OMEMO

Release 1.0.2-stable

Tim Henkes (Syndace)

Nov 11, 2022

CONTENTS

A Python implementation of the OMEMO Multi-End Message and Object Encryption protocol.

A complete implementation of XEP-0384 on protocol-level, i.e. more than just the cryptography. python-omemo supports different versions of the specification through so-called backends. A backend for OMEMO in the `urn:xmpp:omemo:2` namespace (the most recent version of the specification) is available in the python-twomemo Python package. A backend for (legacy) OMEMO in the `eu.siacs.conversations.axolotl` namespace is available in the python-oldmemo package. Multiple backends can be loaded and used at the same time, the library manages their coexistence transparently.

CHAPTER

ONE

INSTALLATION

Install the latest release using pip (`pip install OMEMO`) or manually from source by running `pip install .` in the cloned repository.

CHAPTER

TWO

GETTING STARTED

python-omemo only ships the core functionality common to all versions of XEP-0384 and relies on backends to implement the details of each version. Each backend is uniquely identified by the namespace it implements.

2.1 Backend Selection

There are two official backends:

Namespace	Link
`urn:xmpp:omemo:2`	python-twomemo
`eu.siacs.conversations.axolotl`	python-oldmemo

Both backends (and more) can be loaded at the same time and the library will handle compatibility. You can specify backend priority, which will be used to decide which backend to use for encryption in case a recipient device supports multiple loaded backends.

2.2 Public API and Backends

Backends differ in many aspects, from the wire format of the transferred data to the internal cryptographic primitves used. Thus, most parts of the public API take a parameter that specifies the backend to use for the given operation. The core transparently handles all things common to backends and forwards the backend-specific parts to the corresponding backend.

2.3 Trust

python-omemo offers trust management. Since it is not always obvious how trust and JID/device id/identity key belong together, this section gives an overview of the trust concept followed by python-omemo.

Each XMPP account has a pool of identity keys. Each device is assigned one identity key from the pool. Theoretically, this concept allows for one identity key to be assigned to multiple devices, however, the security implications of doing so have not been addressed in the XEP, thus it is not recommended and not supported by this library.

Trust levels are assigned to identity keys, not devices. I.e. devices are not directly trusted, only implicitly by trusting the identity key assigned to them.

The library works with two types of trust levels: custom trust levels and core trust levels. Custom trust levels are assigned to identity keys and can be any Python string. There is no limitation on the number of custom trust levels. Custom trust levels are not used directly by the library for decisions requiring trust (e.g. during message encryption),

instead they are translated to one of the three core trust levels first: Trusted, Distrusted, Undecided. The translation from custom trust levels to core trust levels has to be supplied by implementing the *_evaluate_custom_trust_level()* method.

This trust concept allows for the implementation of trust systems like BTBV, TOFU, simple manual trust or more complex systems.

An example of a BTBV trust system implementation can be found in `examples/btbv_session_manager.py`. If you happen to aim for BTBV support in your client, feel free to use that code as a starting point.

2.4 Setting it Up

With the backends selected and the trust system chosen, the library can be set up.

This is done in three steps:

1. Create a *Storage* implementation
2. Create a *SessionManager* implementation
3. Instantiate your *SessionManager* implementation

2.4.1 `Storage` Implementation

python-omemo uses a simple key-value storage to persist its state. This storage has to be provided to the library by implementing the *Storage* interface. Refer to the API documentation of the *Storage* interface for details.

> **Warning:** It might be tempting to offer a backup/restore flow for the OMEMO data. However, due to the forward secrecy of OMEMO, restoring old data results in broken sessions. It is strongly recommended to not include OMEMO data in backups, and to at most include it in migration flows that make sure that old data can't be restored over newer data.

2.4.2 `SessionManager` Implementation

Create a subclass of *SessionManager*. There are various abstract methods for interaction with XMPP (device lists, bundles etc.) and trust management that you have to fill out to integrate the library with your client/framework. The API documentation of the *SessionManager* class should contain the necessary information.

2.4.3 Instantiate the Library

Finally, instantiate the storage, backends and then the *SessionManager*, which is the class that offers all of the public API for message encryption, decryption, trust and device management etc. To do so, simply call the *create()* method, passing the backend and storage implementations you've prepared. Refer to the API documentation for details on the configuration options accepted by *create()*.

2.5 Migration

Refer to *Migration from Legacy* for information about migrating from pre-stable python-omemo to python-omemo 1.0+. Migrations within stable (1.0+) versions are handled automatically.

CHAPTER

THREE

MIGRATION FROM LEGACY

Due to the multi-backend approach and storage structure of `python-omemo`, migrations are provided by backends rather than the core library. For users of legacy `python-omemo` (i.e. versions before 1.0.0) with `python-omemo-backend-signal`, the python-oldmemo package provides migrations. Please refer to the backend documentation for details.

CHAPTER

FOUR

PACKAGE: OMEMO

4.1 Module: backend

class omemo.backend.**Backend**(*max_num_per_session_skipped_keys=1000*, *max_num_per_message_skipped_keys=None*)

Bases: ABC

The base class for all backends. A backend is a unit providing the functionality of a certain OMEMO version to the core library.

> **Warning:** Make sure to call *__init__()* from your subclass to configure per-message and per-session skipped message key DoS protection thresholds, and respect those thresholds when decrypting key material using *decrypt_key_material()*.

Note: Most methods can raise *StorageException* in addition to those exceptions listed explicitly.

Note: All usages of "identity key" in the public API refer to the public part of the identity key pair in Ed25519 format. Otherwise, "identity key pair" is explicitly used to refer to the full key pair.

Note: For backend implementors: as part of your backend implementation, you are expected to subclass various abstract base classes like *Session*, *Content*, *PlainKeyMaterial*, *EncryptedKeyMaterial* and *KeyExchange*. Whenever any of these abstract base types appears in a method signature of the *Backend* class, what's actually meant is an instance of your respective subclass. This is not correctly expressed through the type system, since I couldn't think of a clean way to do so. Adding generics for every single of these types seemed not worth the effort. For now, the recommended way to deal with this type inaccuray is to assert the types of the affected method parameters, for example:

```
async def store_session(self, session: Session) -> Any:
    assert isinstance(session, MySessionImpl)

    ...
```

Doing so tells mypy how to deal with the situation. These assertions should never fail.

Note: For backend implementors: you can access the identity key pair at any time via *omemo.identity_key_pair.IdentityKeyPair.get()*.

Parameters

- **max_num_per_session_skipped_keys** (*int*) –
- **max_num_per_message_skipped_keys** (*Optional[int]*) –

__init__(*max_num_per_session_skipped_keys=1000*, *max_num_per_message_skipped_keys=None*)

Parameters

- **max_num_per_session_skipped_keys** (int) – The maximum number of skipped message keys to keep around per session. Once the maximum is reached, old message keys are deleted to make space for newer ones. Accessible via *max_num_per_session_skipped_keys*.
- **max_num_per_message_skipped_keys** (Optional[int]) – The maximum number of skipped message keys to accept in a single message. When set to None (the default), this parameter defaults to the per-session maximum (i.e. the value of the max_num_per_session_skipped_keys parameter). This parameter may only be 0 if the per-session maximum is 0, otherwise it must be a number between 1 and the per-session maximum. Accessible via *max_num_per_message_skipped_keys*.

Return type
None

property max_num_per_session_skipped_keys: int

Returns: The maximum number of skipped message keys to keep around per session.

Return type
int

property max_num_per_message_skipped_keys: int

Returns: The maximum number of skipped message keys to accept in a single message.

Return type
int

abstract property namespace: str

Returns: The namespace provided/handled by this backend implementation.

Return type
str

abstract async load_session(*bare_jid*, *device_id*)

Parameters

- **bare_jid** (str) – The bare JID the device belongs to.
- **device_id** (int) – The id of the device.

Return type
Optional[*Session*]

Returns
The session associated with the device, or *None* if such a session does not exist.

> **Warning:** Multiple sessions for the same device can exist in memory, however only one session per device can exist in storage. Which one of the in-memory sessions is persisted in storage is controlled by calling the *store_session()* method.

abstract async store_session(*session*)

Store a session, overwriting any previously stored session for the bare JID and device id this session belongs to.

Parameters
session (*Session*) – The session to store.

Return type
None

> **Warning:** Multiple sessions for the same device can exist in memory, however only one session per device can exist in storage. Which one of the in-memory sessions is persisted in storage is controlled by calling this method.

Return type
None

Parameters
session (Session) –

abstract async build_session_active(*bare_jid*, *device_id*, *bundle*, *plain_key_material*)

Actively build a session.

Parameters

- **bare_jid** (str) – The bare JID the device belongs to.
- **device_id** (int) – The id of the device.
- **bundle** (*Bundle*) – The bundle containing the public key material of the other device required for active session building.
- **plain_key_material** (*PlainKeyMaterial*) – The key material to encrypt for the recipient as part of the initial key exchange/session initiation.

Return type
Tuple[*Session*, *EncryptedKeyMaterial*]

Returns
The newly built session, the encrypted key material and the key exchange information required by the other device to complete the passive part of session building. The *initiation* property of the returned session must return *ACTIVE*. The *key_exchange* property of the returned session must return the information required by the other party to complete its part of the key exchange.

Raises
KeyExchangeFailed – in case of failure related to the key exchange required for session building.

> **Warning:** This method may be called for a device which already has a session. In that case, the original session must remain in storage and must remain loadable via *load_session()*. Only upon calling *store_session()*, the old session must be overwritten with the new one. In summary, multiple sessions for the same device can exist in memory, while only one session per device can exist in storage, which can be controlled using the *store_session()* method.

abstract async build_session_passive(*bare_jid*, *device_id*, *key_exchange*, *encrypted_key_material*)

Passively build a session.

Parameters

- **bare_jid** (str) – The bare JID the device belongs to.
- **device_id** (int) – The id of the device.
- **key_exchange** (*KeyExchange*) – Key exchange information for the passive session building.
- **encrypted_key_material** (*EncryptedKeyMaterial*) – The key material to decrypt as part of the initial key exchange/session initiation.

Return type
Tuple[*Session*, *PlainKeyMaterial*]

Returns
The newly built session and the decrypted key material. Note that the pre key used to initiate this session must somehow be associated with the session, such that *hide_pre_key()* and *delete_pre_key()* can work.

Raises

- ***KeyExchangeFailed*** – in case of failure related to the key exchange required for session building.
- ***DecryptionFailed*** – in case of backend-specific failures during decryption of the initial message.

> **Warning:** This method may be called for a device which already has a session. In that case, the original session must remain in storage and must remain loadable via *load_session()*. Only upon calling *store_session()*, the old session must be overwritten with the new one. In summary, multiple sessions for the same device can exist in memory, while only one session per device can exist in storage, which can be controlled using the *store_session()* method.

abstract async encrypt_plaintext(*plaintext*)

Encrypt some plaintext symmetrically.

Parameters
plaintext (bytes) – The plaintext to encrypt symmetrically.

Return type
Tuple[*Content*, *PlainKeyMaterial*]

Returns
The encrypted plaintext aka content, as well as the key material needed to decrypt it.

abstract async encrypt_empty()

Encrypt an empty message for the sole purpose of session manangement/ratchet forwarding/key material transportation.

Return type
Tuple[*Content*, *PlainKeyMaterial*]

Returns
The symmetrically encrypted empty content, and the key material needed to decrypt it.

abstract async encrypt_key_material(*session*, *plain_key_material*)

Encrypt some key material asymmetrically using the session.

Parameters

- **session** (*Session*) – The session to encrypt the key material with.
- **plain_key_material** (*PlainKeyMaterial*) – The key material to encrypt asymmetrically for each recipient.

Return type
EncryptedKeyMaterial

Returns
The encrypted key material.

abstract async decrypt_plaintext(*content*, *plain_key_material*)

Decrypt some symmetrically encrypted plaintext.

Parameters

- **content** (*Content*) – The content to decrypt. Not empty, i.e. `Content.empty` will return `False`.
- **plain_key_material** (*PlainKeyMaterial*) – The key material to decrypt with.

Return type
bytes

Returns
The decrypted plaintext.

Raises
DecryptionFailed – in case of backend-specific failures during decryption.

abstract async decrypt_key_material(*session*, *encrypted_key_material*)

Decrypt some key material asymmetrically using the session.

Parameters

- **session** (*Session*) – The session to decrypt the key material with.
- **encrypted_key_material** (*EncryptedKeyMaterial*) – The encrypted key material.

Return type
PlainKeyMaterial

Returns
The decrypted key material

Raises

- ***TooManySkippedMessageKeys*** – if the number of message keys skipped by this message exceeds the upper limit enforced by *max_num_per_message_skipped_keys*.
- ***DecryptionFailed*** – in case of backend-specific failures during decryption.

Warning: Make sure to respect the values of *max_num_per_session_skipped_keys* and *max_num_per_message_skipped_keys*.

Note: When the maximum number of skipped message keys for this session, given by *max_num_per_session_skipped_keys*, is exceeded, old skipped message keys are deleted to make space for new ones.

abstract async signed_pre_key_age()

Return type
int

Returns
The age of the signed pre key, i.e. the time elapsed since it was last rotated, in seconds.

abstract async rotate_signed_pre_key()

Rotate the signed pre key. Keep the old signed pre key around for one additional rotation period, i.e. until this method is called again.

Return type
None

abstract async hide_pre_key(*session*)

Hide a pre key from the bundle returned by *get_bundle()* and pre key count returned by *get_num_visible_pre_keys()*, but keep the pre key for cryptographic operations.

Parameters
session (*Session*) – A session that was passively built using *build_session_passive()*. Use this session to identity the pre key to hide.

Return type
bool

Returns
Whether the pre key was hidden. If the pre key doesn't exist (e.g. because it has already been deleted), or was already hidden, do not throw an exception, but return *False* instead.

abstract async delete_pre_key(*session*)

Delete a pre key.

Parameters
session (*Session*) – A session that was passively built using *build_session_passive()*. Use this session to identity the pre key to delete.

Return type
bool

Returns
Whether the pre key was deleted. If the pre key doesn't exist (e.g. because it has already been deleted), do not throw an exception, but return *False* instead.

abstract async delete_hidden_pre_keys()

Delete all pre keys that were previously hidden using *hide_pre_key()*.

Return type
None

abstract async get_num_visible_pre_keys()

Return type
int

Returns
The number of visible pre keys available. The number returned here should match the number of pre keys included in the bundle returned by *get_bundle()*.

abstract async generate_pre_keys(*num_pre_keys*)

Generate and store pre keys.

Parameters
num_pre_keys (int) – The number of pre keys to generate.

Return type
None

abstract async get_bundle(*bare_jid*, *device_id*)

Parameters

- **bare_jid** (str) – The bare JID of this XMPP account, to be included in the bundle.
- **device_id** (int) – The id of this device, to be included in the bundle.

Return type
Bundle

Returns
The bundle containing public information about the cryptographic state of this backend.

> **Warning:** Do not include pre keys hidden by *hide_pre_key()* in the bundle!

abstract async purge()

Remove all data related to this backend from the storage.

Return type
None

abstract async purge_bare_jid(*bare_jid*)

Delete all data corresponding to an XMPP account.

Parameters
bare_jid (str) – Delete all data corresponding to this bare JID.

Return type
None

exception omemo.backend.**BackendException**

Bases: *OMEMOException*

Parent type for all exceptions specific to *Backend*.

exception omemo.backend.**DecryptionFailed**

Bases: *BackendException*

Raised by various methods of *Backend* in case of backend-specific failures during decryption.

exception omemo.backend.**KeyExchangeFailed**

Bases: *BackendException*

Raised by *Backend.build_session_active()* and *Backend.build_session_passive()* in case of an error during the processing of a key exchange for session building. Known error conditions are:

- The bundle does not contain and pre keys (active session building)
- The signature of the signed pre key could not be verified (active session building)
- An unkown (signed) pre key was referred to (passive session building)

Additional backend-specific error conditions might exist.

exception omemo.backend.**TooManySkippedMessageKeys**

Bases: *BackendException*

Raised by *Backend.decrypt_key_material()* if a message skips more message keys than allowed.

4.2 Module: bundle

class omemo.bundle.**Bundle**

Bases: ABC

The bundle of a device, containing the cryptographic information required for active session building.

Note: All usages of "identity key" in the public API refer to the public part of the identity key pair in Ed25519 format.

---

abstract property namespace: str

Return type
str

abstract property bare_jid: str

Return type
str

abstract property device_id: int

Return type
int

abstract property identity_key: bytes

Return type
bytes

abstract __eq__(*other*)

Check an object for equality with this Bundle instance.

Parameters
other (object) – The object to compare to this instance.

Return type
bool

Returns
Whether the other object is a bundle with the same contents as this instance.

Note: The order in which pre keys are included in the bundles does not matter.

`abstract __hash__()`

Hash this instance in a manner that is consistent with *__eq__()*.

Return type
`int`

Returns
An integer value representing this instance.

4.3 Module: identity_key_pair

`class omemo.identity_key_pair.IdentityKeyPair`

Bases: `ABC`

The identity key pair associated to this device, shared by all backends.

There are following requirements for the identity key pair:

- It must be able to create and verify Ed25519-compatible signatures.
- It must be able to perform X25519-compatible Diffie-Hellman key agreements.

There are at least two different kinds of key pairs that can fulfill these requirements: Ed25519 key pairs and Curve25519 key pairs. The birational equivalence of both curves can be used to "convert" one pair to the other.

Both types of key pairs share the same private key, however instead of a private key, a seed can be used which the private key is derived from using SHA-512. This is standard practice for Ed25519, where the other 32 bytes of the SHA-512 seed hash are used as a nonce during signing. If a new key pair has to be generated, this implementation generates a seed.

Note: This is the only actual cryptographic functionality offered by the core library. Everything else is backend-specific.

`LOG_TAG = 'omemo.core.identity_key_pair'`

`async static get(storage)`

Get the identity key pair.

Parameters
storage (*Storage*) – The storage for all OMEMO-related data.

Return type
IdentityKeyPair

Returns
The identity key pair, which has either been loaded from storage or newly generated.

Note: There is only one identity key pair for storage instance. All instances of this class refer to the same storage locations, thus the same data.

abstract property is_seed: bool

Returns: Whether this is a *IdentityKeyPairSeed*.

Return type
bool

abstract property is_priv: bool

Returns: Whether this is a *IdentityKeyPairPriv*.

Return type
bool

abstract as_priv()

Return type
IdentityKeyPairPriv

Returns
An *IdentityKeyPairPriv* derived from this instance (if necessary).

abstract property identity_key: Ed25519Pub

Returns: The public part of this identity key pair, in Ed25519 format.

Return type
bytes

class omemo.identity_key_pair.**IdentityKeyPairPriv**(*priv*)

Bases: *IdentityKeyPair*

An *IdentityKeyPair* represented by a private key.

Parameters
priv (*Priv*) –

__init__(*priv*)

Parameters
priv (bytes) – The Curve25519/Ed25519 private key.

Return type
None

property is_seed: bool

Returns: Whether this is a *IdentityKeyPairSeed*.

Return type
bool

property is_priv: bool

Returns: Whether this is a *IdentityKeyPairPriv*.

Return type
bool

as_priv()

Return type
IdentityKeyPairPriv

Returns
An *IdentityKeyPairPriv* derived from this instance (if necessary).

property identity_key: Ed25519Pub

Returns: The public part of this identity key pair, in Ed25519 format.

Return type
bytes

property priv: Priv

Returns: The Curve25519/Ed25519 private key.

Return type
bytes

class omemo.identity_key_pair.**IdentityKeyPairSeed**(*seed*)

Bases: *IdentityKeyPair*

An *IdentityKeyPair* represented by a seed.

Parameters
seed (*Seed*) –

__init__(*seed*)

Parameters
seed (bytes) – The Curve25519/Ed25519 seed.

Return type
None

property is_seed: bool

Returns: Whether this is a *IdentityKeyPairSeed*.

Return type
bool

property is_priv: bool

Returns: Whether this is a *IdentityKeyPairPriv*.

Return type
bool

as_priv()

Return type
IdentityKeyPairPriv

Returns
An *IdentityKeyPairPriv* derived from this instance (if necessary).

property identity_key: Ed25519Pub

Returns: The public part of this identity key pair, in Ed25519 format.

Return type
bytes

property seed: Seed

Returns: The Curve25519/Ed25519 seed.

Return type
bytes

4.4 Module: message

class omemo.message.**Content**

Bases: ABC

The encrypted content of an OMEMO-encrypted message. Contains for example the ciphertext, but can contain other backend-specific data that is shared between all recipients.

abstract property empty: bool

Returns: Whether this instance corresponds to an empty OMEMO message purely used for protocol stability reasons.

Return type
bool

class omemo.message.**EncryptedKeyMaterial**

Bases: ABC

Encrypted key material. When decrypted, the key material can in turn be used to decrypt the content. One collection of key material is included in an OMEMO-encrypted message per recipient. Defails are backend-specific.

abstract property bare_jid: str

Return type
str

abstract property device_id: int

Return type
int

class omemo.message.**KeyExchange**

Bases: ABC

Key exchange information, generated by the active part of the session building process, then transferred to and consumed by the passive part of the session building process. Details are backend-specific.

abstract property identity_key: bytes

Return type
bytes

abstract builds_same_session(*other*)

Parameters
other (*KeyExchange*) – The other key exchange instance to compare to this instance.

Return type
bool

Returns
Whether the key exchange information stored in this instance and the key exchange information stored in the other instance would build the same session.

class omemo.message.**Message**(*namespace*, *bare_jid*, *device_id*, *content*, *keys*)

Bases: tuple

Simple structure representing an OMEMO-encrypted message.

Parameters

- **namespace** (*str*) –
- **bare_jid** (*str*) –
- **device_id** (*int*) –
- **content** (Content) –
- **keys** (*FrozenSet[Tuple[EncryptedKeyMaterial, Optional[KeyExchange]]]*) –

property namespace

Alias for field number 0

property bare_jid

Alias for field number 1

property device_id

Alias for field number 2

property content

Alias for field number 3

property keys

Alias for field number 4

class omemo.message.**PlainKeyMaterial**

Bases: ABC

Key material which be used to decrypt the content. Defails are backend-specific.

4.5 Module: session

class omemo.session.**Initiation**(*value*)

Bases: Enum

Enumeration identifying whether a session was built through active or passive session initiation.

ACTIVE: str = 'ACTIVE'

PASSIVE: str = 'PASSIVE'

class omemo.session.**Session**

Bases: ABC

Class representing an OMEMO session. Used to encrypt/decrypt key material for/from a single recipient/sender device in a perfectly forwared secure manner.

> **Warning:** Changes to a session may only be persisted when *store_session()* is called.

> **Warning:** Multiple sessions for the same device can exist in memory, however only one session per device can exist in storage. Which one of the in-memory sessions is persisted in storage is controlled by calling the *store_session()* method.

Note: The API of the *Session* class was intentionally kept thin. All "complex" interactions with session objects happen via methods of *Backend*. This allows backend implementors to have the *Session* class be a simple "stupid" data holding structure type, while all of the more complex logic is located in the implementation of the *Backend* class itself. Backend implementations are obviously free to implement logic on their respective *Session* implementations and forward calls to them from the *Backend* methods.

abstract property namespace: str

Return type
str

abstract property bare_jid: str

Return type
str

abstract property device_id: int

Return type
int

abstract property initiation: *Initiation*

Returns: Whether this session was actively initiated or passively.

Return type
Initiation

abstract property confirmed: bool

In case this session was built through active session initiation, this flag should indicate whether the session initiation has been "confirmed", i.e. at least one message was received and decrypted using this session.

Return type
bool

abstract property key_exchange: *KeyExchange*

Either the key exchange information received during passive session building, or the key exchange information created as part of active session building. The key exchange information is needed by the protocol for stability reasons, to make sure that all sides can build the session, even if messages are lost or received out of order.

Return type
KeyExchange

Returns
The key exchange information associated with this session.

abstract property receiving_chain_length: Optional[int]

Returns: The length of the receiving chain, if it exists, used for own staleness detection.

Return type
Optional[int]

abstract property sending_chain_length: int

Returns: The length of the sending chain, used for staleness detection of other devices.

Return type
int

4.6 Module: session_manager

exception omemo.session_manager.**SessionManagerException**

Bases: *OMEMOException*

Parent type for all exceptions specific to *SessionManager*.

exception omemo.session_manager.**TrustDecisionFailed**

Bases: *SessionManagerException*

Raised by *SessionManager._make_trust_decision()* if the trust decisions that were queried somehow failed. Indirectly raised by the encryption flow.

exception omemo.session_manager.**StillUndecided**

Bases: *SessionManagerException*

Raised by *SessionManager.encrypt()* in case there are still undecided devices after a trust decision was queried via *SessionManager._make_trust_decision()*.

exception omemo.session_manager.**NoEligibleDevices**(*bare_jids*, **args*)

Bases: *SessionManagerException*

Raised by *SessionManager.encrypt()* in case none of the devices of one or more recipient are eligible for encryption, for example due to distrust or bundle downloading failures.

Parameters

- **bare_jids** (*FrozenSet[str]*) –
- **args** (*object*) –

Return type
None

__init__(*bare_jids*, **args*)

Parameters

- **bare_jids** (FrozenSet[str]) – The JIDs whose devices were not eligible. Accessible as an attribute of the returned instance.
- **args** (*object*) –

Return type
None

exception omemo.session_manager.**MessageNotForUs**

Bases: *SessionManagerException*

Raised by *SessionManager.decrypt()* in case the message to decrypt does not seem to be encrypting for this device.

exception omemo.session_manager.**SenderNotFound**

Bases: *SessionManagerException*

Raised by *SessionManager.decrypt()* in case the usual public information of the sending device could not be downloaded.

exception omemo.session_manager.**SenderDistrusted**

Bases: *SessionManagerException*

Raised by *SessionManager.decrypt()* in case the sending device is explicitly distrusted.

exception omemo.session_manager.**NoSession**

Bases: *SessionManagerException*

Raised by *SessionManager.decrypt()* in case there is no session with the sending device, and a new session can't be built either.

exception omemo.session_manager.**PublicDataInconsistency**

Bases: *SessionManagerException*

Raised by *SessionManager.decrypt()* in case inconsistencies were found in the public data of the sending device.

exception omemo.session_manager.**UnknownTrustLevel**

Bases: *SessionManagerException*

Raised by *SessionManager._evaluate_custom_trust_level()* if the custom trust level name to evaluate is unknown. Indirectly raised by the encryption and decryption flows.

exception omemo.session_manager.**UnknownNamespace**

Bases: *SessionManagerException*

Raised by various methods of *SessionManager*, in case the namespace to perform an operation under is not known or the corresponding backend is not currently loaded.

exception omemo.session_manager.**XMPPInteractionFailed**

Bases: *SessionManagerException*

Parent type for all exceptions related to network/XMPP interactions.

exception omemo.session_manager.**BundleUploadFailed**

Bases: *XMPPInteractionFailed*

Raised by *SessionManager._upload_bundle()*, and indirectly by various methods of *SessionManager*.

exception omemo.session_manager.**BundleDownloadFailed**

Bases: *XMPPInteractionFailed*

Raised by *SessionManager._download_bundle()*, and indirectly by various methods of *SessionManager*.

exception omemo.session_manager.**BundleNotFound**

Bases: *XMPPInteractionFailed*

Raised by *SessionManager._download_bundle()*, and indirectly by various methods of *SessionManager*.

exception omemo.session_manager.**BundleDeletionFailed**

Bases: *XMPPInteractionFailed*

Raised by *SessionManager._delete_bundle()*, and indirectly by *SessionManager.purge_backend()*.

exception omemo.session_manager.**DeviceListUploadFailed**

Bases: *XMPPInteractionFailed*

Raised by *SessionManager._upload_device_list()*, and indirectly by various methods of *SessionManager*.

exception omemo.session_manager.**DeviceListDownloadFailed**

Bases: *XMPPInteractionFailed*

Raised by *SessionManager._download_device_list()*, and indirectly by various methods of *SessionManager*.

exception omemo.session_manager.**MessageSendingFailed**

Bases: *XMPPInteractionFailed*

Raised by *SessionManager._send_message()*, and indirectly by various methods of *SessionManager*.

class omemo.session_manager.**SessionManager**

Bases: ABC

The core of python-omemo. Manages your own key material and bundle, device lists, sessions with other users and much more, all while being flexibly usable with different backends and transparenlty maintaining a level of compatibility between the backends that allows you to maintain a single identity throughout all of them. Easy APIs are provided to handle common use-cases of OMEMO-enabled XMPP clients, with one of the primary goals being strict type safety.

Note: Most methods can raise *StorageException* in addition to those exceptions listed explicitly.

Note: All parameters are treated as immutable unless explicitly noted otherwise.

Note: All usages of "identity key" in the public API refer to the public part of the identity key pair in Ed25519 format. Otherwise, "identity key pair" is explicitly used to refer to the full key pair.

Note: The library was designed for use as part of an XMPP library/client. The API is shaped for XMPP and comments/documentation contain references to XEPs and other XMPP-specific nomenclature. However, the library can be used with any economy that provides similar functionality.

DEVICE_ID_MIN = 1

DEVICE_ID_MAX = 2147483647

STALENESS_MAGIC_NUMBER = 53

LOG_TAG = 'omemo.core'

async classmethod create(*backends*, *storage*, *own_bare_jid*, *initial_own_label*, *undecided_trust_level_name*, *signed_pre_key_rotation_period=604800*, *pre_key_refill_threshold=99*, *async_framework=AsyncFramework.ASYNCIO*)

Load or create OMEMO backends. This method takes care of everything regarding the initialization of OMEMO: generating a unique device id, uploading the bundle and adding the new device to the device list. While doing so, it makes sure that all backends share the same identity key, so that a certain level of compatibility between the backends can be achieved. If a backend was created before, this method loads the backend from the storage instead of creating it.

Parameters

- **backends** (List[*Backend*]) – The list of backends to use.
- **storage** (*Storage*) – The storage for all OMEMO-related data.
- **own_bare_jid** (str) – The own bare JID of the account this device belongs to.

- **initial_own_label** (Optional[str]) – The initial (optional) label to assign to this device if supported by any of the backends.
- **undecided_trust_level_name** (str) – The name of the custom trust level to initialize the trust level with when a new device is first encoutered. *_evaluate_custom_trust_level()* should evaluate this custom trust level to *UNDECIDED*.
- **signed_pre_key_rotation_period** (int) – The rotation period for the signed pre key, in seconds. The rotation period is recommended to be between one week (the default) and one month.
- **pre_key_refill_threshold** (int) – The number of pre keys that triggers a refill to 100. Defaults to 99, which means that each pre key gets replaced with a new one right away. The threshold can not be configured to lower than 25.
- **async_framework** (*AsyncFramework*) – The framework to use to create asynchronous tasks and perform asynchronous waiting. Defaults to asyncio, since it's part of the standard library. Make sure the respective framework is installed when using something other than asyncio.

Return type
TypeVar(SessionManagerTypeT, bound= SessionManager)

Returns
A configured instance of *SessionManager*, with all backends loaded, bundles published and device lists adjusted.

Raises

- ***BundleUploadFailed*** – if a bundle upload failed. Forwarded from *_upload_bundle()*.
- ***BundleDeletionFailed*** – if a bundle deletion failed. Forwarded from *_delete_bundle()*.
- ***DeviceListUploadFailed*** – if a device list upload failed. Forwarded from *_upload_device_list()*.
- ***DeviceListDownloadFailed*** – if a device list download failed. Forwarded from *_download_device_list()*.

Warning: The library starts in history synchronization mode. Call *after_history_sync()* to return to normal operation. Refer to the documentation of *before_history_sync()* and *after_history_sync()* for details.

Warning: The library takes care of keeping online data in sync. That means, if the library is loaded without a backend that was loaded before, it will remove all online data related to the missing backend and as much of the offline data as possible (refer to *purge_backend()* for details).

Note: This method takes care of leaving the device lists in a consistent state. To do so, backends are "initialized" one after the other. For each backend, the device list is updated as the very last step, after everything else that could fail is done. This ensures that either all data is consistent or the device list does not yet list the inconsistent device. If the creation of one backend succeeds, the data is persisted in the

storage before the next backend is created. This guarantees that even if the next backend creation fails, the data is not lost and will be loaded from the storage when calling this method again.

Note: The order of the backends can optionally be used by *encrypt()* as the order of priority, in case a recipient device supports multiple backends. Refer to the documentation of *encrypt()* for details.

async purge_backend(*namespace*)

Purge a backend, removing both the online data (bundle, device list entry) and the offline data that belongs to this backend. Note that the backend-specific offline data can only be purged if the respective backend is currently loaded. This backend-specific removal can be triggered manually at any time by calling the *purge()* method of the respecfive backend. If the backend to purge is currently loaded, the method will unload it.

Parameters
namespace (str) – The XML namespace managed by the backend to purge.

Raises

- ***BundleDeletionFailed*** – if a bundle deletion failed. Forwarded from *_delete_bundle()*.
- ***DeviceListUploadFailed*** – if a device list upload failed. Forwarded from *_upload_device_list()*.
- ***DeviceListDownloadFailed*** – if a device list download failed. Forwarded from *_download_device_list()*.

Return type
None

Warning: Make sure to unsubscribe from updates to all device lists before calling this method.

Note: If the backend-specific offline data is not purged, the backend can be loaded again at a later point and the online data can be restored. This is what happens when a backend that was previously loaded is omitted from *create()*.

Return type
None

Parameters
namespace (*str*) –

async purge_bare_jid(*bare_jid*)

Delete all data corresponding to an XMPP account. This includes the device list, trust information and all sessions across all loaded backends. The backend-specific data can be removed at any time by calling the *purge_bare_jid()* method of the respective backend.

Parameters
bare_jid (str) – Delete all data corresponding to this bare JID.

Return type
None

async ensure_data_consistency()

Ensure that the online data for all loaded backends is consistent with the offline data. Refreshes device lists of all backends while making sure that this device is included in all of them. Downloads the bundle for each backend, compares it with the local bundle contents, and uploads the local bundle if necessary.

Raises

- *DeviceListDownloadFailed* – if a device list download failed. Forwarded from *_download_device_list()*.
- *DeviceListUploadFailed* – if a device list upload failed. Forwarded from *update_device_list()*.
- *BundleUploadFailed* – if a bundle upload failed. Forwarded from *_upload_bundle()*.

Return type
None

Note: This method is not called automatically by the library, since under normal working conditions, online and offline data should never desync. However, if clients can spare the network traffic, it is recommended to call this method e.g. once after starting the library and possibly in other scenarios/at regular intervals too.

Return type
None

abstract async static _upload_bundle(*bundle*)

Upload the bundle corresponding to this device, overwriting any previously published bundle data.

Parameters
bundle (*Bundle*) – The bundle to publish.

Raises

- *UnknownNamespace* – if the namespace is unknown.
- *BundleUploadFailed* – if the upload failed. Feel free to raise a subclass instead.

Return type
None

Note: This method is called from *create()*, before *create()* has returned the instance. Thus, modifications to the object (`self`, in case of subclasses) may not have happened when this method is called.

Note: This method must be able to handle at least the namespaces of all loaded backends.

Return type
None

Parameters
bundle (*Bundle*) –

abstract async static _download_bundle(*namespace*, *bare_jid*, *device_id*)

Download the bundle corresponding to a specific device.

Parameters

- **namespace** (str) – The XML namespace to execute this operation under.
- **bare_jid** (str) – The bare JID the device belongs to.
- **device_id** (int) – The id of the device.

Return type
Bundle

Returns
The bundle.

Raises

- ***UnknownNamespace*** – if the namespace is unknown.
- ***BundleDownloadFailed*** – if the download failed. Feel free to raise a subclass instead. Only raise this on a technical bundle download failure. If the bundle just doesn't exist, raise *BundleNotFound* instead.
- ***BundleNotFound*** – if the bundle doesn't exist.

Note: This method is called from *create()*, before *create()* has returned the instance. Thus, modifications to the object (self, in case of subclasses) may not have happened when this method is called.

Note: This method must be able to handle at least the namespaces of all loaded backends.

---

abstract async static _delete_bundle(*namespace*, *device_id*)

Delete the bundle corresponding to this device.

Parameters

- **namespace** (str) – The XML namespace to execute this operation under.
- **device_id** (int) – The id of this device.

Raises

- ***UnknownNamespace*** – if the namespace is unknown.
- ***BundleDeletionFailed*** – if the deletion failed. Feel free to raise a subclass instead. Only raise this on a technical bundle deletion failure. If the bundle just doesn't exist, don't raise.

Return type
None

Note: This method is called from *create()*, before *create()* has returned the instance. Thus, modifications to the object (self, in case of subclasses) may not have happened when this method is called.

Note: This method must be able to handle at least the namespaces of all loaded backends. In case of backend purging via *purge_backend()*, the corresponding namespace must be supported even if the backend

is not currently loaded.

Return type
None

Parameters

- **namespace** (*str*) –
- **device_id** (*int*) –

abstract async static _upload_device_list(*namespace*, *device_list*)

Upload the device list for this XMPP account.

Parameters

- **namespace** (str) – The XML namespace to execute this operation under.
- **device_list** (Dict[int, Optional[str]]) – The device list to upload. Mapping from device id to optional label.

Raises

- ***UnknownNamespace*** – if the namespace is unknown.
- ***DeviceListUploadFailed*** – if the upload failed. Feel free to raise a subclass instead.

Return type
None

Note: This method is called from *create()*, before *create()* has returned the instance. Thus, modifications to the object (self, in case of subclasses) may not have happened when this method is called.

Note: This method must be able to handle at least the namespaces of all loaded backends.

Return type
None

Parameters

- **namespace** (*str*) –
- **device_list** (*Dict[int, Optional[str]]*) –

abstract async static _download_device_list(*namespace*, *bare_jid*)

Download the device list of a specific XMPP account.

Parameters

- **namespace** (str) – The XML namespace to execute this operation under.
- **bare_jid** (str) – The bare JID of the XMPP account.

Return type
Dict[int, Optional[str]]

Returns
The device list as a dictionary, mapping the device ids to their optional label.

Raises

- *UnknownNamespace* – if the namespace is unknown.
- *DeviceListDownloadFailed* – if the download failed. Feel free to raise a subclass instead. Only raise this on a technical device list download failure. If the device list just doesn't exist, return and empty list instead.

Note: This method is called from *create()*, before *create()* has returned the instance. Thus, modifications to the object (`self`, in case of subclasses) may not have happened when this method is called.

Note: This method must be able to handle at least the namespaces of all loaded backends.

---

abstract async _evaluate_custom_trust_level(*device*)

Evaluate a custom trust level to one of the three core trust levels:

- *TRUSTED*: This device is trusted, encryption/decryption of messages to/from it is allowed.
- *DISTRUSTED*: This device is explicitly *not* trusted, do not encrypt/decrypt messages to/from it.
- *UNDECIDED*: A trust decision is yet to be made. It is not clear whether it is okay to encrypt messages to it, however decrypting messages from it is allowed.

Parameters
device (*DeviceInformation*) – Information about the device, including the custom trust level name to translate.

Return type
TrustLevel

Returns
The core trust level corresponding to the custom trust level.

Raises
UnknownTrustLevel – if a custom trust level with this name is not known. Feel free to raise a subclass instead.

abstract async _make_trust_decision(*undecided*, *identifier*)

Make a trust decision on a set of undecided identity keys. The trust decisions are expected to be persisted by calling *set_trust()*.

Parameters

- **undecided** (`FrozenSet[`*DeviceInformation*`]`) – A set of devices that require trust decisions.
- **identifier** (`Optional[str]`) – A piece of application-specific information that callers can pass to *encrypt()*, which is then forwarded here unaltered. This can be used, for example, by instant messaging clients, to identify the chat tab which triggered the call to *encrypt()* and subsequently this call to *_make_trust_decision()*.

Raises
TrustDecisionFailed – if for any reason the trust decision failed/could not be completed. Feel free to raise a subclass instead.

Return type
None

Note: This is called when the encryption needs to know whether it is allowed to encrypt for these devices or not. When this method returns, all previously undecided trust levels should have been replaced by calling *set_trust()* with a different trust level. If they are not replaced or still evaluate to the undecided trust level after the call, the encryption will fail with an exception. See *encrypt()* for details.

Return type
None

Parameters

- **undecided** (*FrozenSet[DeviceInformation]*) –
- **identifier** (*Optional[str]*) –

abstract async static _send_message(*message*, *bare_jid*)

Send an OMEMO-encrypted message. This is required for various automated behaviours to improve the overall stability of the protocol, for example:

- Automatic handshake completion, by responding to incoming key exchanges.
- Automatic heartbeat messages to forward the ratchet if many messages were received without a (manual) response, to assure forward secrecy (aka staleness prevention). The number of messages required to trigger this behaviour is hardcoded in *STALENESS_MAGIC_NUMBER*.
- Automatic session initiation if an encrypted message is received but no session exists for that device.
- Backend-dependent session healing mechanisms.
- Backend-dependent empty messages to notify other devices about potentially "broken" sessions.

Note that messages sent here do not contain any content, they just transport key material.

Parameters

- **message** (*Message*) – The message to send.
- **bare_jid** (str) – The bare JID to send the message to.

Raises

- ***UnknownNamespace*** – if the namespace is unknown.
- ***MessageSendingFailed*** – if for any reason the message could not be sent. Feel free to raise a subclass instead.

Return type
None

async update_device_list(*namespace*, *bare_jid*, *device_list*)

Update the device list of a specific bare JID, e.g. after receiving an update for the XMPP account from PEP.

Parameters

- **namespace** (str) – The XML namespace to execute this operation under.
- **bare_jid** (str) – The bare JID of the XMPP account.
- **device_list** (Dict[int, Optional[str]]) – The updated device list. Mapping from device id to optional label.

Raises

- *UnknownNamespace* – if the backend to handle the message is not currently loaded.
- *DeviceListUploadFailed* – if a device list upload failed. An upload can happen if the device list update is for the own bare JID and does not include the own device. Forwarded from *_upload_device_list()*.

Return type
None

async refresh_device_list(*namespace*, *bare_jid*)

Manually trigger the refresh of a device list.

Parameters

- **namespace** (str) – The XML namespace to execute this operation under.
- **bare_jid** (str) – The bare JID of the XMPP account.

Raises

- *UnknownNamespace* – if the namespace is unknown.
- *DeviceListDownloadFailed* – if the device list download failed. Forwarded from *_download_device_list()*.
- *DeviceListUploadFailed* – if a device list upload failed. An upload can happen if the device list update is for the own bare JID and does not include the own device. Forwarded from *update_device_list()*.

Return type
None

async set_trust(*bare_jid*, *identity_key*, *trust_level_name*)

Set the trust level for an identity key.

Parameters

- **bare_jid** (str) – The bare JID of the XMPP account this identity key belongs to.
- **identity_key** (bytes) – The identity key.
- **trust_level_name** (str) – The custom trust level to set for the identity key.

Return type
None

async replace_sessions(*device*)

Manually replace all sessions for a device. Can be used if sessions are suspected to be broken. This method automatically notifies the other end about the new sessions, so that hopefully no messages are lost.

Parameters
device (*DeviceInformation*) – The device whose sessions to replace.

Return type
Dict[str, *OMEMOException*]

Returns
Information about exceptions that happened during session replacement attempts. A mapping from the namespace of the backend for which the replacement failed, to the reason of failure. If the reason is a *StorageException*, there is a high change that the session was left in an inconsistent state. Other reasons imply that the session replacement failed before having any effect on the state of either side.

Warning: This method can not guarantee that sessions are left in a consistent state. For example, if a notification message for the recipient is lost or heavily delayed, the recipient may not know about the new session and keep using the old one. Only use this method to attempt replacement of sessions that already seem broken. Do not attempt to replace healthy sessions.

Warning: This method does not optimize towards minimizing network usage. One notification message is sent per session to replace, the notifications are not bundled. This is to minimize the negative impact of network failure.

async get_sending_chain_length(*device*)

Get the sending chain lengths of all sessions with a device. Can be used for external staleness detection logic.

Parameters
device (*DeviceInformation*) – The device.

Return type
Dict[str, Optional[int]]

Returns
A mapping from namespace to sending chain length. *None* for the sending chain length implies that there is no session with the device for that backend.

async set_own_label(*own_label*)

Replace the label for this device, if supported by any of the backends.

Parameters
own_label (Optional[str]) – The new (optional) label for this device.

Raises

- ***DeviceListUploadFailed*** – if a device list upload failed. Forwarded from *_upload_device_list()*.
- ***DeviceListDownloadFailed*** – if a device list download failed. Forwarded from *_download_device_list()*.

Return type
None

Note: It is recommended to keep the length of the label under 53 unicode code points.

Return type
None

Parameters
own_label (*Optional[str]*) –

async get_device_information(*bare_jid*)

Parameters
bare_jid (str) – Get information about the devices of the XMPP account belonging to this bare JID.

Return type
FrozenSet[*DeviceInformation*]

Returns
Information about each device of *bare_jid*. The information includes the device id, the identity key, the trust level, whether the device is active and, if supported by any of the backends, the optional label. Returns information about all known devices, regardless of the backend they belong to.

Note: Only returns information about cached devices. The cache, however, should be up to date if PEP updates are correctly fed to *update_device_list()*. A manual update of a device list can be triggered using *refresh_device_list()* if needed.

Warning: This method attempts to download the bundle of devices whose corresponding identity key is not known yet. In case the information can not be fetched due to bundle download failures, the device is not included in the returned set.

async get_own_device_information()

Variation of *get_device_information()* for convenience.

Return type
Tuple[*DeviceInformation*, FrozenSet[*DeviceInformation*]]

Returns
A tuple, where the first entry is information about this device and the second entry contains information about the other devices of the own bare JID.

static format_identity_key(*identity_key*)

Parameters
identity_key (bytes) – The identity key to generate the fingerprint of.

Return type
List[str]

Returns
The fingerprint of the identity key in its Curve25519 form as per the specficiaton, in eight groups of eight lowercase hex chars each. Consider applying Consistent Color Generation to each individual group when displaying the fingerprint, if applicable.

before_history_sync()

Sets the library into "history synchronization mode". In this state, the library assumes that it was offline before and is now running catch-up with whatever happened during the offline phase. Make sure to call *after_history_sync()* when the history synchronization (if any) is done, so that the library can change to normal working behaviour again. The library automatically enters history synchronization mode when loaded via *create()*. Calling this method again when already in history synchronization mode has no effect.

Internally, the library does the following things differently during history synchronization:

- Pre keys are kept around during history synchronization, to account for the (hopefully rather hypothetical) case that two or more parties selected the same pre key to initiate a session with this device while it was offline. When history synchronization ends, all pre keys that were kept around are deleted and the library returns to normal behaviour.

- Empty messages to "complete" sessions or prevent staleness are deferred until after the synchronization is done. Only one empty message is sent per session when exiting the history synchronization mode.

Note: While in history synchronization mode, the library can process live events too.

Return type
None

`async after_history_sync()`

If the library is in "history synchronization mode" started by *create()* or *before_history_sync()*, calling this makes it return to normal working behaviour. Make sure to call this as soon as history synchronization (if any) is done.

Raises
MessageSendingFailed – if one of the queued empty messages could not be sent. Forwarded from *_send_message()*.

Return type
None

`async encrypt(`*bare_jids*, *plaintext*, *backend_priority_order=None*, *identifier=None*`)`

Encrypt some plaintext for a set of recipients.

Parameters

- **bare_jids** (FrozenSet[str]) – The bare JIDs of the intended recipients.
- **plaintext** (Dict[str, bytes]) – The plaintext to encrypt for the recipients. Since different backends may use different kinds of plaintext, for example just the message body versus a whole stanza using Stanza Content Encryption, this parameter is a dictionary, where the keys are backend namespaces and the values are the plaintext for each specific backend. The plaintext has to be supplied for each backend.
- **backend_priority_order** (Optional[List[str]]) – If a recipient device supports multiple versions of OMEMO, this parameter decides which version to prioritize. If None is supplied, the order of backends as passed to *create()* is assumed as the order of priority. If a list of namespaces is supplied, the first namespace supported by the recipient is chosen. Lower index means higher priority.
- **identifier** (Optional[str]) – A value that is passed on to *_make_trust_decision()* in case a trust decision is required for any of the recipient devices. This value is not processed or altered, it is simply passed through. Refer to the documentation of *_make_trust_decision()* for details.

Return type
Tuple[Dict[*Message*, *PlainKeyMaterial*], FrozenSet[EncryptionError]]

Returns
A mapping with one message per backend as the keys encrypted for each device of each recipient and for other devices of this account, and the plain key material that was used to encrypt the content of the respective message as values. This plain key material can be used to implement things like legacy OMEMO's KeyTransportMessages. Next to the messages, a set of non-critical errors encountered during encryption are returned.

Raises

- ***UnknownNamespace*** – if the backend priority order list contains a namespace of a backend that is not currently available.

- *UnknownTrustLevel* – if an unknown custom trust level name is encountered. Forwarded from *_evaluate_custom_trust_level()*.
- *TrustDecisionFailed* – if for any reason the trust decision for undecided devices failed/could not be completed. Forwarded from *_make_trust_decision()*.
- *StillUndecided* – if the trust level for one of the recipient devices still evaluates to undecided, even after *_make_trust_decision()* was called to decide on the trust.
- *NoEligibleDevices* – if at least one of the intended recipients does not have a single device which qualifies for encryption. Either the recipient does not advertize any OMEMO-enabled devices or all devices were disqualified due to missing trust or failure to download their bundles.
- *KeyExchangeFailed* – in case there is an error during the key exchange required for session building. Forwarded from *build_session_active()*.

Note: The own JID is implicitly added to the set of recipients, there is no need to list it manually.

async decrypt(*message*)

Decrypt a message.

Parameters
message (*Message*) – The message to decrypt.

Return type
Tuple[Optional[bytes], *DeviceInformation*, *PlainKeyMaterial*]

Returns
A triple, where the first entry is the decrypted plaintext and the second entry contains information about the device that sent the message. The plaintext is optional and will be None in case the message was an empty OMEMO message purely used for protocol stability reasons. The third entry is the plain key meterial transported by the message, which can be used to implement functionality like legacy OMEMO's KeyTransportMessages.

Raises

- *UnknownNamespace* – if the backend to handle the message is not currently loaded.
- *UnknownTrustLevel* – if an unknown custom trust level name is encountered. Forwarded from *_evaluate_custom_trust_level()*.
- *KeyExchangeFailed* – in case a new session is built while decrypting this message, and there is an error during the key exchange that's part of the session building. Forwarded from *build_session_passive()*.
- *MessageNotForUs* – in case the message does not seem to be encrypted for us.
- *SenderNotFound* – in case the public information about the sending device could not be found or is incomplete.
- *SenderDistrusted* – in case the identity key corresponding to the sending device is explicitly distrusted.
- *NoSession* – in case there is no session with the sending device, and the information required to build a new session is not included either.
- *PublicDataInconsistency* – in case there is an inconsistency in the public data of the sending device, which can affect the trust status.

- *MessageSendingFailed* – if an attempt to send an empty OMEMO message failed. Forwarded from *_send_message()*.
- *DecryptionFailed* – in case of backend-specific failures during decryption. Forwarded from the respective backend implementation.

Warning: Do **NOT** implement any automatic reaction to decryption failures, those automatic reactions are transparently handled by the library! *Do* notify the user about decryption failures though, if applicable.

Note: If the trust level of the sender evaluates to undecided, the message is decrypted.

Note: May send empty OMEMO messages to "complete" key exchanges or prevent staleness.

4.7 Module: storage

class omemo.storage.**Just**(*value*)

Bases: *Maybe*[ValueTypeT]

A *Maybe* that does hold a value.

__init__(*value*)

Initialize a *Just*, representing a *Maybe* that holds a value.

Parameters
value (TypeVar(ValueTypeT)) – The value to store in this *Just*.

Return type
None

property is_just: bool

Returns: Whether this is a *Just*.

Return type
bool

property is_nothing: bool

Returns: Whether this is a *Nothing*.

Return type
bool

from_just()

Return type
TypeVar(ValueTypeT)

Returns
The value if this is a *Just*.

Raises
NothingException – if this is a *Nothing*.

maybe(*default*)

Parameters
default (TypeVar(DefaultTypeT)) – The value to return if this is in instance of *Nothing*.

Return type
TypeVar(ValueTypeT)

Returns
The value if this is a *Just*, or the default value if this is a *Nothing*. The default is returned by reference in that case.

fmap(*function*)

Apply a mapping function.

Parameters
function (Callable[[TypeVar(ValueTypeT)], TypeVar(MappedValueTypeT)]) – The mapping function.

Return type
Just[MappedValueTypeT]

Returns
A new *Just* containing the mapped value if this is a *Just*. A new *Nothing* if this is a *Nothing*.

class omemo.storage.**Maybe**(**args*, ***kwds*)

Bases: ABC, Generic[ValueTypeT]

typing's *Optional[A]* is just an alias for *Union[None, A]*, which means if *A* is a union itself that allows *None*, the *Optional[A]* doesn't add anything. E.g. *Optional[Optional[X]] = Optional[X]* is true for any type *X*. This Maybe class actually differenciates whether a value is set or not.

All incoming and outgoing values or cloned using copy.deepcopy(), such that values stored in a Maybe instance are not affected by outside application logic.

abstract property is_just: bool

Returns: Whether this is a *Just*.

Return type
bool

abstract property is_nothing: bool

Returns: Whether this is a *Nothing*.

Return type
bool

abstract from_just()

Return type
TypeVar(ValueTypeT)

Returns
The value if this is a *Just*.

Raises
NothingException – if this is a *Nothing*.

abstract maybe(*default*)

Parameters
default (TypeVar(DefaultTypeT)) – The value to return if this is in instance of *Nothing*.

Return type
Union[TypeVar(ValueTypeT), TypeVar(DefaultTypeT)]

Returns
The value if this is a *Just*, or the default value if this is a *Nothing*. The default is returned by reference in that case.

abstract fmap(*function*)

Apply a mapping function.

Parameters
function (Callable[[TypeVar(ValueTypeT)], TypeVar(MappedValueTypeT)]) – The mapping function.

Return type
Maybe[MappedValueTypeT]

Returns
A new *Just* containing the mapped value if this is a *Just*. A new *Nothing* if this is a *Nothing*.

class omemo.storage.**Nothing**

Bases: *Maybe*[ValueTypeT]

A *Maybe* that does not hold a value.

__init__()

Initialize a *Nothing*, representing an empty *Maybe*.

Return type
None

property is_just: bool

Returns: Whether this is a *Just*.

Return type
bool

property is_nothing: bool

Returns: Whether this is a *Nothing*.

Return type
bool

from_just()

Return type
TypeVar(ValueTypeT)

Returns
The value if this is a *Just*.

Raises
NothingException – if this is a *Nothing*.

maybe(*default*)

Parameters
default (TypeVar(DefaultTypeT)) – The value to return if this is in instance of *Nothing*.

Return type
TypeVar(DefaultTypeT)

Returns
The value if this is a *Just*, or the default value if this is a *Nothing*. The default is returned by reference in that case.

fmap(*function*)

Apply a mapping function.

Parameters
function (Callable[[TypeVar(ValueTypeT)], TypeVar(MappedValueTypeT)]) – The mapping function.

Return type
Nothing[MappedValueTypeT]

Returns
A new *Just* containing the mapped value if this is a *Just*. A new *Nothing* if this is a *Nothing*.

exception omemo.storage.**NothingException**

Bases: Exception

Raised by *Maybe.from_just()*, in case the *Maybe* is a *Nothing*.

class omemo.storage.**Storage**(*disable_cache=False*)

Bases: ABC

A simple key/value storage class with optional caching (on by default). Keys can be any Python string, values any JSON-serializable structure.

Warning: Writing (and deletion) operations must be performed right away, before returning from the method. Such operations must not be cached or otherwise deferred.

Warning: All parameters must be treated as immutable unless explicitly noted otherwise.

Note: The *Maybe* type performs the additional job of cloning stored and returned values, which essential to decouple the cached values from the application logic.

Parameters
disable_cache (*bool*) –

__init__(*disable_cache=False*)

Configure caching behaviour of the storage.

Parameters
disable_cache (bool) – Whether to disable the cache, which is on by default. Use this parameter if your storage implementation handles caching itself, to avoid pointless double caching.

abstract async _load(*key*)

Load a value.

Parameters
key (str) – The key identifying the value.

Return type
Maybe[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]]]

Returns
The loaded value, if it exists.

Raises
StorageException – if any kind of storage operation failed. Feel free to raise a subclass instead.

abstract async _store(*key*, *value*)

Store a value.

Parameters

- **key** (str) – The key identifying the value.
- **value** (Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]]]) – The value to store under the given key.

Raises
StorageException – if any kind of storage operation failed. Feel free to raise a subclass instead.

Return type
None

abstract async _delete(*key*)

Delete a value, if it exists.

Parameters
key (str) – The key identifying the value to delete.

Raises
StorageException – if any kind of storage operation failed. Feel free to raise a subclass instead. Do not raise if the key doesn't exist.

Return type
None

async load(*key*)

Load a value.

Parameters
key (str) – The key identifying the value.

Return type
Maybe[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]]]]

Returns
The loaded value, if it exists.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_load()*.

async store(*key*, *value*)

Store a value.

Parameters

- **key** (str) – The key identifying the value.
- **value** (Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]]], Mapping[str, Union[None, float, int, str, bool]]]], Mapping[str, Union[None, float, int, str, bool, List[Union[None, float, int, str, bool]], Mapping[str, Union[None, float, int, str, bool]]]]]]]) – The value to store under the given key.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_store()*.

Return type
None

async delete(*key*)

Delete a value, if it exists.

Parameters
key (str) – The key identifying the value to delete.

Raises
StorageException – if any kind of storage operation failed. Does not raise if the key doesn't exist. Forwarded from *_delete()*.

Return type
None

async store_bytes(*key*, *value*)

Variation of *store()* for storing specifically bytes values.

Parameters

- **key** (str) – The key identifying the value.
- **value** (bytes) – The value to store under the given key.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_store()*.

Return type
None

async load_primitive(*key*, *primitive*)

Variation of *load()* for loading specifically primitive values.

Parameters

- **key** (str) – The key identifying the value.
- **primitive** (Type[TypeVar(PrimitiveTypeT, None, float, int, str, bool)]) – The primitive type of the value.

Return type
Maybe[TypeVar(PrimitiveTypeT, None, float, int, str, bool)]

Returns
The loaded and type-checked value, if it exists.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_load()*.

async load_bytes(*key*)

Variation of *load()* for loading specifically bytes values.

Parameters
key (str) – The key identifying the value.

Return type
Maybe[bytes]

Returns
The loaded and type-checked value, if it exists.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_load()*.

async load_optional(*key*, *primitive*)

Variation of *load()* for loading specifically optional primitive values.

Parameters

- **key** (str) – The key identifying the value.
- **primitive** (Type[TypeVar(PrimitiveTypeT, None, float, int, str, bool)]) – The primitive type of the optional value.

Return type
Maybe[Optional[TypeVar(PrimitiveTypeT, None, float, int, str, bool)]]

Returns
The loaded and type-checked value, if it exists.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_load()*.

async load_list(*key*, *primitive*)

Variation of *load()* for loading specifically lists of primitive values.

Parameters

- **key** (str) – The key identifying the value.
- **primitive** (Type[TypeVar(PrimitiveTypeT, None, float, int, str, bool)]) – The primitive type of the list elements.

Return type
Maybe[List[TypeVar(PrimitiveTypeT, None, float, int, str, bool)]]

Returns
The loaded and type-checked value, if it exists.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_load()*.

async load_dict(*key*, *primitive*)

Variation of *load()* for loading specifically dictionaries of primitive values.

Parameters

- **key** (str) – The key identifying the value.
- **primitive** (Type[TypeVar(PrimitiveTypeT, None, float, int, str, bool)]) – The primitive type of the dictionary values.

Return type
Maybe[Dict[str, TypeVar(PrimitiveTypeT, None, float, int, str, bool)]]

Returns
The loaded and type-checked value, if it exists.

Raises
StorageException – if any kind of storage operation failed. Forwarded from *_load()*.

exception omemo.storage.**StorageException**

Bases: *OMEMOException*

Parent type for all exceptions specifically raised by methods of *Storage*.

4.8 Module: types

class omemo.types.**AsyncFramework**(*value*)

Bases: Enum

Frameworks for asynchronous code supported by python-omemo.

ASYNCIO: str = 'ASYNCIO'

TWISTED: str = 'TWISTED'

class omemo.types.**DeviceInformation**(*namespaces*, *active*, *bare_jid*, *device_id*, *identity_key*, *trust_level_name*, *label*)

Bases: tuple

Structure containing information about a single OMEMO device.

Parameters

- **namespaces** (*FrozenSet[str]*) –
- **active** (*FrozenSet[Tuple[str, bool]]*) –
- **bare_jid** (*str*) –
- **device_id** (*int*) –
- **identity_key** (*bytes*) –
- **trust_level_name** (*str*) –
- **label** (*Optional[str]*) –

property **namespaces**

Alias for field number 0

property **active**

Alias for field number 1

property **bare_jid**

Alias for field number 2

property **device_id**

Alias for field number 3

property **identity_key**

Alias for field number 4

property **trust_level_name**

Alias for field number 5

property **label**

Alias for field number 6

exception omemo.types.**OMEMOException**

Bases: Exception

Parent type for all custom exceptions in this library.

class omemo.types.**TrustLevel**(*value*)

Bases: Enum

The three core trust levels.

TRUSTED: str = 'TRUSTED'

DISTRUSTED: str = 'DISTRUSTED'

UNDECIDED: str = 'UNDECIDED'

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

X