'interno del package **application**, estende la classe **Application** di **javafx** e implementa quindi il metodo **start()** che è il main entry point dell'applicazione. All'interno del metodo **main** il programma viene lanciato con il metodo statico **launch()**.

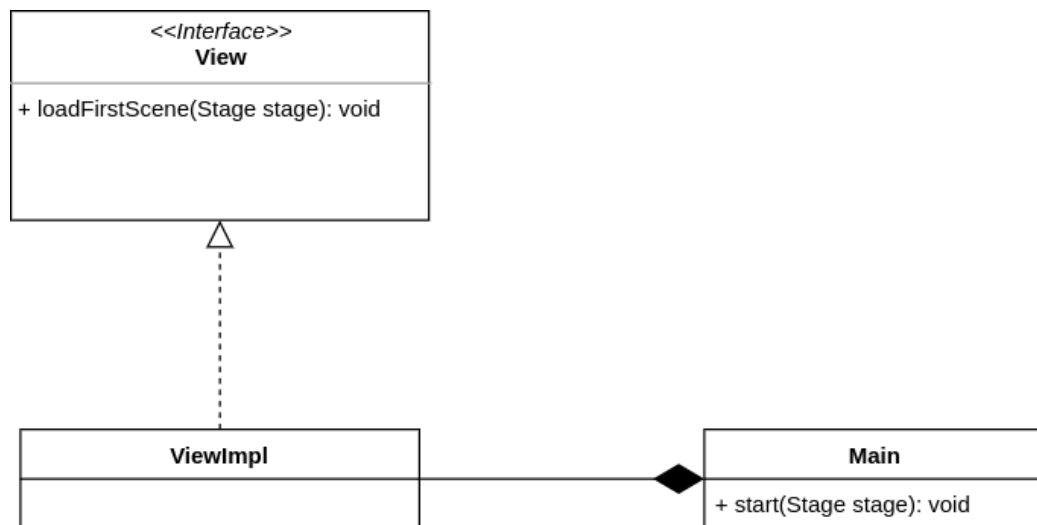


Figura 2.6: rappresentazione UML dell'esecuzione dell'applicazione

## Massimiliano Conti

Il mio compito invece è stato di creare e gestire il **database** e gli elementi contenuti in esso, tra cui **Entry** e **Group** (package **model.db**), e di convertire il database stesso in un file di tipo XML (package **model.export**). Per progettare questa parte non ho usato interfacce perchè non erano utili ai fini dell'applicazione implementazioni aggiuntive. In fine ho gestito i **Controllers** relativi alla visualizzazione del database e relativa creazione di **Entry** e **Group** presenti nel package **view.controllers**.

## Database

**Database** è la classe che si occupa di gestire e mantenere i dati inseriti dall'utente. Contiene i metodi necessari per l'uso e la modifica degli

`ArrayList` usati con relativa creazione ed eliminazione degli oggetti. Oltre a contenere un oggetto `String` per il mantenimento del nome assegnato e un oggetto `KDB` (creato da Giovanni).

Gli oggetti gestiti dalla classe `Database` sono:

- `Entry`
- `Group`

**Entry** è la classe su cui si basa il `Database` perchè con i suoi campi `nomeAccount`, `username`, `password`, `groupName`, `url` e `note` mantiene ogni dato che l'utente inserisce riguardo l'account che sta salvando sull'applicazione.

**Group** è la classe adibita alle categorie che l'utente crea o può creare. Ho creato una classe apposita per una più corretta gestione in modo da poter attribuire un nome e anche una descrizione oltre che impedire all'utente nel momento di creazione di una `Entry` l'inserimento di un nome sbagliato fornendo la lista già presente o la possibilità di creare un nuovo `Group`.

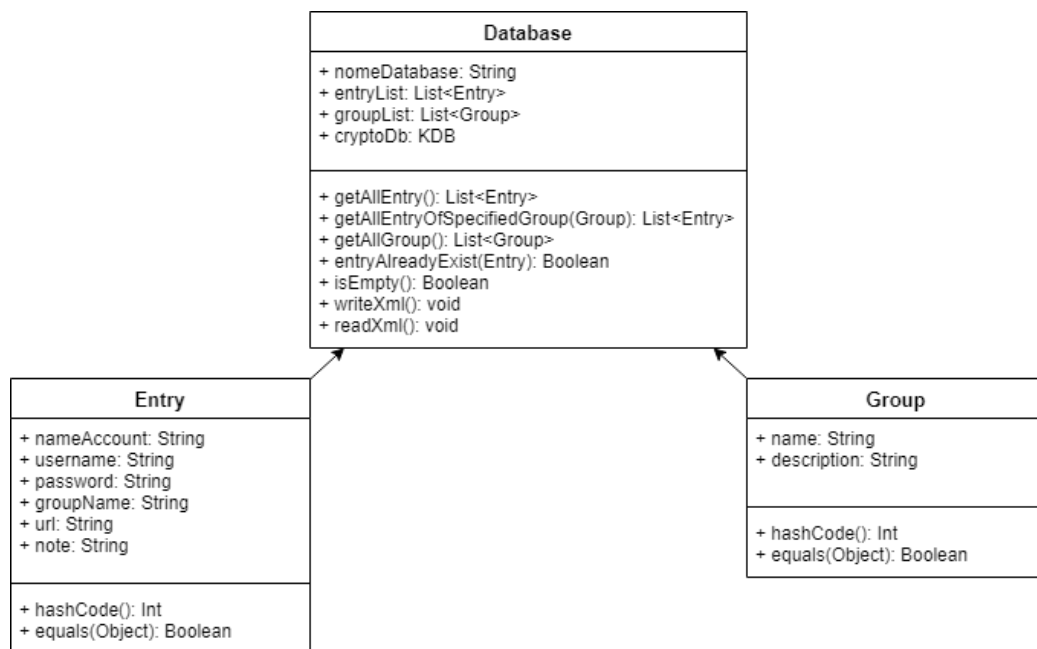


Figura 2.7: rappresentazione UML del Database con Entry e Group

## ConvertXML

**ConvertXML** fa uso della libreria JAXB, architettura per mappare Java objects e documenti XML (link to project homepage.). Questa classe viene usata da due metodi della classe Database, tramite cui creare o recupera dati da un documento XML.

Il funzionamento si basa su due metodi *static*:

- toXml
- fromXml

**toXml** prende come parametri di input una istanza di Database e con l'uso degli oggetti **JAXBContext** e **Marshaller** restituisce in output una String contenente un documento XML propriamente riempito con Entries, Groups e il nome del Database.

**fromXml** invece come paramentro di input prende un documento XML contenuto in una String e tramite gli oggetti **JAXBContext** e **Unmarshaller** lo converte e crea una nuova istanza della classe Database e la restituisce in output per riempire l'oggetto che ha chiamato la conversione.

La classe ha poi un metodo privato **getTempFile** a cui il metodo fromXml fa appoggio per la creazione di un file usato dall'oggetto Unmarshaller, ma con il vantaggio di non scriverlo fisicamente su disco.

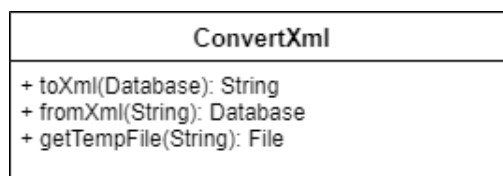


Figura 2.8: rappresentazione UML di ConvertXml



## Controllers

Mi sono poi occupato di gestire i **controller** che si occupano di inserire Entry e Group dentro la classe Database e visualizzare i dati.

Tra cui:

- AddEntryController
- AddNewGroupController
- ManageMenuController

In **AddEntryController** in particolare ho riempito la ComboBox relativa ai Groups (inseriti tramite **AddNewGroupController**) e controllato se i campi necessari come Title, username e password non sono vuoti prima di creare una nuova Entry che poi viene inserita nella list dentro Database e poi salvato il file aggiornato.

(inserimento e controllo della password sono parte di Davide).

In **ManageMenuController** ho creato e riempito le colonne dentro la *TableView* in cui mostro la lista di Entry e la lista di Group.

Tramite i bottoni è poi possibile eliminare l'Entry selezionata nella tabella e accedere poi alle view di AddEntry e AddGroup.

## Davide Cellot

Il mio compito principale nella realizzazione del progetto è stato quello di gestire la parte legata alle password. Dato che il progetto si basa sul salvataggio di password in database e la loro crittazione, è abbastanza importante gestire in modo ottimale tali password.

Per prima cosa ho cercato su internet vari siti e applicazioni che inglobavano anche l'inserimento di password e da tali ho preso ispirazione per la creazione del controllo di validità. Dopodiché mi sono concentrato sul calcolo della robustezza (strength) ovvero una misura di efficienza contro eventuali attacchi che una password può subire. Tanto questo numero è più alto, tanto è più sicura la password.

## PasswordStrength

**PasswordStrengthImpl** è la classe incaricata di calcolare la robustezza delle password tramite il metodo pubblico *getStrength()*. Tale metodo l'ho impostato statico in modo tale da non dover creare ogni volta un'istanza della classe per invocarlo, evitando un procedimento reputato superfluo.

Inoltre per analizzare il vettore di caratteri nel quale è salvata la password presa in esame, ho utilizzato altre due classi: **CheckCharacters** e **ConsecutiveCharacters**. Il motivo è puramente pratico in quanto queste due classi le avevo già implementate per un progetto extrauniversitario e riutilizzando lo stesso codice, con minime modifiche, ho potuto risparmiare tempo.

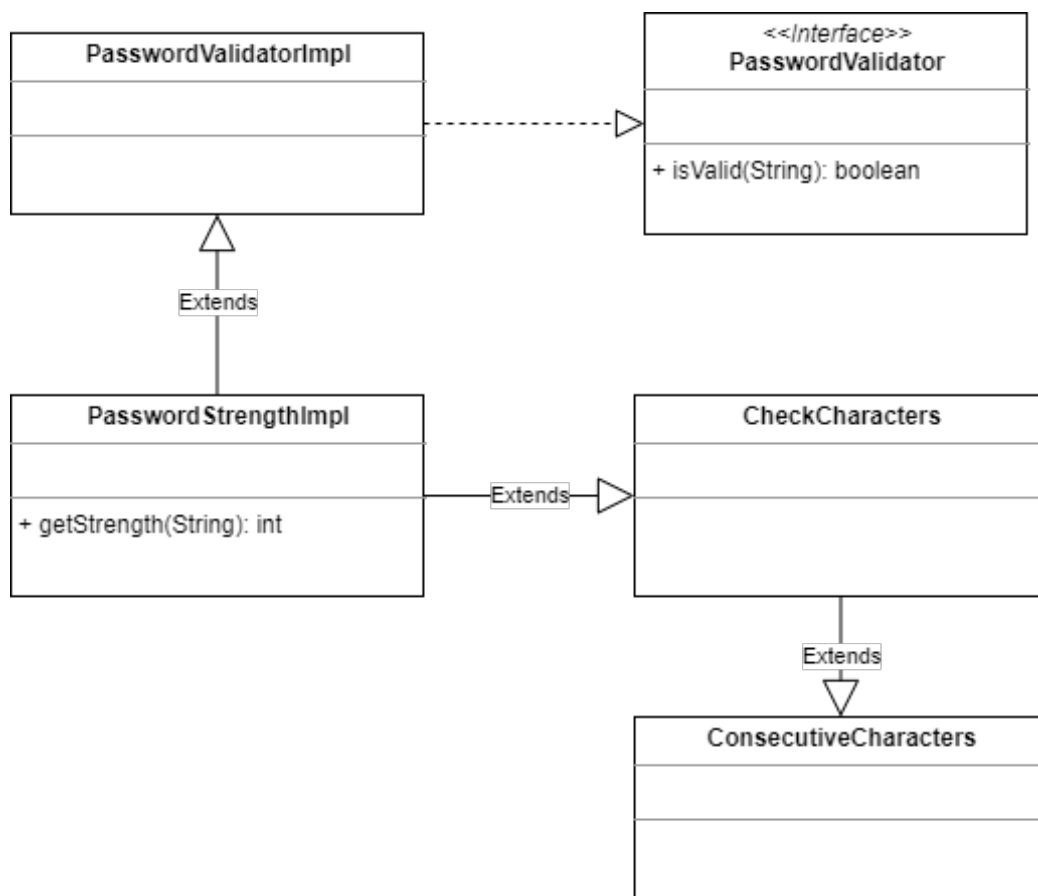


Figura 2.9: rappresentazione UML sezione Strength

## GeneratePasswordRandom

Tale interfaccia viene implementata nella classe **GeneratePasswordRandomImpl** che crea un vettore di lunghezza prefissata e lo riempie in modo casuale con:

- Lettere Maiuscole.
- Lettere minuscole.
- Numeri
- Caratteri speciali, specificati dal programmatore

Ciò avviene tramite l'uso della classe *Random*.

Tale vettore viene convertito in stringa e viene testata la sua validità. Infatti si è deciso che una password per essere valida deve avere almeno otto caratteri e deve contenere almeno un numero.

Generata la password casuale, ci si appoggia alla classe **PasswordValidatorImpl** che tramite il suo metodo *isValid()* ne testa la validità. Se la password risulta valida viene ritornata dal metodo **generatePassword()**, altrimenti viene rigenerata da zero.

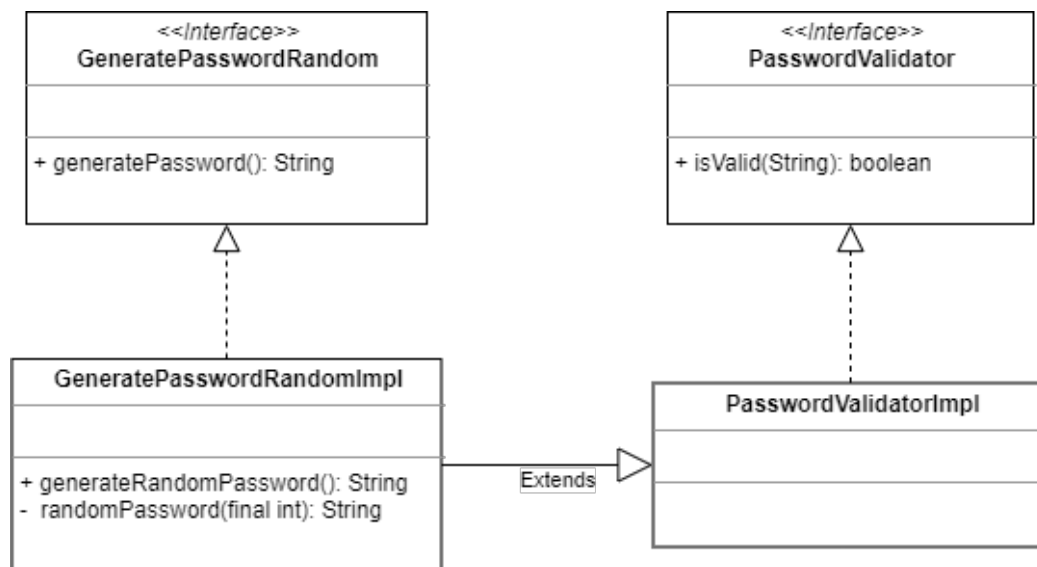


Figura 2.10: rappresentazione UML sezione Generate Random

## AddEntryController

Infine mi sono occupato della GUI referente all'aggiunta della "Entry" inserendo una *ProgressBar* per visualizzare la robustezza della password immessa, ho fatto in modo che tale barra si settasse in automatico all'inserimento di una nuova password. Per settare la *ProgressBar* viene richiamata la funzione **PasswordStrengthImpl**.

Ho inserito poi un bottone con una semplice *label* associata per vedere la password in chiaro.

Successivamente mi sono concentrato sulla generazione random, inserendo un bottone che se premuto riempie in automatico la *passwordField* con una password causale, richiamando la funzione precedentemente descritta (**GeneratePasswordRandom**).

Ho gestito tutte le azioni nel controller nel pieno rispetto del pattern MVC.

Per ultimo ho impostato dei controlli suppletivi sulla password, ovvero una volta premuto il pulsante per creare la Entry, viene controllato se la password immessa è corretta (controllo effettuato richiamando **Password-Validator**), in caso positivo viene creata la “Entry”, altrimenti appare un *AlertDialog* con le istruzioni necessarie per inserire una password corretta.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Durante lo sviluppo del nostro progetto abbiamo utilizzato `Junit` per testare il corretto funzionamento delle varie classi. **Funzionalità testate automaticamente:**

- *CryptoCipher*: viene testato il corretto comportamento dei vari cipher utilizzando lo specifico `Factory`.
- *KDF*: vengono testate le varie KDF utilizzando lo specifico `Factory`.
- *Crypto Util*: test di varie utility per crittografia, come il PKCS#7 padding e SHA256.
- *KDBHeader*: viene testato il corretto funzionamento del **KDBHeader**, settando vari parametri e controllando il corretto funzionamento dei getter.
- *KDB*: test sulla scrittura e lettura di diverse combinazioni di cipher e KDF.
- *Database*: viene controllato la corretta creazione ed uso delle liste usate e dei relativi getter.
- *Entry*: test sui getter e setter.
- *ConvertXml*: test sulla corretta conversione di un Database sample da e verso xml.

I test vengono eseguiti anche in remoto tramite l'apposita `bitbucket pipelines` che esegue la build del progetto e lancia i vari test.

## 3.2 Metodologia di lavoro

Nel complesso, le parti inizialmente concordate sono state rispettate ma con qualche modifica rispetto agli assegnamenti iniziali. La **"gestione import/export dei dati su database locale"** inizialmente assegnata a Davide Cellot, è stata svolta da Massimiliano Conti che, in un primo momento, aveva la **"gestione generazione password random e controllo robustezza password tramite apposita GUI"**, parte svolta appunto da Davide. La **"gestione suddivisione account per categoria"** assegnata a Francesco Ercolani, è stata assegnata a Massimiliano Conti per motivi equità delle parti. La **"gestione sezione Statistics"** non è stata implementata per valutazioni successive effettuate sulla base della sua complessità di implementazione. Per la comunicazione tra i componenti del gruppo sono stati utilizzati: un gruppo **Telegram** e un canale **Discord** per riunioni a voce con eventuale condivisioni di schermo.

Dopo questi chiarimenti, elenchiamo le parti definitive di ciascun membro del gruppo:

- **Giovanni Di santi**: gestione aspetti crittografici che riguardano l'encryption/decryption del database e gestione dei parametri disponibili per eseguire la cifratura.
- **Francesco Ercolani**: gestione degli aspetti grafici dell'applicazione con creazione dei rispettivi file FXML e dei relativi controller. Gestione della logica dei controller legati alla fase di creazione del database e della prima schermata di apertura dello stesso.
- **Massimiliano Conti**: gestione del Database **in chiaro** e della sua conversione in documento Xml. Implementazione dei controller per aggiunta ed eliminazione Entry, Group e del Menu di visualizzazione.
- **Davide Cellot**: gestione generazione casuale password in fase di inserimento di una nuova Entry con controllo sulla sua robustezza e la relativa implementazione grafica e logica nel rispettivo controller.

Durante la fase di sviluppo ci sono stati momenti in cui più elementi del gruppo hanno lavorato su file comuni, in particolare su: **ManageMenu.fxml**, **AddEntry.fxml**, **AddGroup.fxml**, e suoi relativi controllers **ManageMenuController.java**, **AddEntryController.fxml** e **AddNewGroupController.fxml**. Si è lavorato in comune inoltre sul file **FxmlFilesLoaderImpl.java** del package **controller**.

Durante la fase di sviluppo è stato utilizzato *Distributed version control systems (DVCS)* **Git**. Ciascun del gruppo ha clonato la repository remota precedentemente creata su BitBucket in modo da poter lavorare autonomamente in locale. È stato impostato da tutti il comando `git config --global pull.rebase true` in modo che ad ogni `pull` venisse effettuato il `rebase` in automatico. Sono stati creati due branch principali: **develop** e **main**. L'applicazione è stata sviluppata interamente sul primo, mentre sul secondo è stato fatto il merge finale. In fase di terminazione è stato creato anche il branch **paper** riguardante la stesura della relazione finale.

### 3.3 Note di sviluppo

#### Giovanni Di Santi

- *javax.crypto* e *java.security*: usata per lavorare con Cipher, KDF, e MessageDigest.
- *Stream*: usata per manipolare l'header in modo efficace ed elegante.
- *Libreria Google Guava*: lavorare con i byte array in java non è comodo. Questa libreria ha varie classi per semplificare il lavoro tra cui Bytes.
- *ByteBuffer*: per convertire in little endian i bytes, i `Data{Input,Output}Stream` in java lavorano solo in big-endian.
- *Libreria Argon2 e Scrypt*: Questi due KDF non erano disponibili dentro *javax.crypto*.
- *Libreria Apache Commons*: per convertire byte array in formato esadecimale.

Ho iniziato lo sviluppo leggendo i sorgenti di keepass2, keepassxc e di pykeepass. Non ho trovato librerie simili a `construct` in java per parsare l'header utilizzando un linguaggio dichiarativo. L'unica alternativa era `kaitai.io`, ma il supporto a java era limitato. Avendo avuto esperienze con le librerie crittografiche in python e C, il passaggio a java non è stato difficile. Per i pattern progettuali ho consultato il materiale didattico e online, ho trovato che il pattern factory mi aiutasse a risolvere vari tipi di problemi per la generazione automatica di oggetti e l'ho usato.



## Francesco ercolani

In quanto incaricato di sviluppare la maggior parte dell'interfaccia grafica dell'applicazione, ho fatto un largo uso della libreria esterna **JavaFX**, in particolare dei moduli:

- **javafx.controls**: per la gestione dei componenti grafici.
- **javafx.base**: per la gestione degli elementi logici (proprietà, collezioni, eventi ecc...).
- **javafx.fxml**: per la gestione dei file FXML.

## Massimiliano Conti

Nella gestione del Database non ho fatto uso di particolari librerie, anzi ho cercato di usare dove potevo l'interfaccia degli stream.

In oltre ho imparato ad usare:

- **JAXB**: in quanto utile nella mia parte di parsing di documenti xml.
- **FXML**: per delle modifiche minori alle view di Database e dati.
- **javafx.fxml**: per la gestione dei controller relativi a Database, Entry e Group.

## Davide Cellot

Nella parte riguardante le classi, le interfacce e l'ereditarietà non ho fatto uso di nessuna libreria esterna. Nella parte invece dove ho utilizzato le classi e i metodi precedentemente creati, ho fatto uso della libreria **JavaFX** per la parte grafica, in particolare i moduli:

- **javafx.controls**: per la gestione dei componenti grafici
- **javafx.base**: per la gestione degli eventi

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### Giovanni Di Santi

L'idea iniziale del progetto è stata mia perché mi piace lavorare con la crittografia. Mi è piaciuto come ho realizzato la parte di **CryptoCipher**, tuttavia l'architettura che riguarda **KDF** non mi sembra eccellente. Un difetto che ho trovato è stata l'integrazione manuale dei vari metodi in **KDF** dentro gli accessor method di **KDBHeader**. Dovrebbero esistere librerie, plugin o pattern di programmazione su misura per quel tipo di problema ma non li ho usati. In **KDB** non ho aderito al Single-responsibility principle (SRP) per comodità, infatti la classe effettua sia la **write** che la **read**, al posto di suddividere i compiti in due classi diverse. L'ho fatto principalmente perché entrambi i metodi usano lo stato dell'oggetto (campi) per essere eseguiti, quindi era molto conveniente tenerli all'interno della stessa classe.

#### Francesco Ercolani

Il mio ruolo all'interno del progetto KeePassJ è stato più difficile di quanto mi aspettassi. Le interfacce da me create sono molto basiche in quanto la mia esperienza con la libreria JavaFX era nulla, e per questo spero di trovare il tempo per proseguire lo sviluppo estetico su un branch personale. La gestione logica dei controller non è ottimale in quanto fa uso di classi da me create che contengono codice ridondante. L'applicazione in sé è abbastanza "povera" di grafica in quanto priva di diverse funzionalità presenti invece in password manager attualmente in commercio. Il progetto è molto valido, le applicazioni password manager sono estremamente utili e ricoprono una grande responsabilità nel mantenimento in sicurezza dei propri dati personali,

ed è per questo che ne sono stato entusiasta fin dall'inizio quando mi è stato proposto come idea da Giovanni Di Santi. Lo sviluppo di questo processo, nonostante i problemi riscontrati, mi ha fatto trovare la voglia di progettare un'applicazione da zero e, allo stesso tempo, come già detto, spero in futuro di ritagliarmi uno spazio per proseguire lo sviluppo di KeePassJ.

## Davide Cellot

Per il progetto sono stato contattato da Francesco Ercolani e Giovanni di Santi, dato che ero alla ricerca di un gruppo di lavoro. Ho accettato subito la loro proposta perché mi sembrava un progetto diverso dal solito, non si trattava di un gioco ma di un gestionale e avevo molta curiosità di vedere come si sviluppava un simile tipo di progetto.

Mi è piaciuto molto lavorare sulla mia parte, con la gestione delle password e quindi con la manipolazione delle stringhe. Ad esempio, in **PasswordValidator** ho usato le *wildcards* per fare matching sulla stringa cercando se al suo interno è presente almeno un numero, sfruttando anche le conoscenze apprese durante il corso.

## 4.2 Difficoltà incontrate e commenti per i docenti

### Giovanni Di Santi

Ho dovuto lavorare di più del dovuto perché i miei compagni fino a settembre non hanno implementato nessuna parte di codice significativo. A inizio giugno la modellazione e l'implementazione della maggior parte del mio compito erano completate, tuttavia mi sono ritrovato ad aggiungere parti di codice in più a settembre, poiché i miei compagni hanno iniziato a lavorare solo in quel mese. Vari problemi sarebbero nati subito se avessimo lavorato insieme. Ad esempio ho dovuto aggiungere vari metodi che aiutassero Francesco a modellare la GUI, difatti tutti i miei commit di settembre sono getter e setter aggiuntivi. Avendo progettato l'architettura abbastanza bene non ho avuto problemi ad aggiungere quelle parti di codice, tuttavia ho dovuto riprendere in mano un codice scritto 3 mesi prima e continuare a lavorarci sopra. Avrei preferito consegnare con la deadline C.

L'intenzione iniziale quando ho iniziato a lavorare sul progetto era di renderlo compatibile con KDBX4, tuttavia ho notato dopo le prime 15 ore che

quel formato è molto complicato da implementare. Non esiste documentazione del formato e quindi ho speso molto tempo a reverse ingegnerizzare il codice originale. Dopo le prime ore di implementazioni ho mantenuto la struttura, ma ho deciso di usare nuovi schemi crittografici. Non mi sono pentito della scelta fatta poiché mi ha reso più creativo.

## Francesco Ercolani

L'architettura del progetto è rimasta molto vaga per i primi mesi dopo l'avvio del progetto. Inizialmente le funzionalità di KeePassJ dovevano essere tante e questo ha suscitato in me da subito perplessità riguardo al fatto di riuscire a sviluppare un software con così tante features complesse in poco tempo. Inoltre la situazione Covid, e quindi l'impossibilità di vedersi fisicamente, ha creato all'interno del gruppo diverse difficoltà di comunicazione dovute a problemi tecnici e/o personali. L'inizio dello sviluppo è partito in momenti diversi da parte dei componenti e la mancanza di collaborazione e di specifiche su come si dovessero svolgere le parti assegnate ha creato ulteriori difficoltà di comprensione. Personalmente ritengo che fino al mese di agosto ci sia stata troppa poca collaborazione e troppo poco impegno nel delineare **chiaramente** quali fossero **concretamente** le parti software da sviluppare. Mi prendo le mie responsabilità per il ritardo di consegna della mia parte, ma ritengo che parte dei problemi siano nati da situazioni che non dipendono da me o dagli altri componenti del gruppo. Tuttavia nel complesso reputo che il lavoro svolto sia comunque molto molto buono data l'inesperienza progettuale e di linguaggio comune a tutti e 4.

## Davide Cellot

Ho incontrato alcune difficoltà nello sviluppo del progetto, dovute principalmente alla mancanza di esperienza. Ho sbagliato la programmazione del lavoro ritrovandomi all'ultimo con molto lavoro da svolgere. La vicinanza della deadline con la sessione di esami non ha certamente aiutato e il progetto si è dilungato nel tempo.

Ho anche riscontrato uno scarso lavoro collettivo, sia all'inizio che successivamente, questo è forse da imputare alla mancanza di un leader all'interno del gruppo, di un capo che scandisse i tempi e che tenesse sotto controllo l'avanzamento del progetto. Secondo me anche questo punto è dovuto alla mancanza di esperienza.

Per me questo era il primo progetto di gruppo a cui ho lavorato e posso solo dire che questa esperienza, con i suoi lati positivi e negativi, ha contribuito a farmi crescere in esperienza e competenza.

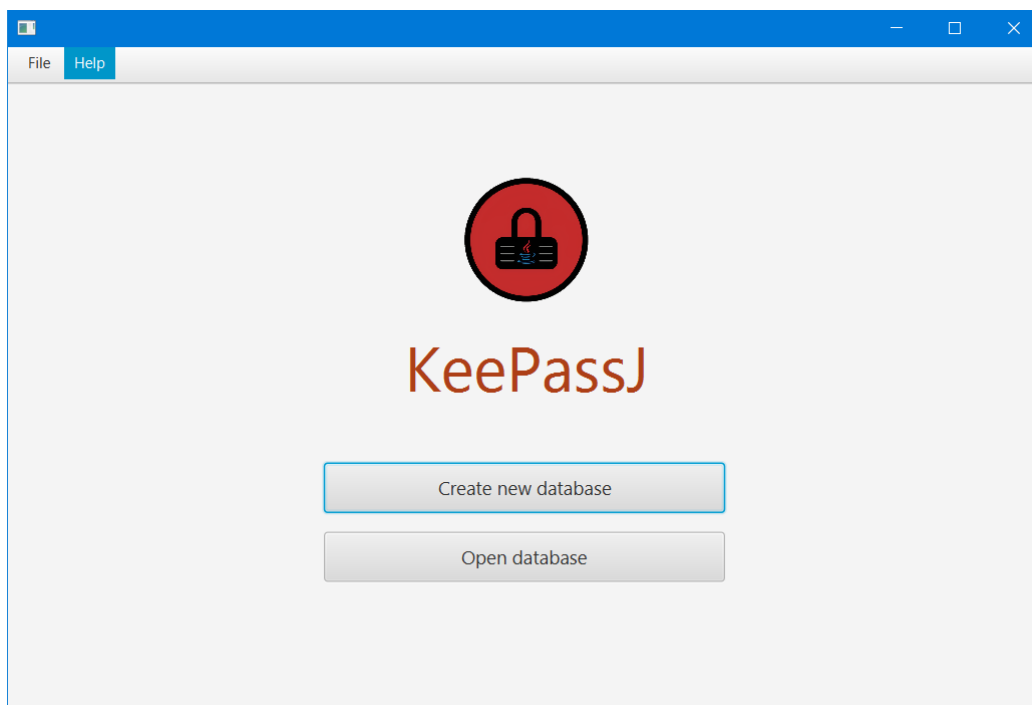
# Appendice A

## Guida utente

Segue la Guida Utente per il corretto utilizzo dell'applicazione.

### Initial Window

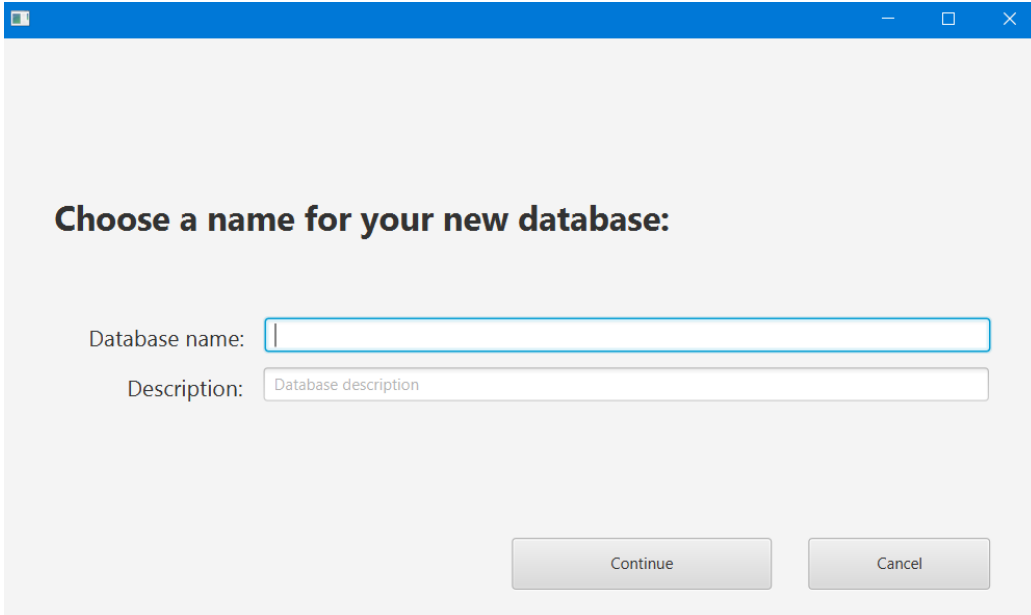
L'applicazione si apre mostrando la finestra riportata di seguito. In tale finestra è possibile compiere due azioni: creare un nuovo database dove salvare le nostre password, oppure caricarlo se lo si è già precedentemente creato.



Cliccando su *Open Database* si aprirà la finestra di Esplora Risorse da dove poter cercare il nostro database in locale.

## Create New Database

Invece premendo su *Create new database* si aprirà la seguente finestra.

A screenshot of a Windows-style dialog box titled "Choose a name for your new database:". The dialog has a blue title bar with standard window controls. Inside, there are two text input fields. The first is labeled "Database name:" and is empty. The second is labeled "Description:" and contains the placeholder text "Database description". At the bottom right, there are two buttons: "Continue" and "Cancel".

**Choose a name for your new database:**

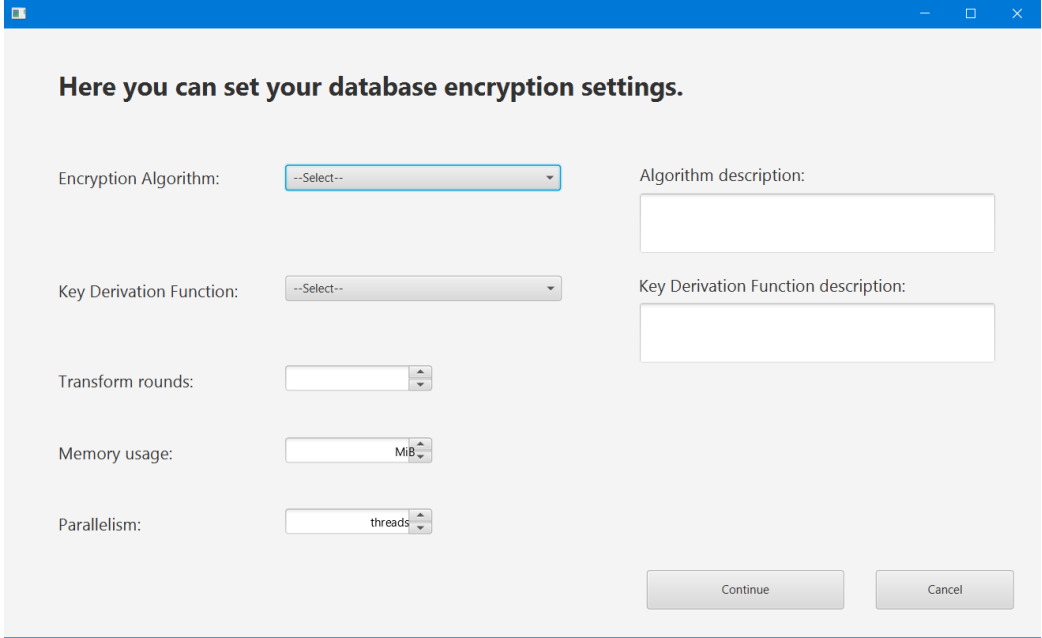
Database name:

Description:

Si immette il nome del database che vogliamo creare e una breve descrizione. Nel caso si tentasse di procedere senza aver immesso il nome e/o la descrizione, il sistema lancia un messaggio di errore.

Una volta settato il nome e cliccato su *Continue*, si apre una nuova finestra (riportata di seguito) dove settare le impostazioni di crittografia.

## Database Encryption Settings



The screenshot shows a Windows-style dialog box titled "Here you can set your database encryption settings." It contains several configuration options:

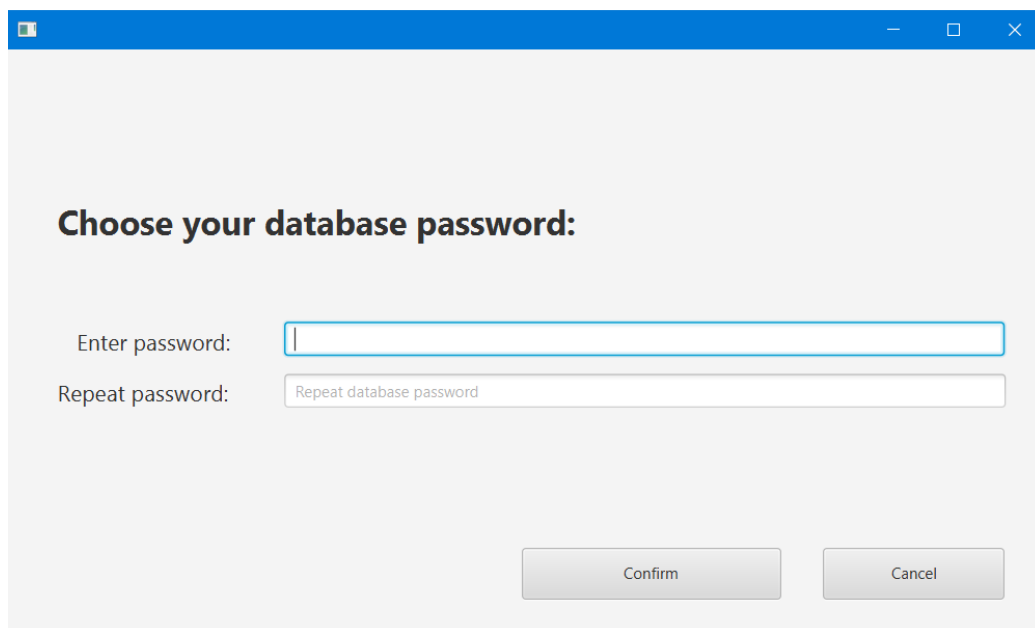
- Encryption Algorithm:** A dropdown menu currently showing "--Select--".
- Algorithm description:** A text input field.
- Key Derivation Function:** A dropdown menu currently showing "--Select--".
- Key Derivation Function description:** A text input field.
- Transform rounds:** A numeric input field with up and down arrow buttons.
- Memory usage:** A numeric input field with a unit dropdown set to "MiB".
- Parallelism:** A numeric input field with a unit dropdown set to "threads".

At the bottom right, there are two buttons: "Continue" and "Cancel".

Tramite le comboBox si può scegliere tra varie opzioni di **Encryption Algorithm** e **Key Derivation Function**. Dopo aver impostato tutte le caratteristiche si può continuare, cliccando su *Continue*.

## Choose database password

Si aprirà la seguente finestra dove impostare la password per accedere al nuovo database.

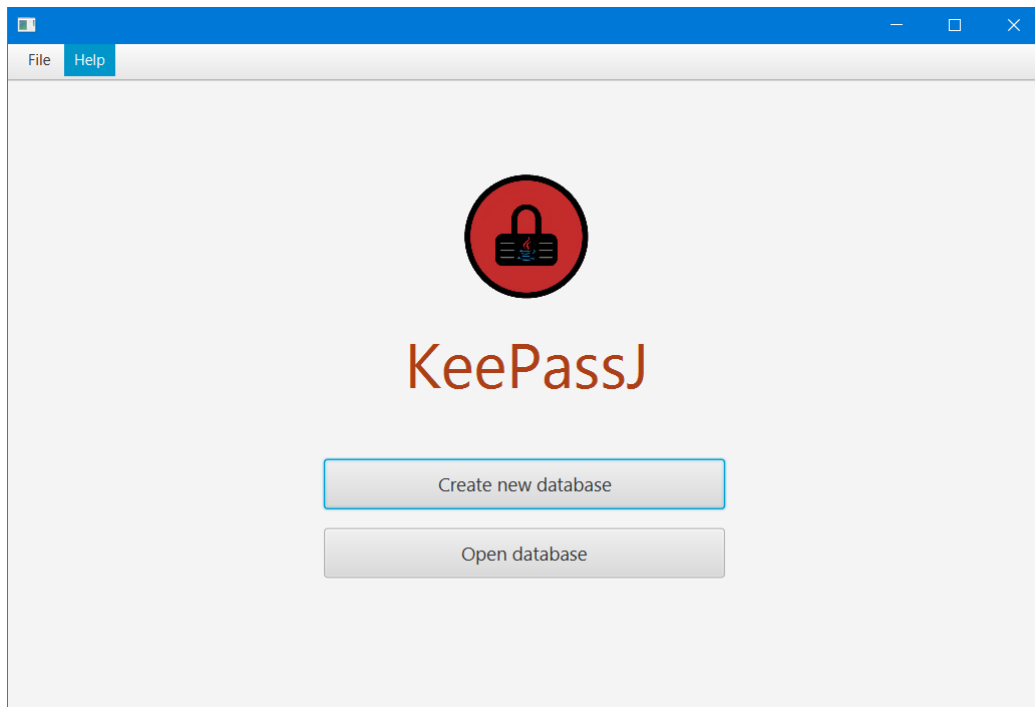


The image shows a Windows-style dialog box titled "Choose your database password:". It has a blue title bar with standard minimize, maximize, and close buttons. The main area is light gray. Below the title, there are two input fields. The first is labeled "Enter password:" and is empty. The second is labeled "Repeat password:" and contains the placeholder text "Repeat database password". At the bottom right, there are two buttons: "Confirm" and "Cancel".

Se non si immette nessuna password e si tenta di proseguire, verrà mostrato un messaggio di errore. Una volta immessa la password e premuto su *Confirm* si apre l'Esplora Risorse e possiamo scegliere dove salvare il database. Una volta salvato si tornerà alla prima finestra.



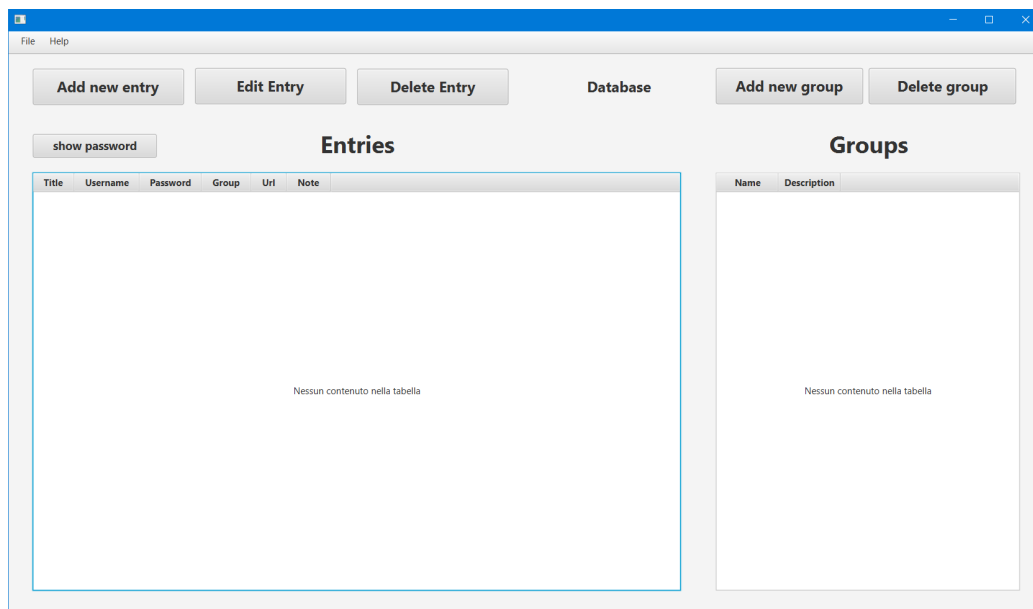
## Open Database



Si può procedere quindi all'apertura, cliccando su *Open Database* e andando a cercare il database e ad aprirlo, inserendo la password corretta.

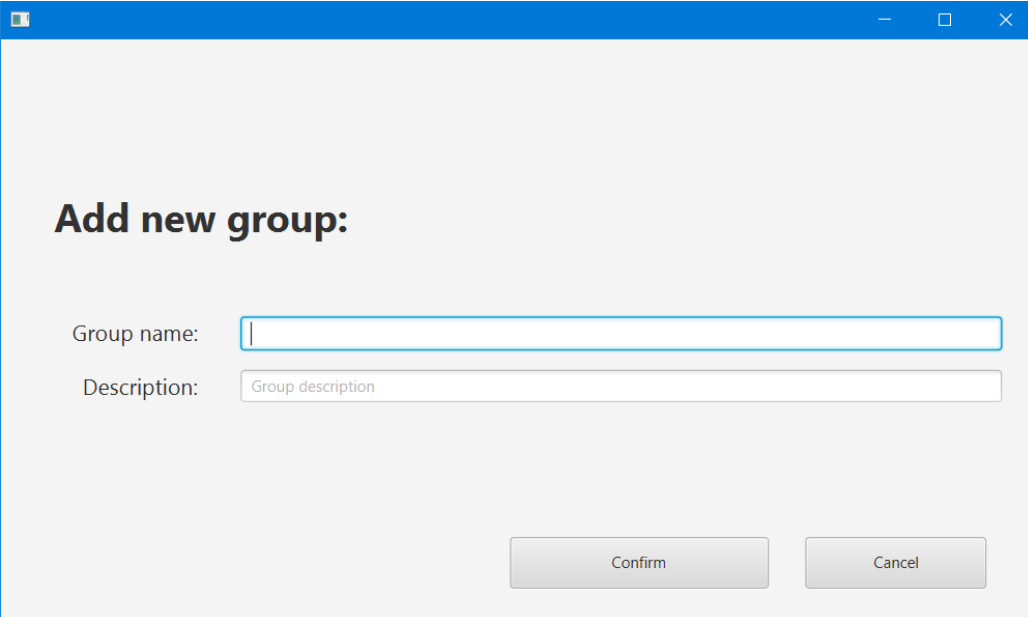
# Main Window

Si accederà quindi alla seguente finestra principale.



In essa è presente una serie di bottoni.

- **Add new Group** serve a creare un gruppo, ogni entry deve appartenere obbligatoriamente ad un gruppo. Cliccando su tale bottone si apre la seguente finestra.



**Add new group:**

Group name:

Description:

Si inserisce il nome e una descrizione e si clicca su *Confirm*, in caso si man-  
canza di un campo appare il solito messaggio di errore. Dopo aver confermato  
la creazione si ritorna alla finestra principale.

- ***Delete Group*** permette di cancellare il gruppo precedentemente se-  
lezionato dall'elenco presente sulla destra della finestra.
- ***Add new Entry*** apre la seguente finestra, dove è possibile creare un  
nuova entry.

**Add entry:**

Title:

Username:

Password:  **SHOW**

ProValPass

**Generate password** 80.0%

URL:

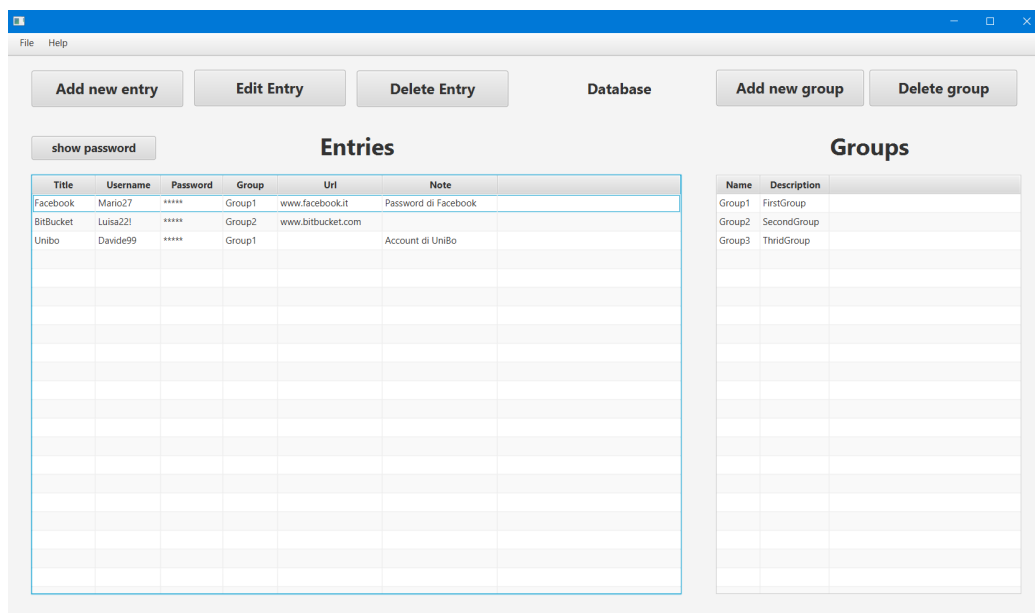
Notes:

Group:  **Add new group**

**Confirm** **Cancel**

In questa finestra si può creare la entry settando tutti i campi presenti. Cliccando su **Show** è possibile vedere in chiaro la password immessa. **GeneratePassword** crea una nuova password casuale. È possibile anche vedere la robustezza della password immessa tramite l'apposita ProgressBar. Cliccando su *Confirm* si crea la entry, prima però viene controllato se la password è valida oppure no, in caso contrario appare un AlertDialog con le istruzioni per immettere una corretta password. È inoltre necessario settare un gruppo di appartenenza, altrimenti non verrà creata la entry. Se non c'è nessun gruppo si può cliccare su *Add new group* che rimanda alla finestra precedentemente illustrata.

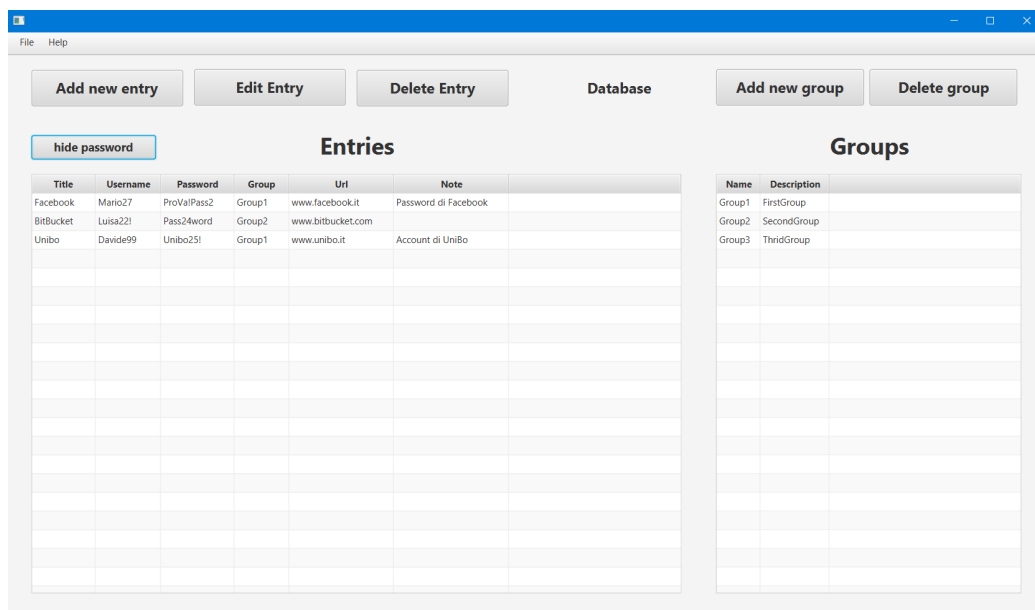
Una volta creata la entry si ritorna alla finestra principale.



Si noti la presenza del nome del database aperto, presente in alto al centro (nel caso in figura: *Database*).

Ora è possibile vedere la entry appena creata sulla sinistra, mentre sulla destra si vedono i gruppi presenti nel database.

- Il bottone **Edit Entry** serve a modificare una entry precedentemente selezionata. Si apre la finestra di *Add Entry* nella quale è possibile modificare tutti i parametri impostati e poi salvare nuovamente la entry.
- Infine il bottone **Delete Entry** cancella la entry selezionata.
- Cliccando su **Show Password** si assisterà alle modifiche visibili nella seguente finestra.



Le password delle entries sono visibili in chiaro mentre il bottone *Show Password* è cambiato in *Hide Password* e se cliccato nuovamente torna ad oscurare le password presenti.

## End of Paper

Come visto nella Guida Utente, l'applicazione in questione gestisce la creazione e il caricamento di database criptati. Dentro ognuno dei quali vengono creati dei gruppi nei quali vengono suddivise le entries create.

# Bibliografia

- [1] Gasti and Rasmussen. On the security of password manager database formats, 2012. <https://www.cs.ox.ac.uk/files/6487/pwvault.pdf>.
- [2] IETF. Authenticated encryption with aes-cbc and hmac-sha. 2014. <https://tools.ietf.org/html/draft-mcgrew-aead-aes-cbc-hmac-sha2-05>.
- [3] IETF. Chacha20-poly1305. 2018. <https://tools.ietf.org/html/rfc8439>.
- [4] NIST. Aes-gcm. 2007. <https://csrc.nist.gov/publications/detail/sp/800-38d/final>.
- [5] Dominik Reichl. Keepass2, 2020. <https://keepass.info>.
- [6] Phillip Rogaway. Authenticated-encryption with associated-data. 2002. <https://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf>.
- [7] KeePassXC Team. Keepassxc, 2020. <https://keepassxc.org/>.