# Relazione Tecnica: Password Manager

## Panoramica Architetturale

L'applicazione è basata su Flask (microframework Python) e segue un pattern MVC semplificato:  
- Model: Gestione database SQLite (database.py, services.py)  
- View: Template HTML/Jinja2 (file .html)  
- Controller: Route Flask (routes.py)  
- Security: Crittografia AES-256 (cryptography.py) e generazione password sicure (password\_generator.py)  
- Config: Impostazioni centralizzate (config.py)

## Moduli Principali

### routes.py (Controller)

Gestisce la logica di navigazione, autenticazione e operazioni CRUD.  
  
Funzioni Chiave:  
- login\_required (Decorator)  
 - Ruolo: Restringe l'accesso alle pagine protette.  
 - Logica: Verifica la presenza di utente\_id in sessione.  
   
 def login\_required(f):  
 def decorated\_function(\*args, \*\*kwargs):  
 if 'utente\_id' not in session:  
 flash('Per favore, effettua il login', 'error')  
 return redirect(url\_for('main.login'))  
 return f(\*args, \*\*kwargs)  
 return decorated\_function  
  
- aggiungi\_password  
 - Ruolo: Aggiunge una nuova password crittografata.  
 - Flusso:  
 1. Deriva la chiave master con PBKDF2  
 2. Crittografa la password con Fernet  
 3. Salva nel DB  
  
- cambia\_password\_master  
 - Ruolo: Ri-crittografa tutte le password con una nuova chiave master.  
 - Logica:  
 1. Decrittografa tutte le password con la vecchia chiave  
 2. Genera nuova chiave derivata dalla nuova password  
 3. Ri-crittografa e aggiorna il DB

### services.py (Model/Business Logic)

Gestisce operazioni su utenti e password.  
  
Classi/Funzioni:  
- UserManager  
 - create\_user(username, password)  
 - Logica: Genera hash della password (Werkzeug) e salt crittografico.  
 - Dipendenza: CryptographyManager.generate\_salt()  
 - get\_master\_key(user\_id, password)  
 - Ruolo: Deriva la chiave master per crittografia.  
 - Output: Chiave AES-256 derivata con PBKDF2.  
  
- PasswordService  
 - get\_user\_passwords(user\_id, master\_key)  
 - Logica: Decrittografa tutte le password dell'utente.  
 - Interazione: Chiama CryptographyManager.decrypt\_password()

### cryptography.py (Security Core)

Gestisce crittografia end-to-end.  
  
Funzioni:  
- derive\_key(password, salt)  
 - Algoritmo: PBKDF2-HMAC-SHA512 con 200.000 iterazioni.  
 - Output: Chiave Fernet per cifratura.  
   
 kdf = PBKDF2HMAC(algorithm=hashes.SHA512(), length=32, salt=salt, iterations=200\_000)  
 return base64.urlsafe\_b64encode(kdf.derive(password.encode()))  
  
- encrypt\_password(password, master\_key)  
 - Meccanismo: Cifratura simmetrica con AES-256 (Fernet).  
 - Output: Password cifrata in Base64.

### password\_generator.py (Utility)

Genera password sicure con criteri personalizzabili.  
  
Funzione Principale:  
- generate\_secure\_password()  
 - Parametri: Lunghezza, caratteri speciali, maiuscole, numeri.  
 - Logica: Combinazione di caratteri casuali con almeno 1 simbolo per categoria.  
   
 chars = string.ascii\_lowercase  
 if use\_uppercase: chars += string.ascii\_uppercase  
 if use\_numbers: chars += string.digits  
 if use\_special\_chars: chars += "!@#$%&\*"

### database.py (Persistence)

Inizializza il DB e gestisce le connessioni.  
  
Funzioni:  
- init\_database()  
 - Schema:  
 - utenti: username, password\_hash, encryption\_salt  
 - password\_salvate: password\_sito\_encrypted (crittografata)  
 - Dipendenza: SQLite3 integrato in Python.  
  
- get\_connection()  
 - Ruolo: Fornisce connessioni DB con pattern context manager (with).

### Template HTML (View)

- base.html: Layout comune con navbar dinamica.  
- dashboard.html: Tabella password con pulsanti "Mostra/Copia".  
- aggiungi.html: Form con generatore password integrato (JavaScript).

## Dipendenze e Configurazioni

### Dipendenze Principali:

| Pacchetto | Ruolo |  
|--------------|-----------------------------------|  
| Flask | Routing e gestione richieste HTTP |  
| Werkzeug | Hashing password (sha256) |  
| cryptography | Crittografia AES/Fernet |  
| SQLite3 | Database integrato |

### File di Configurazione (config.py):

class Config:  
 SECRET\_KEY = os.urandom(24) # Sessione sicura  
 DATABASE = 'password\_manager.db'  
 PBKDF2\_ITERATIONS = 100 # Difesa brute-force  
 SALT\_LENGTH = 32 # Lunghezza salt crittografico  
 MIN\_PASSWORD\_LENGTH = 8

## Flusso Crittografia Password

1. Registrazione Utente:  
 - Genera salt casuale (secrets.token\_bytes(32))  
 - Deriva chiave master con PBKDF2  
 - Salva hash password + salt nel DB  
  
2. Salvataggio Password:  
 - Crittografa password con chiave master (Fernet)  
 - Salva testo cifrato nel DB  
  
3. Recupero Password:  
 - Deriva chiave master da password utente  
 - Decrittografa password con chiave derivata

## Sicurezza e Best Practice

- Critografia End-to-End: Ogni password è cifrata con chiave derivata dalla password master.  
- Salt Unici: Ogni utente ha un salt casuale per PBKDF2.  
- Difesa Brute-Force: 100 iterazioni PBKDF2.  
- Input Validation: Sanificazione input in tutti i form (es. strip()).  
- Error Handling: Gestione centralizzata errori crittografici e DB.

[Link presentazione](https://gamma.app/docs/Relazione-Tecnica-Password-Manager-parxn3h919p1rph)