# Pass Butler documentation

Pass Butler is a modern password manager which provides a self-hosted cloud solution to synchronize the sensible data between multiple devices securely and privacy compliant. All user data is end-to-end (E2E) encrypted to ensure the data can't even read by the server administrator. Additionally, Pass Butler offers a password item sharing, which means a user can grant or revoke access to desired items to other users (e.g. to share Wi-Fi passwords in a team or family).

In this document, all the involved technologies and cryptographic concepts are documented on technical level.

## Model entities

### User {#user}

| Column | Encrypted | Description |
|---|---|---|
| id | no | The primary key of the entity (UUID) |
| username | no | A unique user identifying string |
| fullName | no | The full name of the user |
| serverComputedAuthenticationHash | no | The Server Computed Authentication Hash |
| masterKeyDerivationInformation | no | A random salt and iteration count to derive the Master Key |
| masterEncryptionKey | yes | The Master Encryption Key |
| itemEncryptionPublicKey | no | The public part of the Item Encryption Key Pair |
| itemEncryptionSecretKey | yes | The private part of the Item Encryption Key Pair |
| settings | yes | The user settings |
| deleted | no | Indicates if the user was deleted |
| modified | no | The Unix timestamp of last modification |
| created | no | The Unix timestamp of creation |

### Item {#item}

| Column | Encrypted | Description |
|---|---|---|
| id | no | The primary key of the entity (UUID) |
| userId | no | The user ID of the creator / owner of the item |
| data | yes | The actual sensible data of the item (username, password, notes, etc.), see Item Data |
| deleted | no | Indicates if the item was deleted |
| modified | no | The Unix timestamp of last modification |
| created | no | The Unix timestamp of creation |

### Item Data {#item-data}

| Column | Encrypted | Description |
|---|---|---|
| title | no | The title of the item |
| username | no | The username of the item |
| password | no | The password of the item |
| url | no | The website URL of the item |
| notes | no | Some notes for the item |
| tags | no | A list of tags for the item |

### Item Authorization {#item-authorization}

| Column | Encrypted | Description |
|---|---|---|
| id | no | The primary key of the entity (UUID) |
| userId | no | The user ID that can use the authorization to access the item |
| itemId | no | The item ID to which was granted access to |
| itemKey | yes | The symmetric key to decrypt the item data |
| readOnly | no | Indicates if the authorization to access the item is read only |
| deleted | no | Indicates if the authorization was revoked |
| modified | no | The Unix timestamp of last modification |
| created | no | The Unix timestamp of creation |

## Security architecture

The Pass Butler security architecture is based on the following principles:

- Usage of strong, modern cryptographic algorithms only
- Usage of well-known cryptographic implementations (Java Security and JavaX Crypto)

- Unit test cryptographic code with official test vectors to ensure correct usage of the cryptographic implementations
- Never ever persist the Master Password nor the Master Key to disk (store it only temporarily in memory for computations)

# Cryptographic technology

The following cryptographic technology is used:

### PBKDF2-SHA256 {#pbkdf2-sha256}

A key derivation algorithm that uses SHA-256. It needs a salt and an iteration count to slow down computing (brute forcing).

### AES-256-GCM {#aes-256-gcm}

A symmetric encryption algorithm with a key length of 256 bit in Galois/Counter mode (GCM) that ensures not only the confidentiality but also the integrity of the data to protect against tampering the encrypted data. A random initialization vector (IV) that must never be reused is needed for operation. The algorithm is fast and suitable for all kinds of data amount.

### RSA-2048-OAEP {#rsa-2048-oaep}

An asymmetric encryption algorithm with a key length of 2048 bit that consists of a public and a private part. The public part allows to encrypt data, the private part allows to decrypt the data. The algorithm is slow and only suitable for small data.

### SecureRandom

The random number generator, which is used for generating cryptographic keys and initialization vectors. Pass Butler uses the default constructor of `java.security.SecureRandom` which utilizes `/dev/urandom` on Unix based systems. It is non-blocking and is capable to provide secure cryptographic keys.

# Cryptographic entities

### Master Password {#master-password}

The one password that protects all other data. It should be long and complex because the complete security architecture relies on it! The *Master Password* is stored in memory only temporary for computing and is overridden afterwards immediately.

### Master Key {#master-key}

The *Master Key* is a symmetric key for AES-256-GCM that is derived from the Master Password with PBKDF2-SHA256 using a random salt and an iteration count stored in `User.masterKeyDerivationInformation`. Like the Master Password, it is derived and stored in memory only temporary for computing and is overridden afterwards immediately.

### Master Encryption Key {#master-encryption-key}

The Master Key could be used directly to encrypt user data, but if the user wants to change its Master Password, all encrypted data would have to be re-encrypted. This is a resource consuming task and takes the risk of data corruption. To avoid this situation, the *Master Encryption Key* is introduced:

It is a symmetric key for AES-256-GCM which is generated once and encrypts sensible data of the user (e.g. the user settings). The *Master Encryption Key* is stored in the `User.masterEncryptionKey` field and is itself encrypted with the Master Key. If the user wants to change its Master Password now, only the same *Master Encryption Key* needs to be re-encrypted.

## Item Key {#item-key}

For a normal password manager, it would be reasonable to encrypt the sensible Item Data of an Item just with the Master Encryption Key. But because an item sharing functionality is featured, a bit more complexity (and keys) must be introduced.

The *Item Key* is a symmetric key for AES-256-GCM which actually encrypts the Item Data.

Every user that have access to an Item has also an appropriate Item Authorization – this contains a user ID, an item ID and the *Item Key* stored in the field `ItemAuthorization.itemKey`. It is itself encrypted with the public part of the Item Encryption Key Pair of the user for whom the Item Authorization is intended.

## Item Encryption Key Pair {#item-encryption-key-pair}

The Item Encryption Key Pair is an asymmetric key pair for RSA-2048-OAEP which consists of a public and a private part:

- the public part (stored in `User.itemEncryptionPublicKey` field) is needed to be able to share an item to another user (the Item Key of the Item desired to share is re-encrypted with it)
- the private part (stored in `User.itemEncryptionSecretKey` field) is needed for the other user to decrypt the encrypted Item Key and access the Item Data of the shared Item. The private part is itself encrypted with the Master Encryption Key.

## Local Computed Authentication Hash {#local-computed-authentication-hash}

The username and this hash are used to authenticate the client to the server. Because the Master Password must **never ever** leave the local client to ensure E2E encryption, only a hash is sent to the server to prove that the client knows it.

This *Local Computed Authentication Hash* is derived from the Master Password with PBKDF2-SHA256 using the username as the salt and 100001 iterations (one iteration more than for Master Key derivation to clearly distinguish between it).

## Server Computed Authentication Hash {#server-computed-authentication-hash}

The Local Computed Authentication Hash sent by the client could be used for direct comparison to the known value. But this idea has a major problem: If an attacker gained access to the database (e.g. through an old backup), he could directly use the included hashes to straight forward authenticate as that users on the server without any more knowledge.

To avoid this “hash-is-the-password“ situation, the received Local Computed Authentication Hash is hashed again with PBKDF2-SHA256 using the random salt and iteration count stored in `User.masterPasswordAuthenticationHash` field. If the calculated hash matches the hash value also stored in this field, the authentication is successful.

## How does the item sharing work?

For example Alice and Bob living in a shared apartment, Bob wants to use the wireless internet but does not know the Wi-Fi password yet. So Alice wants to share the item “Apartment Wi-Fi Password“ to her roommate Bob.

1. Alice enters her Master Password and derives her Master Key
2. Alice decrypt its Master Encryption Key with the Master Key
3. Alice decrypt its private part of her Item Encryption Key Pair with the Master Encryption Key
4. Alice decrypt the Item Key in her Item Authorization of “Apartment Wi-Fi Password“ with the private part of her Item Encryption Key Pair
5. Alice encrypt the Item Key with the public part of the Item Encryption Key Pair of Bob
6. Alice create a new Item Authorization with the item ID of “Apartment Wi-Fi Password“, the user ID of Bob and the re-encrypted Item Key of previous step

Now Bob is able to access the “Apartment Wi-Fi Password“ with the following steps:

1. Bob enters his Master Password and derives his Master Key
2. Bob decrypt its Master Encryption Key with the Master Key
3. Bob decrypt its private part of his Item Encryption Key Pair with the Master Encryption Key
4. Bob decrypt the Item Key in his Item Authorization of “Apartment Wi-Fi Password“ with the private part of his Item Encryption Key Pair
5. Bob decrypt the Item Data of “Apartment Wi-Fi Password“ with the decrypted Item Key and can access the item

Now Bob thankfully can access the wireless network and enjoy his favorite series. And if the Wi-Fi password is changed some time in the future, he automatically sees the updated password in Pass Butler.

## How does the server authentication work?

All normal requests to the server must be authenticated with a valid bearer token (JSON Web Token / JWT). Only the token request must be authenticated with the username and the Local Computed Authentication Hash.

The token authentication tackles two problems:

1. Sending a sensible long-time static secret (the Local Computed Authentication Hash) every single request to the server (the server connection may be TLS encrypted, but this shouldn't be the assumption) – instead only a short-time token is sent, which becomes totally worthless after the short validity period
2. The authentication with username and the Local Computed Authentication Hash is a lot slower because of the resource intense computing of the PBKDF2-SHA256 hashes – the token validity check is very fast and cheap

The token request process works like the following:

1. The client requests a token by sending username and Local Computed Authentication Hash to server
2. The server checks if the requested user is not deleted (`User.deleted == 0`) and calculates the Server Computed Authentication Hash from the received Local Computed Authentication Hash: If the calculated result matches the expected value, the server responds with a new token with a validity of 1 hour

Later requests are only authenticated with the token. If the token is rejected by the server (e.g. because it is expired or just invalid), the server responds with HTTP 401 error, so the client can automatically try to request a new token.

# Synchronization algorithm

All model entities are identified via UUID (no auto increment integer primary keys to avoid conflicts between clients generating the same auto incremented ID on different devices). The up-to-date state of a model entity is determined with the modified timestamp in the `modified` field. Model entities are never really deleted in the database – instead they contain a `deleted` field – this makes the detection of new/deleted state much more simple.

The following steps are executed for all model entities:

1. Load list of local entities
2. Load list of remote entities
3. Detect new local entities and insert them locally
4. Detect new remote entities and insert them remotely
5. Detect modified local entities (according to `modified` field) and update them locally
6. Detect modified remote entities (according to `modified` field) and update them remotely