PassEZ

➢ PROJECT OVERVIEW:

→ PassEz is a full-stack application that allows users to securely store, manage, and retrieve their credentials.

➢ Features:

- ✓ User authentication: Secure login and registration.
- ✓ CRUD operations for stored credentials.
- ✓ Responsive web interface.
- ✓ Zero knowledge architecture.
- ✓ Data Security: Encrypted storage.
- ✓ Runs on Docker.
- ✓ GitHub Actions.

➢ Tech Stack:

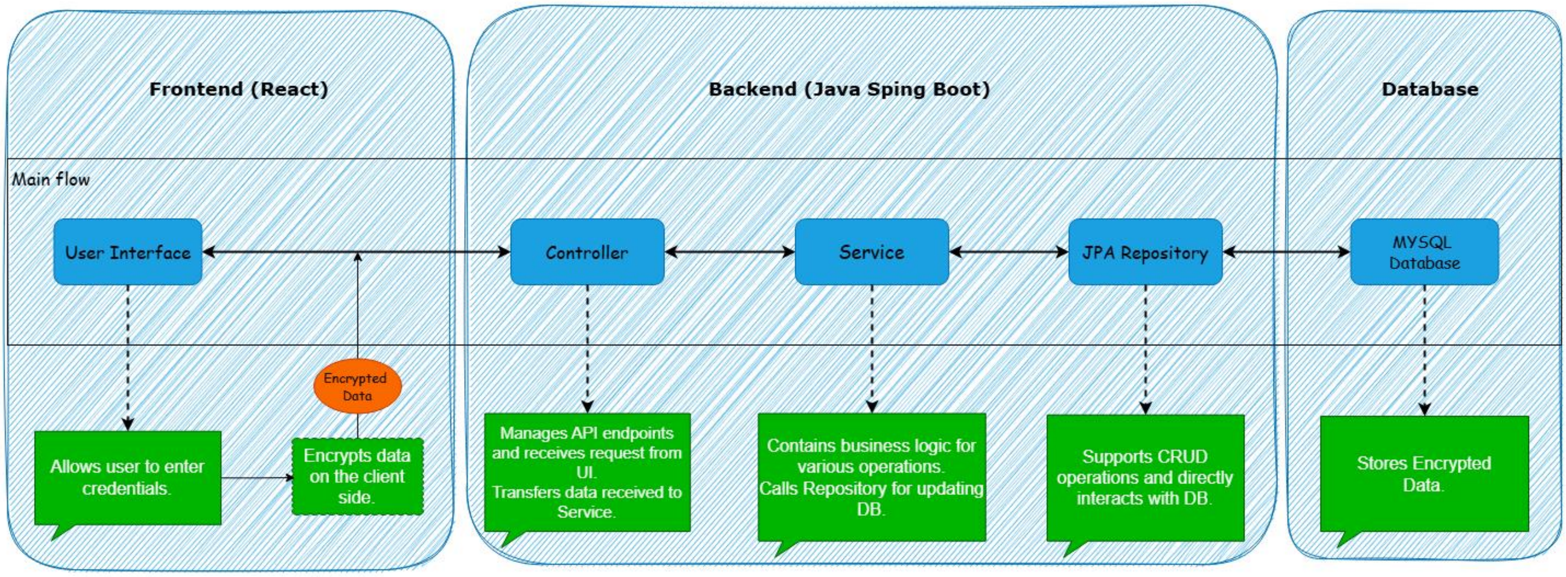
- ✓ Backend (Java Spring Boot):
  - REST API Development.
  - Secure Credential Storage.
  - Validation and Error Handling using Hibernator Validator.
  - Logging using SLF4J.
- ✓ Frontend (React):
  - Responsive Design with React and Tailwind CSS.
  - API Integration with Axios.
- ✓ MYSQL Database:
  - Integration with MYSQL Database using JPA.
  - Custom Query methods.
  - Auto Schema-Generation by JPA.
- ✓ Others:
  - Deployment: Hosted backend on Render.
  - Testing: Junit(backend).
  - Version Control: Git and GitHub.

➢ User flow:

1. User Inputs Details on Frontend:

   → The user enters details into the form on the webpage:

   - **Service Name** (e.g., "Gmail").
   - **Username** (e.g., "user@gmail.com").
   - **Password** (e.g., "secureP@ssword123").



Frontend (React)
Backend (Java Sping Boot)
Database
Main flow
User Interface
Controller
Service
JPA Repository
MYSQL Database
Encrypted Data
Allows user to enter credentials.
Encrypts data on the client side.
Manages API endpoints and receives request from UI.
Transfers data received to Service.
Contains business logic for various operations.
Calls Repository for updating DB.
Supports CRUD operations and directly interacts with DB.
Stores Encrypted Data.

2. Frontend Encrypts Data:
   - → Before sending data to the backend, the username and password fields are encrypted using the AES-ECB encryption algorithm with a secret key (only available to the client).:
   - → The encrypted data is sent to the backend through a POST request over HTTPS.
3. Backend Receives Encrypted Data:
   - → The Spring Boot Controller receives the encrypted data as part of the API request payload and forwards it to the Service layer.
4. Service Saves Data Without Decrypting:
   - → The Service layer passes the data to the Repository layer for storage.
   - → At no point does the backend attempt to decrypt the data; it handles the encrypted information as-is.
5. Repository Stores Encrypted Data in MySQL:
   - → The JPA repository persists the encrypted data in the credentials table.
6. User Requests Data Retrieval:
   - → For retrieval, the user sends a GET request to the backend (e.g., `/api/credentials/:id`).
7. Backend Returns Encrypted Data:
   - → The backend retrieves the stored encrypted data from the database and sends it back to the client as-is, without decryption.
8. Frontend Decrypts Data:

→ Upon receiving the encrypted data, the React frontend uses the same secret key (generated every time on the client side) to decrypt the `username` and `password`. The decrypted details are then displayed to the user on their device.

➢ Zero Knowledge Architecture:

- **What is Zero Knowledge Architecture?**

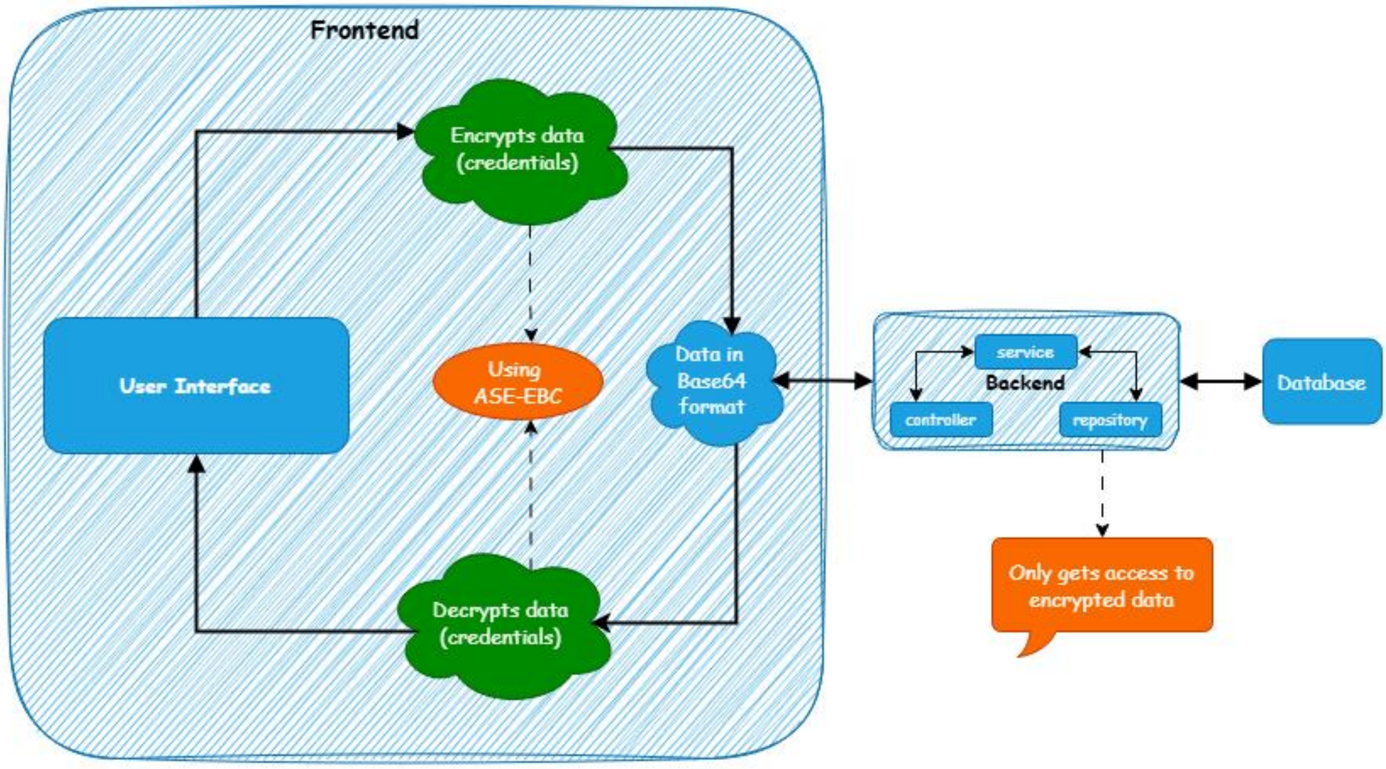
→ It is a cryptographic principle that ensures a party (here, Backend) cannot access or learn about the data being processed or stored.

→ Data encryption and decryption happen exclusively on the client side, and only the user has access to the decryption keys.

- **Key features:**

→ End-to-End Encryption (E2EE):

- All data transmitted between the client and the server is encrypted using AES-EBC.
- Encryption happens on the user's device (client-side), and the server only sees and stores encrypted data.



→ Client-Side Key Management:

- Users are the sole owners of their encryption keys.
- Without the keys, even the server cannot decrypt the data.

→ No Plaintext Data on Server:

- The server never sees data in its original form.
- Any breaches at the server level expose only encrypted data, which is useless without the decryption keys.

➢ **Encryption :**

- **Signup and Login Processes :**

Signup -

1. Collect Email and Master Password:

   → The user provides their email address and a master password during signup.

2. Generate a Secure Salt:

   → A cryptographically secure 128-bit (16-byte) salt is generated using a strong random salt generator.

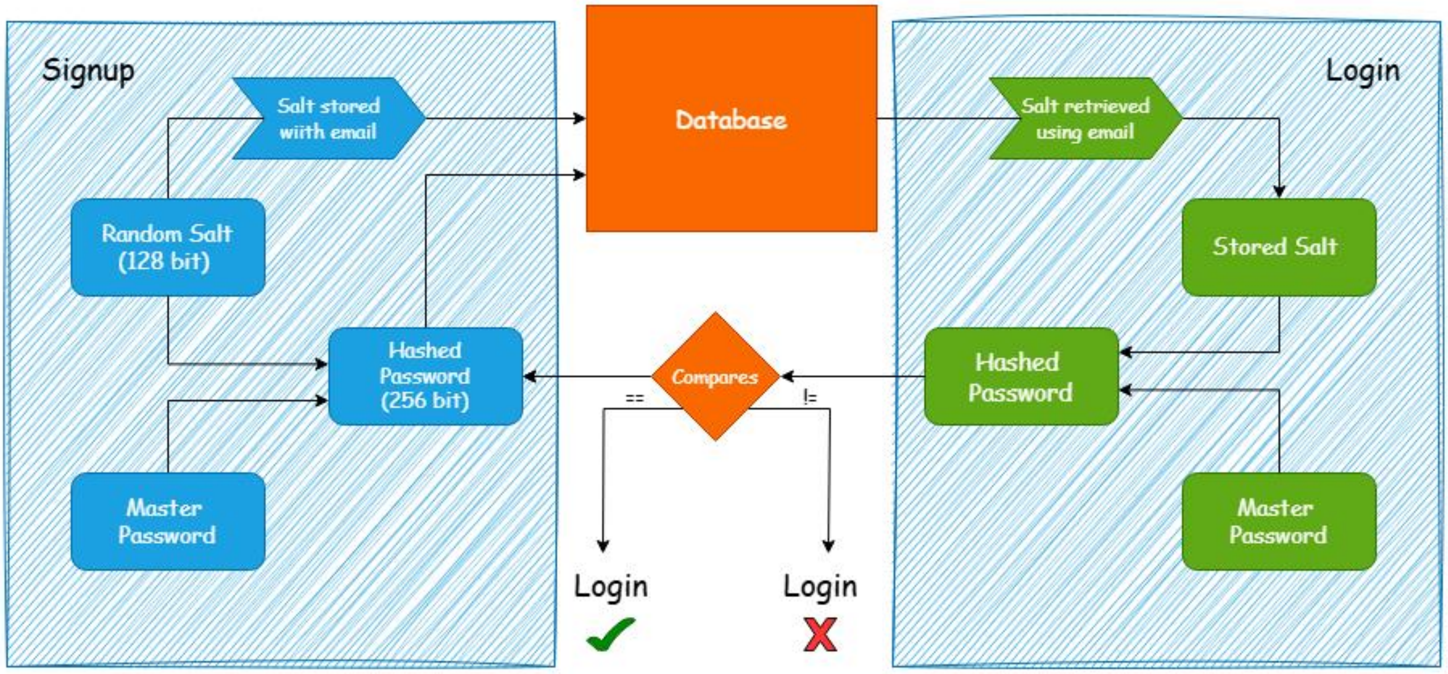
3. Hash the Master Password:

   → The provided master password is combined with the generated salt.
   → A secure hashing algorithm **PBKDF2** is used to derive a 256-bit hashed password.

4. Store Credentials Securely:

   → Store the following in the database:

   - The user's email.
   - The 128 bit generated salt
   - The 256-bit hashed password.



Login –

1. Authenticate User Input:

   → The user provides their email address and master password.

2. Fetch Stored Data:

   → Use the provided email address to retrieve the stored salt and hashed password from the database.

3. Rehash the Password:

   → Combine the user-provided master password with the retrieved salt.
   → Hash this combination using the same algorithm and configuration (PBKDF2) used during signup.

4. Validate Hash:

   → Compare the newly computed hash with the stored hashed password.

   - If they match:

     ✓ Authentication is successful.
     ✓ User is logged in.

   - Otherwise:

     ✗ Reject the log in attempt with a generic error message.

- **Credential encryption :**

→ Credentials are encrypted client-side before being sent to the backend for secure storage in the database.

1. Key Generation:

→ During user login, the system securely stores a unique salt(in DB) alongside the master password(in session memory). These are used to generate a cryptographically secure key using the PBKDF2 hashing algorithm. The salt ensures that each user's key is unique.

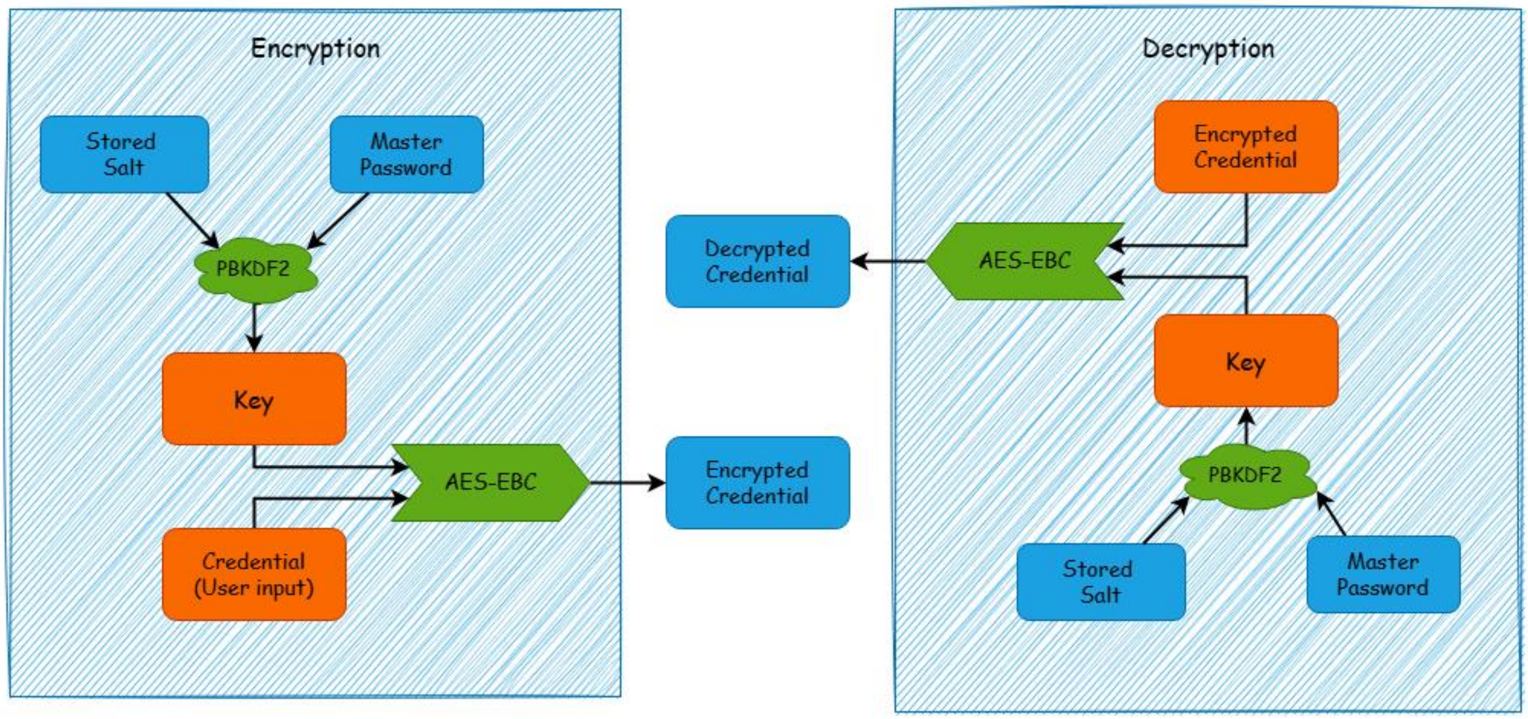
2. Encryption:

→ When the user submits credentials, the secure key is used in combination with the AES-ECB encryption algorithm to encrypt each credential.

3. Decryption:

→ When retrieving stored credentials, the secure key is regenerated using the stored salt and the master password. As the same master password and salt are used, the resulting key is unique to the user and consistent for each session.

→ Using the regenerated key, encrypted credentials are decrypted with AES-ECB and displayed to the user. This ensures that credentials can only be accessed with the correct master password, maintaining zero-knowledge principles.



- **Developers:**

Atharva Dholakia

- Backend (Java Spring Boot)

Keya Patel

- Frontend (React)