Soledad Documentation

Release 0.10.5

LEAP Encryption Access Project

Nov 30, 2017

Contents

Synchronization of Locally Encrypted Data Among Devices.

CHAPTER 1

What is Soledad?

Soledad is a client library and a server daemon that together allow applications to securely share a common state among devices. It is LEAP's solution for mail delivery and for synchronizing client-encrypted data among a user's devices that access an account in a LEAP provider.

Soledad is cross-platform, open source, scalable, and features a highly efficient synchronization algorithm. The Client and Server are written in Python using Twisted. Source code is available at 0xacab and is licensed under the GPLv3. Client and server are packaged together and distributed in pypi. Debian packages are also provided for the server-side component.

CHAPTER 2

Soledad documentation

2.1 Introduction

Soledad consists of a client library and server daemon that allows applications to securely share a common state among devices. The local application is presented with a simple, document-centric searchable database API. Any data saved to the database by the application is client-encrypted, backed up in the cloud, and synchronized among a user's devices. Soledad is cross-platform, open source, scalable, and features a highly efficient synchronization algorithm.

Key aspects of Soledad include:

- **Client and server:** Soledad includes a *server daemon* and a *client application library*.
- **Client-side encrypted sync:** Soledad puts very little trust in the server by *encrypting all data* before it is *synchronized* to the server and by limiting ways in which the server can modify the user's data.
- **Encrypted local storage:** All data cached locally is *stored in an encrypted database*.
- **Document database:** An application using the Soledad client library is presented with a *document-centric database API* for storage and sync. Documents may be indexed, searched, and versioned.
- **Encrypted attachments:** storage and synchronization of *Blobs* is supported.

Soledad is an acronym of "Synchronization of Locally Encrypted Documents Among Devices" and means "solitude" in Spanish.

See also:

2.1.1 The importance of data availability

Users today demand high data availability in their applications. As a user switches from device to device, the expectation is that each application will reflect the same state information across devices. Additionally, if all devices are lost or destroyed, the contemporary user expects to be able to restore her or his application data from the cloud.

In many ways, data availability has become a necessary precondition for an application to be considered "user friendly". Unfortunately, most applications attempt to provide high data availability by rolling their own custom solution or relying on a third party API, such as Dropbox. This approach is has several drawbacks: the user has no

control or access to the data should they wish to switch applications or data providers; custom data synchronizations schemes are often an afterthought, poorly designed, and vulnerable to attack and data breaches; and the user must place total trust in the provider to safeguard her or his information against requests by repressive governments.

Soledad provides secure data availability in a way that is easy for application developers to incorporate into their code.

2.1.2 Goals

Security goals

- **Client-side encryption:** Before any data is synced to the cloud, it should be encrypted on the client device.
- **Encrypted local storage:** Any data cached in the client should be stored in an encrypted format.
- **Resistant to offline attacks:** Data stored on the server should be highly resistant to offline attacks (i.e. an attacker with a static copy of data stored on the server would have a very hard time discerning much from the data).
- **Resistant to online attacks:** Analysis of storing and retrieving data should not leak potentially sensitive information.
- **Resistance to data tampering:** The server should not be able to provide the client with old or bogus data for a document.

Synchronization goals

- **Consistency:** multiple clients should all get sync'ed with the same data.
- **Selective sync:** the ability to partially sync data. For example, so a mobile device doesn't need to sync all email attachments.
- **Multi-platform:** supports both desktop and mobile clients.
- **Quota:** the ability to identify how much storage space a user is taking up.
- **Scalable cloud:** distributed master-less storage on the cloud side, with no single point of failure.
- **Conflict resolution:** conflicts are flagged and handed off to the application logic to resolve. Usability goals
- **Availability:** the user should always be able to access their data.
- **Recovery:** there should be a mechanism for a user to recover their data should they forget their password.

Known limitations

These are currently known limitations:

- The server knows when the contents of a document have changed.
- There is no facility for sharing documents among multiple users.
- Soledad is not able to prevent server from withholding new documents or new revisions of a document.
- Deleted documents are never deleted, just emptied. Useful for security reasons, but could lead to DB bloat.

Non-goals

- Soledad is not for filesystem synchronization, storage or backup. It provides an API for application code to synchronize and store arbitrary schema-less JSON documents in one big flat document database. One could model a filesystem on top of Soledad, but it would be a bad fit.
- Soledad is not intended for decentralized peer-to-peer synchronization, although the underlying synchronization protocol does not require a server. Soledad takes a cloud approach in order to ensure that a client has quick access to an available copy of the data.

2.1.3 Related projects

- Crypton: Similar goals to Soledad, but in javascript for HTML5 applications.
- Mylar: Like Crypton, Mylar can be used to write secure HTML5 applications in javascript. Uniquely, it includes support for homomorphic encryption to allow server-side searches.
- Firefox Sync: A client-encrypted data sync from Mozilla, designed to securely synchronize bookmarks and other browser settings.
- U1DB: Document synchronization API used as a basis for Soledad, without encryption.

2.2 Soledad Server

Soledad Server is a document store and a blobs server that can synchronize data with a Soledad Client.

2.2.1 Configuring

Soledad Server looks for a configuration file in `/etc/soledad/soledad-server.conf` and will read the following configuration options from the `[soledad-server]` section:

Option	Description	Default value
couch_url	The URL of the CouchDB backend storage.	`http://localhost:5984`
create_cmd	The shell command to create user databases.	None
admin_netrc	The netrc file to be used for authenticating with the CouchDB backend storage.	`/etc/couchdb/couchdb.netrc`
batching	Whether to use batching capabilities for synchronization.	`true`
blobs	Whether to provide the Blobs functionality or not.	`false`
blobs_path	The path for blobs storage in the server's file system.	`/var/lib/soledad/blobs`
concurrent_blob_writes	Limit of concurrent blob writes to the filesystem.	50
services_tokens_file	The file containing authentication tokens for services provided through the Services API.	`/etc/soledad/services.tokens`

2.2.2 Running

This is written as a Twisted application and intended to be run using the twistd command. To start the soledad server, run:

```
twistd -n --python /path/to/leap/soledad/server/server.tac
```

An systemd script is included in the Debian packages to make it feasible to start and stop the Soledad server service using a standard interface.

2.2.3 Migrations

Some updates of Soledad need manual intervention for database migration because of changes to the storage backend. In all such cases, we will document the steps needed for migration in this page.

Soledad Server 0.8 to 0.9 - Couch Database schema migration needed

Starting with Soledad Server 0.9.0, the CouchDB database schema was changed to improve speed of the server side storage backend. Because of that, this script has to be run for all Leap providers that used to provide email using Soledad Server < 0.9.0.

The migration script can be found:

- In the Soledad repository.
- In `/usr/share/soledad-server/migration/0.9/` when the `soledad-server` debian package is installed.

Instructions for migration can be found in the `README.md` file. Make sure to read it carefully and backup your data before starting the migration process.

2.3 Soledad Client

The Soledad Client is a Python library aimed to provide access to a document store that can be synchronized securelly with other deviced through the Soledad Server. Key aspects of Soledad Client include:

- **Encrypted local storage:** All data cached locally is stored in an encrypted database.
- **Client-side encrypted sync:** Soledad puts very little trust in the server by encrypting all data before it is synchronized to the server and by limiting ways in which the server can modify the user's data.
- **Document database:** An application using the Soledad client library is presented with a document-centric database API for storage and sync. Documents may be indexed, searched, and versioned.
- **Blobs storage:** The client and server API provide blobs storage, which can be used both for data delivery in the server side (i.e. email) and payload storage on the client side.

2.3.1 Setting-up

The following information is needed in order to instantiate a soledad client:

- `uuid`: the user's uuid.
- `passphrase`: the user's passphrase.
- `secrets_path`: a local path for secrets storage.
- `local_db_path`: a local path for the documents database.
- `server_url`: the Soledad Server's URL.

- `cert_file`: a local path for the CA certificate.
- `auth_token`: an authentication token obtained after logging into the provider.

Once all pieces are in place, you can instantiate the client as following:

```
from leap.soledad.client import Soledad

client = Soledad(
    uuid,
    passphrase,
    secrets_path=secrets_path,
    local_db_path=local_db_path,
    server_url=server_url,
    cert_file=cert_file,
    auth_token=token)
```

2.3.2 Usage example

Soledad is written in the Twisted asynchronous model, so you will need to make sure a reactor is running.

An example of usage of Soledad Client for creating a document and Creation of a document and synchronization is done as follows:

```
from twisted.internet import defer, reactor

@defer.inlineCallbacks
def client_usage_example():

    # create a document and sync it with the server
    yield client.create_doc({'my': 'doc'}, doc_id='some-id')
    doc = yield client.get_doc('some-id')
    yield client.sync()

    # modify the document and sync again
    doc.content = {'new': 'content'}
    yield client.put_doc(doc)
    yield client.sync()

d = client_usage_example()
d.addCallback(lambda _: reactor.stop())

reactor.run()
```

2.4 Reference

This page gathers reference documentation to understanding the internals of Soledad.

2.4.1 Environment Variables

Some environment variables affect the behaviour of Soledad:

variable	affects	description
SOLEDAD_COUCH_URL	server	override the CouchDB url.
SOLEDAD_HTTP_PERSIST	client	persist HTTP connections.
SOLEDAD_USE_PYTHON_LOGGING	client / server	use python logging instead of twisted's logger.
SOLEDAD_LOG_TO_STDOUT	client / server	log to standard output.
SOLEDAD_SERVER_CONFIG_FILE	server	use this configuration file instead of the default one.
LOCAL_SERVICES_PORT	server	which port to use for local TCP services.
HTTPS_PORT	server	which port to use for public HTTPS services.

2.4.2 Storage secrets

Soledad randomly generates secrets that are used to derive encryption keys for protecting all data that is stored in the server and in the local storage. These secrets are themselves encrypted using a key derived from the user's passphrase, and saved locally on disk.

The encrypted secrets are stored in a local file in the user's in a JSON structure that looks like this:

```
{
    'version': 2,
    'kdf': 'scrypt',
    'kdf_salt': <base64 encoded salt>,
    'kdf_length': <the length of the derived key>,
    'cipher': <a code indicating the cipher used for encryption>,
    'length': <the length of the plaintext>,
    'iv': <the initialization vector>,
    'secrets': <base64 encoding of ciphertext>,
}
```

When a client application first wants to use Soledad, it must provide the user's password to unlock the storage secrets. Currently, the storage secrets are shared among all devices with access to a particular user's Soledad database.

The storage secrets are currently backed up in the provider (encrypted with the user's passphrase) for the case where the user looses or resets her device (see *Shared database* for more information). There are plans to make this feature optional, allowing for less trust in the provider while increasing the responsibility of the user.

If the user looses her passphrase, there is currently no way of recovering her data.

2.4.3 Server-side databases

Soledad Server works with one database per user and one shared database in which user's encrypted secrets might be stored.

User databases

User databases in the server are named 'user-<uuid>' and Soledad Client may perform synchronization between its local replicas and the user's database in the server. Authorization for creating, updating, deleting and retrieving information about the user database as well as performing synchronization is handled by the *leap.soledad.server.auth* module.

Shared database

Each user may store password-encrypted recovery data in the shared database.

Recovery documents are stored in the database without any information that may identify the user. In order to achieve this, the doc_id of recovery documents are obtained as a hash of the user's uid and the user's password. User's must have a valid token to interact with recovery documents, but the server does not perform further authentication because it has no way to know which recovery document belongs to each user.

This has some implications:

- The security of the recovery document doc_id, and thus of access to the recovery document (encrypted) content, as well as tampering with the stored data, all rely on the difficulty of obtaining the user's password (supposing the user's uid is somewhat public) and the security of the hash function used to calculate the doc_id.
- The security of the content of a recovery document relies on the difficulty of obtaining the user's password.
- If the user looses his/her password, he/she will not be able to obtain the recovery document.
- Because of the above, it is recommended that recovery documents expire (not implemented yet) to prevent excess storage.

The authorization for creating, updating, deleting and retrieving recovery documents on the shared database is handled by *leap.soledad.server.auth* module.

2.4.4 Client-side databases

These are some important information about Soledad's client-side databases:

- Soledad Client uses SQLCipher for storing data.
- *Documents* and *blobs* are stored in different databases protected with the same symmetric key.
- The symmetric key used to unlock databases is chosen randomly and is stored encrypted by the user's passphrase (see *Storage secrets* for more details).

The database files currently used in the client-side are:

- `<user_id>.db`: The database for JSON documents storage.
- `<user_id>_blobs.db`: The database for storage of blobs.

Depending on how local databases are configured, you may also find files with the same names of the above but ending in `-wal` and `-shm`, which correspond to SQLCipher's Write-Ahead Logging implementation.

2.4.5 Client-side encryption and authentication

Before any user data is sent to the server, Soledad Client **symmetrically encrypts** it using AES-256 operating in GCM mode. That mode of encryption automatically calculates a **Message Authentication Code** (a MAC) during block encryption, and so gives Soledad the ability to encrypt on the fly while transmitting data to the server. Similarly, when downloading a symmetrically encrypted document from the server, Soledad Client will decrypt it and verify the MAC tag in the end before accepting the document.

The symmetric key used to encrypt a document is derived from the storage secret and the document id, with HMAC using SHA-256 as a hash function.

MAC verification of JSON documents

JSON documents are versioned (see *Document synchronization*), so in this case the calculation of the MAC also takes into account the document revision to avoid tampering. Soledad Client will refuse to accept a document if it does not include a higher revision. In this way, the server cannot rollback a document to an older revision. The server also cannot delete a document, since document deletion is handled by removing the document contents, marking it

as deleted, and incrementing the revision. However, a server can withhold from the client new documents and new revisions of a document (including withholding document deletion).

MAC verification of Blobs

Because Blobs are immutable (see *Blobs Synchronization*), in this case the MAC is calculated over the content of the blob only (i.e. no metadata is taken into account). Blob downloads will fail irrecoverably if the client is unable to verify the MAC after a certain number of retries.

Outsourced encryption by third-party applications

Soledad Client will allways do **symmetric encryption** with a secret known only to the client. But data can also be delivered directly to the user's database in the server by other applications. One example is mail delivery: the MX application receives a message targetted to a user, encrypts it with the user's OpenPGP public key and delivers it directly to the user's database in the server.

Server-side applications can define their own encryption schemes and Soledad Client will not attempt decryption and verification in those cases.

2.4.6 Document synchronization

Soledad follows the U1DB synchronization protocol with some modifications:

- A synchronization always happens between the Soledad Server and one Soledad Client. Many clients can synchronize with the same server.
- Soledad Client *always encrypts* before sending data to the server.
- Soledad Client refuses to receive a document if it is encrypted and the MAC is incorrect.
- Soledad Server doesn't try to decide about document convergence based on the document's content, because the content is client-encrypted.

Synchronization protocol

Synchronization between the Soledad Server and one Soledad Client consists of the following steps:

1. The client asks the server for the information it has stored about the last time they have synchronized (if ever).
2. The client validates that its information regarding the last synchronization is consistent with the server's information, and raises an error if not. (This could happen for instance if one of the replicas was lost and restored from backup, or if a user inadvertently tries to synchronize a copied database.)
3. The client generates a list of changes since the last change the server knows of.
4. The client checks what the last change is it knows about on the server.
5. If there have been no changes on either side that the other side has not seen, the synchronization stops here.
6. The client encrypts and sends the changed documents to the server, along with what the latest change is that it knows about on the server.
7. The server processes the changed documents, and records the client's latest change.
8. The server responds with the documents that have changes that the client does not yet know about.
9. The client decrypts and processes the changed documents, and records the server's latest change.

10. If the client has seen no changes unrelated to the synchronization during this whole process, it now sends the server what its latest change is, so that the next synchronization does not have to consider changes that were the result of this one.

Synchronization metadata

The synchronization information stored on each database replica consists of:

- The replica id of the other replica. (Which should be globally unique identifier to distinguish database replicas from one another.)
- The last known generation and transaction id of the other replica.
- The generation and transaction id of this replica at the time of the most recent succesfully completed synchronization with the other replica.

Transactions

Any change to any document in a database constitutes a transaction. Each transaction increases the database generation by 1, and is assigned a transaction id, which is meant to be a unique random string paired with each generation.

The transaction id can be used to detect the case where replica A and replica B have previously synchronized at generation N, and subsequently replica B is somehow reverted to an earlier generation (say, a restore from backup, or somebody made a copy of the database file of replica B at generation < N, and tries to synchronize that), and then new changes are made to it. It could end up at generation N again, but with completely different data.

Having random unique transaction ids will allow replica A to detect this situation, and refuse to synchronize to prevent data loss. (Lesson to be learned from this: do not copy databases around, that is what synchronization is for.)

2.4.7 Authentication

Authentication with the Soledad server is made using Twisted's Pluggable Authentication system. The validation of credentials is performed by verifying a token provided by the client.

There are currently two distinct authenticated entry points:

- A public TLS encrypted **Users API**, providing the *Synchronization* and *Blobs* services, verified against the Leap Platform `tokens` database.
- A local plaintext **Services API**, currently providing only the delivery part of the *Incoming* service, authenticated against tokens defined in a file specified on the server configuration file (see the *Services API tokens file* section).

Authorization header

The client has to provide a token encoded in an HTTP auth header, as in:

```
Authorization: Token <base64-encoded uuid:token>
```

If no token is provided, the request is considered an "anonymous" request. Anonymous requests can only access *GET /*, which returns information about the server (as the version of the server and runtime configuration options).

Services API tokens file

Credentials for services accessible through the local Services API entrypoint can be added into a file, one in each line with the format `servicename:token`, like this:

```
incoming:Zm9yYSB0ZW1lciEK
```

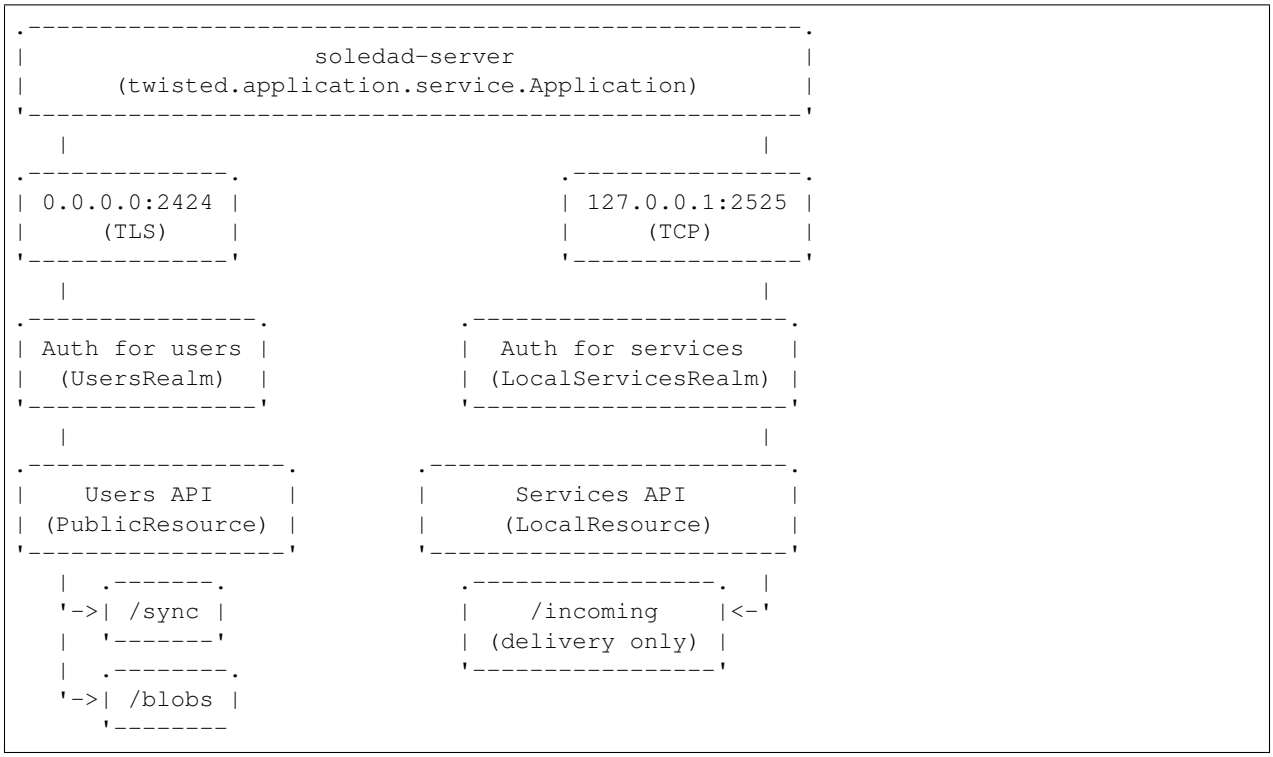
By default, Soledad Server will look for the tokens file in `/etc/soledad/services.tokens` but that is configurable (see *Configuring* for more information).

Currently, the only special credential provided is for the *Incoming* service.

Implementation

Soledad Server package includes a systemd service file that spawns a `twistd` daemon that loads a .tac file. When the server is started, two services are spawned:

- A local entrypoint for services (serving on localhost only).
- A public entrypoint for users (serving on public IP).
- Localhost and public IP ports are configurable. Default is 2424 for public IP and 2525 for localhost.



2.4.8 Blobs

The first versions of Soledad used to store all data as JSON documents, which means that binary data was also being treated as strings. This has many drawbacks, as increased memory usage and difficulties to do transfer and crypto in a proper binary pipeline.

Starting with version **0.10.0**, Soledad now has a proper blob infrastructure that decouples payloads from metadata both in storage and in the synchronization process.

Server-side blobs

The server-side implementation of blobs provides HTTP APIs for data storage using a filesystem backend.

HTTP APIs

Soledad Server provides two different REST APIs for interacting with blobs:

- A *Public Blobs HTTP API*, providing the *Blobs* service for Soledad Client (i.e. actual users of the infrastructure).
- A *Local Incoming Box HTTP API*, providing the delivery part of the *Incoming Box* service, currently used for the MX mail delivery.

Authentication is handled differently for each of the endpoints, see *Authentication* for more details.

Filesystem backend

On the server side, all blobs are currently stored in the filesystem, under `/var/lib/soledad/blobs` by default. Blobs are split in subdirectories according to the user's uuid, the namespace, and the 1, 3 and 6-letter prefixes of the blobs uuid to prevent too many files in the same directory. A second file with the extension `.flags` stores the flags for a blob.

As an example, a `PUT` request to `/blobs/some-user-id/some-blob-id` would result in the following filesystem structure in the server:

```
/var/lib/soledad/blobs
- some-user-id
    - default
        - s
            - som
                - some-b
                    - some-blob-id
                    - some-blob-id.flags
```

Client-side blobs

Data storage

On the client-side, blobs can be managed using the `BlobManager` API, which is responsible for managing storage of blobs both in local and remote storages. See *Blobs creation, retrieval, deletion and flagging* and *Blobs Synchronization* for information on the client-side API.

All data is stored locally in the `blobs` table of a SQLCipher database called `{uuid}_blobs.db` that lies in the same directory as the Soledad Client's JSON documents database (see *Client-side databases*). All actions performed locally are mirrored remotelly using the *Public Blobs HTTP API*.

Namespaces

The Blobs API supports **namespaces** so that applications can store and fetch blobs without interfering in each another. Namespaces are also used to implement the server-side *Local Incoming Box HTTP API*, used for mail delivery. All methods that deal with blobs storage, transfer and flagging provide a *namespace* parameter. If no namespace is given, the value *default* is used. See *Blobs creation, retrieval, deletion and flagging* for information on how to use namespaces.

Remote flags

In order to allow clients to control the processing of blobs that are delivered by external applications, the Blobs API has the concept of **remote flags**. The client can get and set the following flags for Blobs that reside in the server: `PENDING`, `PROCESSING`, `PROCESSED`, and `FAILED`. See *Blobs creation, retrieval, deletion and flagging* for more information on how to use flags.

Remote listing

The client can obtain a list of blobs in the server side so it can compare with its own local list and queue up blobs for download and upload. The remote listing can be ordered by *upload date* and filtered by *namespace* and *flag*. The listing can also only return the number of matches instead of the whole content. See *Blobs creation, retrieval, deletion and flagging* for more information on how to use remote listing.

Blobs Synchronization

Because blobs are immutable, synchronization is much simpler than the JSON-based *Document synchronization*. The synchronization process is as follows:

1. The client asks the server for a list of Blobs and compares it with the local list.
2. A local list is updated with pending downloads and uploads.
3. Downloads and uploads are triggered.

Immutability brings some characteristics to the blobs system:

- There's no need for storage of versions or revisions.
- Updating is not possible (you would need to delete and recreate a blob).

Client-side encryption and authentication

When uploading, the content of the blob is encrypted with a symmetric secret prior to being sent to the server. When downloading, the content of the blob is decrypted accordingly. See *Client-side encryption and authentication* for more details.

When a blob is uploaded by a client, a preamble is created and prepended to the encrypted content. The preamble is an encoded struct that contains the following metadata:

- A 2 character **magic hexadecimal number** for easy identification of a Blob data type. Currently, the value used for the magic number is: `\x13\x37`.
- The **cryptographic scheme** used for encryption. Currently, the only valid schemes are `symkey` and `external`.
- The **encryption method** used. Currently, the only valid methods are `aes_256_gcm` and `pgp`.
- The **initialization vector**.
- The **blob_id**.
- The **revision**, which is a fixed value (`ImmutableRev`) in the case of blobs.
- The **size** of the blob.

The final format of a blob that is uploaded to the server is the following:

- The URL-safe base64-encoded **preamble** (see above).

- A space to act as a **separator**.
- The URL-safe base64-encoded concatenated **encrypted data and MAC tag**.

Synchronization status

In the client-side, each blob has an associated synchronization status, which can be one of:

- `SYNCED`: The blob exists both in this client and in the server.
- `PENDING_UPLOAD`: The blob was inserted locally, but has not yet been uploaded.
- `PENDING_DOWNLOAD`: The blob exists in the server, but has not yet been downloaded.
- `FAILED_DOWNLOAD`: A download attempt has been made but the content is corrupted for some reason.

Concurrency limits

In order to increase the speed of synchronization on the client-size, concurrent database operations and transfers to the server are allowed. Despite that, to prevent indiscriminated use of client resources (cpu, memory, bandwith), concurrenty limits are set both for database operations and data transfer.

Transfer retries

When a blob is queded for download or upload, it will stay in that queue until the transfer has been successful or until there has been an unrecoverable transfer error. Currently, the only unrecoverable transfer error is a failed verification of the blob tag (i.e. a failed MAC verification).

Successive failed transfer attempts of the same blob are separated by an increasing time interval to minimize competition for resources used by other concurrent transfer attempts. The interaval starts at 10 seconds and increases to 20, 30, 40, 50, and finally 60 seconds. All further retries will be separated by a 60 seconds time interval.

2.4.9 Incoming Box

A mechanism for Trusted Applications to write encrypted data for a given user into the Soledad Server, which will sync it to the client to be processed afterwards.

- **Version**: 0.2.0
- **Date**: 27 jun 2017
- **Authors**: kali, drebs, vshyba

Overview

The `Incoming Box` is a new feature of Soledad, (from 0.10 forward), that aim at improving the mechanism by which external Trusted Applications can write data that will be delievered to a user's database in the server side, from where it will be further processed by the Soledad Client. This processing includes decrypting the payload, since the incoming data is expected to be encrypted by the Trusted Application.

Design Goal

Use the particular story about MX delivery to guide the design of a general mechanism that makes sense in the context of Soledad as a generic Encrypted Database Solution. The final solution should still be able to be the backend for different types of applications, without introducing abstractions that are too tied to the encrypted email use case.

Features

1. Deliver data (expected to be encrypted with a public-key mechanism) to the Soledad Server, so that it is written into the data space for a particular user.
2. Provide a mechanism by which Soledad Client, acting on behalf of an user, can download the data, process it in any needed way, and eventually store it again in the conventional main storage that Soledad provides. Since the data is expected to be encrypted by the delivery party, the client-side processing includes any decryption step if needed.
3. Allow to purge a document that has been correctly processed
4. Allow to mark a document that has failed to be decrypted, to avoid trying to decrypt it in every processing loop

Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Terminology

- `Blob` refers to an encrypted payload that is stored by soledad server as-is. It is assumed that the blob was end-to-end encrypted by an external component before reaching the server. (See the *Blobs* for more detail)
- A `BlobsBackend` implementation is a particular backend setup by the Soledad Server that stores all the blobs that a given user owns. For now, only a filesystem backend is provided.
- An `Incoming Message` makes reference to the representation of an abstract entity that matches exactly one message item, no matter how it is stored (ie, docs vs. blobs, single blob vs chunked, etc). It can represent one Email Message, one URL, an uploaded File, etc. For the purpose of the email use case, an Incoming Message refers to the encrypted message that MX has delivered to the incoming endpoint, which is pgp-encrypted, and can have been further obfuscated.
- By `Message Processing` we understand the sequence of downloading an incoming message, decrypting it, transform it in any needed way, and deleting the original incoming message.
- An `Incoming Box` is a subset of the Blobs stored for an user, together with the mechanisms that provide semantics to store, list, fetch and process incoming message data chronologically.

Components

- Soledad Server is the particular implementation that runs in the server (the twisted implementation is the only one to the moment). This exposes several endpoints (documents synchronization, blob storage, incoming message box).
- BlobsBackend is a particular implementation of the IBlobsBackend interface, such as FilesystemBlobsBackend, that the server has been configured to use.
- Soledad Client is the particular client that runs in different desktop replicas in the u1db sense (the twisted implementation is the only one to the moment). It executes a sync periodically, that syncs all metadata docs (soledad l2db json documents), and then downloads the blobs on demand. The blobs that need to be synced are both the encrypted blobs that are linked from metadata docs, and in this case the blobs stored in the space for a given Incoming Box.
- BlobsManager is the component that orchestrates the uploads/downloads and storage in the client side. The client storage backend currently is SQLCipher, using the BLOB type.
- A Trusted Application is any application that is authorized to write data into the user incoming box. Initially, LEAP's Encrypting Remailer Proxy (MX, for short) is going to be the main trusted application that will drive the development of the Incoming Box.
- Client Trusted Application Consumer is any implementation of Soledad's client IIncomingBoxConsumer, which is the interface that the client counterpart of the Trusted Application needs to implement in order to receive messages. It provides implementation for processing and saving Incoming Messages. In the encrypted email case, this component is the Incoming Mail Service in Bitmask Mail.

The User Experience

- From the end user perspective (ie, the user of Bitmask Mail in this case), the behaviour of the Incoming Box client in Soledad is completely transparent. Periodically, Soledad will call the Client Trusted Application Con-

sumer with new "messages" of any particular type backed by the Soledad Client Storage, without any other intervention that introducing the master passphrase.

- From the client API perspective (considering the user is a Client Trusted Application developer), all it needs to do is to implement `IIncomingBoxConsumer` interface and register this new consumer on Soledad API, telling the desired namespace it wants to consume messages from. Soledad will then take care of calling the consumer back for processing and saving of Incoming Messages.

Writing Data Into The Incoming Box

- Any payload MUST arrive already encrypted to the endpoint of the Incoming Box. Soledad Server, at version 1 of this spec, will not add any encryption to the payloads.
- The details of the encryption scheme used by the Trusted Application to encrypt the delivered payload (MX in this case) MUST be shared with the domain-specific application that processes the incoming message on the client side (Incoming Mail Service in Bitmask Mail, in this case). This means that the encryption schema MUST be communicated to the Incoming Box API in the moment of the delivery.
- Incoming Boxes MUST NOT be writeable by any other user or any external applications.

Authentication

- The Trusted Application and the Soledad Server exposing the Incoming Box endpoint MUST share a secret, that is written into the configuration files of both services.
- The Incoming Box MUST NOT be accessible as a public service from the outside.

Listing All Incoming Messages

- Soledad server will list all the messages in the Incoming Box every time that a client requests it.
- The server MUST return the number of pending messages.
- The server SHOULD skip messages from the returned set beyond a given size limit, if the client requests it so.
- The server MAY allow pagination.

Processing Incoming Messages

- The Blobs containing the Incoming Messages need the capability to be marked as in one of the following states: PENDING, PROCESSING, PROCESSED, FAILED.
- The default state for a new message in the Incoming Box is PENDING.
- Before delivering a Message to a client for processing, the server MUST mark the blob that contains it as PROCESSING, reserving the message for this client so other replicas don't try to repeat the processing.
- The server MAY expire the PROCESSING flag if the defined PROCESSING_THRESHOLD is passed, to avoid data left unusable by stalled clients.
- A message marked as PROCESSING MUST only be marked as PROCESSED by the server when it receives a confirmation by the replica that initiated the download request. This confirmation signals that the message is ready to be deleted.
- A Client MUST request to the server to mark an incoming message as PROCESSED only when there are guarantees that the incoming message has been processed without errors, and the parts resulting of its processing are acknowleged to have been uploaded successfully to the central replica in the server.

Marking a Message as Failed

- A Soledad Client MUST be able to mark a given message as FAILED. This covers the case in which a given message failed to be decrypted by a implementation-related reason (for instance: uncatched exceptions related to encoding, wrong format in serialization). The rationale is that we don't want to increase overhead by retrying decryption on every syncing loop, but we don't want to discard a particular payload. Future versions of the client might implement bugfixes or workarounds to try succeed in the processing.
- Therefore, a Soledad Client SHOULD be able to add its own version when it marks a message as temporarily failed.
- After some versions, a message SHOULD be able to be marked as permanently failed.
- Reservation MUST be released, as any other replica MUST be able to pick the message to retry. This covers the case of a new replica being added with an updated version, which can be able to handle the failure.

Deleting Incoming Messagges

- Any message in the `Incoming Box` marked as PROCESSED MAY be deleted by the server.
- Any message in the `Incoming Box` marked as PERMANENTLY FAILED MAY be deleted by the server.

Implementation Details

Server Blob Backend

In the Server Side, the implementation of the `Incoming Box` MUST be done exclusively at the level of the Blob-Storage. The Blobs implementation in both Soledad Server and Client have enough knowledge of the incoming box semantics to allow its processing to be done without resorting to writing documents in the main soledad json storage.

For simplicity, the `IncomingBox` endpoint is assumed to be running under the same process space than the rest of the Soledad Server.

Preffix Namespaces

The Trusted Application code responsible for delivering messages into Soledad Server `Incoming Box` endpoint MUST specify as a request parameter the dedicated namespace that it desires to use and be authorized to write into it. This is done so Soledad can list, filter, flag, fetch and reserve only messages known to the Trusted Application, avoiding to mix operations with blobs from the global namespace or messages from another Trusted Application.

LIST commands

The server MUST reply to several LIST commands, qualified by namespace and by other query parameters. Some of these commands are optional, but the server SHOULD reply to them signaling that they are not supported by the implementation.

The Server MUST return a list with the UIDs of the messages.

Supported listing features are: * Filter by namespace, which selects the namespace to do the list operation. * Filter by flag, which lists only messages/blobs marked with the provided flag. * Ordering, which changes the order of listing, such as older or newer first. * Count, which returns the total list size after filtering, instead of the list itself.

Client side Processing

It is assumed, for simplicity, that the Trusted Application consumer implementation shares the process memory space with the soledad client, but this doesn't have to hold true in the future.

The class responsible for client side processing on Soledad Client is named `IncomingBoxProcessingLoop`. Its role is to orchestrate processing with the Incoming Box features on server side, so it can deliver messages to it's registered Trusted Application Consumers.

- To begin a processing round, the client starts by asking a list of the pending messages.
- To avoid potentially costly traversals, the client limits the query to the most recent N blobs flagged as PENDING.
- To avoid downloading bulky messages in the incoming queue (for example, messages with very big attachments), the client MAY limit the query on a first pass to all pending blobs smaller than X Kb.
- After getting the list of Incoming Messages in the PENDING set, the client MUST start downloading the blobs according to the uuids returned.
- Download SHOULD happen in chronological order, from the list. Download may happen in several modalities: concurrently, or sequentially.
- The Soledad Client MUST provide a mechanism so that any clientside counterpart of the Trusted Application (ie: Bitmask Mail) can register a consumer for processing and saving each downloaded message.
- In the reference implementation, since the consumers that the client registers are executed in the common event loop of the Soledad Client process, attention SHOULD be payed to the callbacks not blocking the main event loop.

Example of a Trusted Application Client Consumer:

```
@implementer(interfaces.IIncomingBoxConsumer)
class MyConsumer(object):
    def __init__(self):
        self.name = 'My Consumer'

    def process(self, item, item_id, encrypted=True):
        cleartext = my_custom_decrypt(item) if encrypted else item
        processed_parts = my_custom_processing(item)
        return defer.succeed(processed_parts)

    def save(self, parts, item_id):
        return defer.gatherResults([db.save(part) for part in parts])
```

Future Features

Still subject to discussion, but some features that are desired for future iterations are:

- Provide a mechanism to retry documents marked as failed by previous revisions.
- Internalizing public key infrastructure (using ECC).
- ACLs to allow other users to push documents to an user Incoming Box.
- Provide alternative implementations of the Incoming Box endopoint (for example, in Rust)

Writing Data When User Quota is Exceeeded

- The server SHOULD move the payload to the permanent storage in the user storage space only after checking that the size of the storage currently occupied by the user data, plus the payload size does not exceed the allowed quota, if any, plus a given tolerance limit.
- The Trusted Application SHOULD receive an error message as a response to its storage request, so that it can register the failure to store the data, or inform the sender in the case in which it is acting as a delegate to deliver a message.

LIST QUALIFIERS

In order to improve performance and responsiveness, a list request MAY be qualified by the following parameters that the server SHOULD satisfy. The responses are, in any case, a list of the `uuids` of the Blobs.

- Pagination.
- Skip by SIZE THRESHOLD.
- Include messages with PROCESSING flag.

PAGINATION

- `LIMIT`: number of messages to receive in a single response
- `PAGE`: when used with limit, which page to return (limited by the number in LIMIT). (Note that, in reality, any client will just process the first page under a normal functioning mode).

Example:

```
IncomingBox.list('mx', limit=20, page=1)
```

SKIP-BY-SIZE

- SIZE_LIMIT: skips messages bigger than a given size limit, to avoid downloading payloads too big when client is interested in a quick list of incoming messages.

Example:

```
IncomingBox.list('mx', size_limit=10MB)
```

2.4.10 Document attachments

Contents:

Reasoning

The type of a Soledad document's content is JSON, which is good for efficient lookup and indexing. On the other hand, this is particularly bad for storing larger amounts of binary data, because:

- the only way to store data in JSON is as unicode string, and this uses more space than what is actually needed for binary data storage.
- upon synchronization, the content of a Soledad document needs to be completelly transferred and decrypted for the document to be available for use.

Document attachments were introduced as a means to efficiently store large payloads of binary data while avoiding the need to wait for their transfer to have access to the documents' contents.

Server-side

In the server, attachments are stored as *Blobs*. See *Public Blobs HTTP API* for more information on how to interact with the server using HTTP.

The IBlobsBackend interface is provided, so in the future there can be different ways to store attachments in the server side (think of a third-party storage, for example). Currently, the *Filesystem backend* is the only one that implements that interface.

Client-side

In the client, attachments are relations between JSON documents and blobs.

See client-side-attachment-api for reference.

Usage example

The attachments API is currently available in the *Document* class, and the document needs to know about the store to be able to manage attachments. When you create a new document with soledad, that document will already know about the store that created it, and can put/get/delete an attachment:

```
from twisted.internet.defer import inlineCallbacks

@inlineCallbacks
def attachment_example(soledad):
    doc = yield soledad.create_doc({})

    state = yield doc.get_attachment_state()
    dirty = yield doc.is_dirty()

    assert state == AttachmentStates.NONE
    assert dirty == False

    yield doc.put_attachment(open('hackers.txt'))
    state = yield doc.get_attachment_state()
    dirty = yield doc.is_dirty()

    assert state | AttachmentState.LOCAL
    assert dirty == True

    yield soledad.put_doc(doc)
    dirty = yield doc.is_dirty()
```

```
    assert dirty == False

    yield doc.upload_attachment()
    state = yield doc.get_attachment_state()

    assert state | AttachmentState.REMOTE
    assert state == AttachmentState.SYNCED

    fd = yield doc.get_attachment()
    assert fd.read() == open('hackers.txt').read()
```

2.5 Development

As an open source project, we welcome contributions of all forms. This page should help you get started with development for Soledad.

2.5.1 Contributing

Thank you for your interest in contributing to Soledad!

Filing bug reports

Bug reports are very welcome. Please file them on the 0xacab issue tracker. Please include an extensive description of the error, the steps to reproduce it, the version of Soledad used and the provider used (if applicable).

Patches

All patches to Soledad should be submitted in the form of pull requests to the main Soledad repository. These pull requests should satisfy the following properties:

- The pull request should focus on one particular improvement to Soledad. Create different pull requests for unrelated features or bugfixes.
- Code should follow PEP 8, especially in the "do what code around you does" sense.
- Pull requests that introduce code must test all new behavior they introduce as well as for previously untested or poorly tested behavior that they touch.
- Pull requests are not allowed to break existing tests. We will consider pull requests that are breaking the CI as work in progress.
- When introducing new functionality, please remember to write documentation.
- Please sign all of your commits. If you are merging code from other contributors, you should sign their commits.

Review

Pull requests must be reviewed before merging. The final responsibility for the reviewing of merged code lies with the person merging it. Exceptions to this rule are modifications to documentation, packaging, or tests when the need of agility of merging without review has little or no impact on the security and functionality of working code.

Getting help

If you need help you can reach us through one of the following means:

- IRC: `#leap` at `irc.freenode.org`.
- Mailing list: leap-discuss@lists.riseup.net

2.5.2 Backwards-compatibility and deprecation policy

Since Soledad has not reached a stable *1.0* release yet, no guarantees are made about the stability of its API or the backwards-compatibility of any given version.

Currently, the internal storage representation is experimenting changes that will take some time to mature and settle up. For the moment, no given SOLEDAD release is offering any backwards-compatibility guarantees.

Although serious efforts are being made to ensure no data is corrupted or lost while upgrading soledad versions, it's not advised to use SOLEDAD for any critical storage at the moment, or to upgrade versions without any external data backup (for instance, an email application that uses SOLEDAD should allow to export mail data or PGP keys in a convertible format before upgrading).

Deprecation Policy

The points above standing, the development team behind SOLEDAD will strive to provide clear migration paths between any two given, consecutive **minor releases**, in an automated form wherever possible.

This means, for example, that a migration script will be provided with the `0.10` release, to migrate data stored by any of the `0.9.x` soledad versions. Another script will be provided to migrate from `0.10` to `0.11`, etc (but not, for instance, from `0.8` to `0.10`).

At the same time, there's a backwards-compatibility policy of **deprecating APIs after 2 minor releases**. This means that a feature will start to be marked as deprecated in `0.10`, with a warning being raised for 2 minor releases, and the API will disappear completely no sooner than in `0.12`.

2.5.3 Tests

We use pytest as a testing framework and Tox as a test environment manager. Currently, tests reside in the *testing/* folder and some of them need a couchdb server to be run against.

If you do have a couchdb server running on localhost on default port, the following command should be enough to run tests:

```
tox
```

CouchDB dependency

In case you want to use a couchdb on another host or port, use the *–couch-url* parameter for *pytest*:

```
tox -- --couch-url=http://couch_host:5984
```

If you want to exclude all tests that depend on couchdb, deselect tests marked with *needs_couch*:

```
tox -- -m 'not needs_couch'
```

Benchmark tests

A set of benchmark tests is provided to measure the time and resources taken to perform some actions. See the *documentation for benchmarks*.

2.5.4 Benchmarks

We currently use pytest-benchmark to write tests to assess the time and resources taken by various tasks.

To run benchmark tests, once inside a cloned Soledad repository, do the following:

```
tox -e benchmark
```

Results of automated benchmarking for each commit in the repository can be seen in: https://benchmarks.leap.se/.

Benchmark tests also depend on *tox* and *CouchDB*. See the *Tests* page for more information on how to setup the test environment.

Test repetition

`pytest-benchmark` runs tests multiple times so it can provide meaningful statistics for the time taken for a tipical run of a test function. The number of times that the test is run can be manually or automatically configured.

When automatically configured, the number of runs is decided by taking into account multiple `pytest-benchmark` configuration parameters. See the the corresponding documenation for more details on how automatic calibration works.

To achieve a reasonable number of repetitions and a reasonable amount of time at the same time, we let `pytest-benchmark` choose the number of repetitions for faster tests, and manually limit the number of repetitions for slower tests.

Currently, tests for *synchronization* and *sqlcipher asynchronous document creation* are fixed to run 4 times each. All the other tests are left for `pytest-benchmark` to decide how many times to run each one. With this setup, the benchmark suite is taking approximatelly 7 minutes to run in our CI server. As the benchmark suite is run twice (once for time and cpu stats and a second time for memory stats), the whole benchmarks run takes around 15 minutes.

The actual number of times a test is run when calibration is done automatically by `pytest-benchmark` depends on many parameters: the time taken for a sample run and the configuration of the minimum number of rounds and maximum time allowed for a benchmark. For a snapshot of the number of rounds for each test function see the soledad benchmarks wiki page.

Sync size statistics

Currenly, the main use of Soledad is to synchronize client-encrypted email data. Because of that, it makes sense to measure the time and resources taken to synchronize an amount of data that is realistically comparable to a user's email box.

In order to determine what is a good example of dataset for synchronization tests, we used the size of messages of one week of incoming and outgoing email flow of a friendly provider. The statistics that came out from that are (all sizes are in KB):

	outgoing	incoming
min	0.675	0.461
max	25531.361	25571.748
mean	252.411	110.626
median	5.320	14.974
mode	1.404	1.411
stddev	1376.930	732.933

Sync test scenarios

Ideally, we would want to run tests for a big data set (i.e. a high number of documents and a big payload size), but that may be infeasible given time and resource limitations. Because of that, we choose a smaller data set and suppose that the behaviour is somewhat linear to get an idea for larger sets.

Supposing a data set total size of 10MB, some possibilities for number of documents and document sizes for testing download and upload can be seen below. Scenarios marked in bold are the ones that are actually run in the current sync benchmark tests, and you can see the current graphs for each one by following the corresponding links:

- 10 x 1M
- **20 x 500K** (upload, download)
- **100 x 100K** (upload, download)
- 200 x 50K
- **1000 x 10K** (upload, download)

In each of the above scenarios all the documents are of the same size. If we want to account for some variability on document sizes, it is sufficient to come up with a simple scenario where the average, minimum and maximum sizes are somehow coherent with the above statistics, like the following one:

- 60 x 15KB + 1 x 1MB

2.5.5 Compatibility

This page keeps notes about compatibility between different versions of Soledad and between Soledad and other components of the LEAP Platform.

- Upgrades of Soledad Server < 0.9.0 to >= 0.9.0 need database migration because older code used to use CouchDB's design documents, while newer code got rid of that because it made everything cpu and memory hungry. See the documentation for more information.
- Soledad Server >= 0.7.0 is incompatible with client < 0.7.0 because of modifications on encrypted document MAC calculation.
- Soledad Server >= 0.7.0 is incompatible with LEAP Platform < 0.6.1 because that platform version implements ephemeral tokens databases and Soledad Server needs to act accordingly.

2.5.6 Changelog

0.10.6 - master

Note: This version is not yet released and is under active development.

0.10.5 - Wed 22 Nov, 2017

Server

- [feature] add the SOLEDAD_COUCH_URL environment variable on server side
- [bug] properly shutdown server if startup checks fail
- [bug] fix and improve logging of server startup checks

0.10.4 - Wed 15 Nov, 2017

Server

- [feature] improve server entrypoint (several refactors and introduction of binary in debian package)
- [feature] improve speed of server startup
- [bug] wait for couch schema, configuration and environment checks before running servers.
- [bug] debian package now reports the correct version
- [pkg] improve user db creation script (fixes and man page)
- [bug] limit concurrent blob writes in server

Client

- [feature] support unsynced local only blobs
- [feature] add/fix blob deletion
- [feature] add and control concurrency of blobs local and remote operations
- [feature] add retries for blob transfers
- [bug] fixes in blob download/upload pipeline
- [bug] improve resilience of blobs concurrent access to sqlcipher
- [bug] fix blobs preamble flakiness
- [refactor] split blobs backend in many modules

Misc

- [doc] many documentation improvements
- [doc] documentation was mirrored in https://leap.se/en/docs/design/soledad
- [benchmarks] add server scalability tests
- [benchmarks] add outlier detection
- [test] add e2e test for incoming mail pipeline
- [pkg] Add packages for debian buster.
- [pkg] deb: Make soledad-client depend on soledad-common

0.10.3 - Mon 11 Sep, 2017

Server

- [feat] Finished adding support for Incoming API
- [feat] Get config file name from environment variable.
- [bug] Add DELETE method to url mapper.
- [bug] Use correct keyword argument for server state initialization.
- #8924: [bug] FileBodyProducer consumer usage wasn't closing the file

Client

- [feat] Add columns for sync state of blobs inside sqlcipher
- [bug] Several bugfixes for BlobManager initialization.
- [bug] Fix usage of StringIO class in gzip middleware.
- #8924: [bug] FileBodyProducer consumer usage wasn't closing the file

Misc

- Use latest version of pytest-benchmark.
- Find correct twistd when outside tox envs
- Build packages for zesty and stretch.
- Add benchmark comparing legacy vs blobs sync.
- Add reactor responsiveness tests.

0.10.2 - Mon 21 Aug, 2017

Server

- Enforce namespace to default on server
- Add path partitioning to namespaces

Client

- Add namespace to local blobs db table
- Track namespace information on blobs client

0.10.1 - Mon 07 Aug, 2017

Server

- Fixes IncomingBox missing preamble separator (space) which causes client to fail parsing.

Client

- Adds IncomingBoxProcessLoop and implement the process flow for IncominBox specification.
- Adds IIncomingBoxConsumer interface, which can be used by Soledad apps to implement consumers for IncomingBox feature.

0.10.0 - 18 July, 2017

Server

- Add an incoming API for email delivery. In the future, this may be used by external applications for message delivery.
- Add namespace capability.
- List incoming blobs in chronological order.
- Finish minimal filesystem backend for blobs.
- Update BlobManager to support new server features, such as: namespaces, incoming and listing.
- Make the backend configurable for incoming API, so it can use CouchDB now and Blobs later.

Client

- Use OpenSSL backend for scrypt if OpenSSL >= 1.1

Misc

- Refactor preamble to account for PGP encryption scheme
- Removes scrypt dependency
- Unification of Client, Server and Common in a Single python package.
- Build soledad debian package with git-buildpackage.
- Document deprecation policy.
- Documentation is automatically uploaded to: https://soledad.readthedocs.io/
- Launch benchmarks website: https://benchmarks.leap.se/

0.9.6 - 31 May, 2017

Server

- Minimal Filesystem BlobsBackend implementation, disabled by default.

Client

- Minimal Blobs manager implementation
- Blobs API
- Ability to generate recovery code.
- Fix deprecated multibackend call (cryptography).

Misc

- Post benchmark results to elasticsearch
- Build docker image and push it to registry every time the dockerfile used for tests is changed
- Fix flaky tests
- Cleanup old documentation.
- Added dependency on treq.
- Improve cpu/memory profiling.
- Bumped version to upload wheels to pypi, to workaround for dbschema.sql not found after installation in virtualenv.

0.9.5 - 17 March, 2017

Server

- Make database creation appear in logs

Client

- #8721: Remove offline flag
- Fix raising of invalid auth token error
- Add default version when decrypting secrets
- Secrets version defaults to v1

Misc

- First steps porting soledad to python3

0.9.3 - 06 March, 2017

Server

- Refactor authentication code to use twisted credential system.
- Announce server blobs capabilities
- #8764: Allow unauthenticated users to retrieve the capabilties banner.

- #6178: Add robots.txt
- #8762: Add a systemd service file
- Add script to deploy from git

Client

- #8758: Add blob size to the crypto preamble
- Improve secrets generation and storage code
- Add offline status to soledad client api.
- Remove syncable property

Misc

- Improvements in performance benchmarks.

0.9.2 - 22 December, 2016

Performance improvements

- use AES 256 GCM mode instead of CTR-HMAC.
- streaming encryption/decryption and data transfer.

Server

- move server to a twisted resource entrypoint.

Client

- use twisted http agent in the client.
- maintain backwards compatibility with old crypto scheme (AES 256 CTR-HMAC). No migration for now, only in 0.10.
- remove the encryption/decryption pools, replace for inline streaming crypto.
- use sqlcipher transactions on sync.

0.9.1 - 27 November, 2016

Server side bug fixes

- fix import on create-user-db script
- patch twisted logger so it works with twistd –syslog
- delay couch state initialization
- improve missing couch config doc error logging

- separate server application into another file

0.9.0 - 11 November, 2016

Main features

- Server-side changes in couch backend schema.
- Use of tox and pytest to run tests.
- Performance tests.

Server

*** Attention: Migration needed! ***

This version of soledad uses a different database schema in the server couch backend. The difference from the old schema is that the use of design documents for storing and accessing soledad db metadata was removed because incurred in too much memory and time overhead for passing data to the javascript interpreter.

Because of that, you need to run a migration script on your database. Check the *scripts/migration/0.9.0/* diretctory for instructions on how to run the migration script on your database. Don't forget to backup before running the script!

Bugfixes

- Fix order of multipart serialization when writing to couch.

Features

- Log to syslog.
- Remove usage of design documents in couch backend.
- Use _local couch docs for metadata storage.
- Other small improvements in couch backend.

0.8.1 - 14 July, 2016

Client

Features

- Add recovery document format version for future migrations.
- Use DeferredLock instead of its locking cousin.
- Use DeferredSemaphore instead of its locking cousin.

Bugfixes

- #8180: Initialize OpenSSL context just once.
- Remove document content conversion to unicode. Users of API are responsible for only passing valid JSON to Soledad for storage.

Misc

- Add ability to get information about sync phases for profiling purposes.
- Add script for setting up develop environment.
- Refactor bootstrap to remove shared db lock.
- Removed multiprocessing from encdecpool with some extra refactoring.
- Remove user_id argument from Soledad init.

Common

Features

- Embed l2db, forking u1db.

Misc

- Toxify tests.

0.8.0 - 18 Apr, 2016

Client

Features

- #7656: Emit multi-user aware events.
- Client will now send documents at a limited size batch due to changes on SyncTarget. The default limit is 500kB. Disabled by default.

Bugfixes

- #7503: Do not signal sync completion if sync failed.
- Handle missing design doc at GET (get_sync_info). Soledad server can handle this during sync.

Misc

- #7195: Use cryptography instead of pycryptopp.

Known Issues

- Upload phase of client syncs is still quite slow. Enabling size limited batching can help, but you have to make sure that your server is compatible.

Server

Features

- General performance improvements.
- #7509: Moves config directory from /etc/leap to /etc/soledad.
- Adds a new config parameter 'create_cmd', which allows sysadmin to specify which command will create a database. That command was added in pkg/create-user-db and debian package automates steps needed for sudo access.
- Read netrc path from configuration file for create-user-db command.
- 'create-user-db' script now can be configured from soledad-server.conf when generating the user's security document.
- Migrating a user's database to newest design documents is now possible by using a parameter '–migrate-all' on 'create-user-db' script.
- Remove tsafe monkeypatch from SSL lib, as it was needed for Twisted <12
- Added two methods to start and finish a batch on backend. They can be used to change database behaviour, allowing batch operations to be optimized.

Common

Features

- Add a sanitized command executor for database creation and re-enable user database creation on CouchServer-State via command line.

Bugfixes

- #7626: Subclass a leaky leap.common.couch exception to avoid depending on couch.

2.6 API Reference

2.6.1 Soledad Client API

class `leap.soledad.client.`**`Soledad`**(*uuid*, *passphrase*, *secrets_path*, *local_db_path*, *server_url*, *cert_file*, *shared_db=None*, *auth_token=None*, *with_blobs=False*)

Bases: `object`

Soledad provides encrypted data storage and sync.

A Soledad instance is used to store and retrieve data in a local encrypted database and synchronize this database with Soledad server.

This class is also responsible for bootstrapping users' account by creating cryptographic secrets and/or storing/fetching them on Soledad server.

__init__(*uuid*, *passphrase*, *secrets_path*, *local_db_path*, *server_url*, *cert_file*, *shared_db=None*, *auth_token=None*, *with_blobs=False*)

Initialize configuration, cryptographic keys and dbs.

Parameters

- **uuid** (*str*) – User's uuid.
- **passphrase** (*unicode*) – The passphrase for locking and unlocking encryption secrets for local and remote storage.
- **secrets_path** (*str*) – Path for storing encrypted key used for symmetric encryption.
- **local_db_path** (*str*) – Path for local encrypted storage db.
- **server_url** (*str*) – URL for Soledad server. This is used either to sync with the user's remote db and to interact with the shared recovery database.
- **cert_file** (*str*) – Path to the certificate of the ca used to validate the SSL certificate used by the remote soledad server.
- **shared_db** (*HTTPDatabase*) – The shared database.
- **auth_token** (*str*) – Authorization token for accessing remote databases.
- **with_blobs** – A boolean that specifies if this soledad instance should enable blobs storage when initialized. This will raise if it's not the first initialization and the passed value is different from when the database was first initialized.

Raises **BootstrapSequenceError** – Raised when the secret initialization sequence (i.e. retrieval from server or generation and storage on server) has failed for some reason.

change_passphrase(*new_passphrase*)

Change the passphrase that encrypts the storage secret.

Parameters **new_passphrase** (*unicode*) – The new passphrase.

Raises **NoStorageSecret** – Raised if there's no storage secret available.

close()

Close underlying U1DB database.

create_doc(**args*, ***kwargs*)

Create a new document.

You can optionally specify the document identifier, but the document must not already exist. See 'put_doc' if you want to override an existing document. If the database specifies a maximum document size and the document exceeds it, create will fail and raise a DocumentTooBig exception.

Parameters

- **content** (*dict*) – A Python dictionary.
- **doc_id** (*str*) – An optional identifier specifying the document id.

Returns A deferred whose callback will be invoked with a document.

Return type twisted.internet.defer.Deferred

create_doc_from_json(*json*, *doc_id=None*)
Create a new document.

You can optionally specify the document identifier, but the document must not already exist. See 'put_doc' if you want to override an existing document. If the database specifies a maximum document size and the document exceeds it, create will fail and raise a DocumentTooBig exception.

Parameters

- **json** (*dict*) – The JSON document string
- **doc_id** (*str*) – An optional identifier specifying the document id.

Returns A deferred whose callback will be invoked with a document.

Return type twisted.internet.defer.Deferred

create_index(*index_name*, **index_expressions*)
Create a named index, which can then be queried for future lookups.

Creating an index which already exists is not an error, and is cheap. Creating an index which does not match the index_expressions of the existing index is an error. Creating an index will block until the expressions have been evaluated and the index generated.

Parameters

- **index_name** (*str*) – A unique name which can be used as a key prefix
- **index_expressions** – index expressions defining the index information.

  Examples:

  "fieldname", or "fieldname.subfieldname" to index alphabetically sorted on the contents of a field.

  "number(fieldname, width)", "lower(fieldname)"

Returns A deferred.

Return type twisted.internet.defer.Deferred

create_recovery_code()

default_prefix = '/home/docs/.config/leap/soledad'
A dictionary that holds locks which avoid multiple sync attempts from the same database replica. The dictionary indexes are the paths to each local db, so we guarantee that only one sync happens for a local db at a time.

delete_doc(*doc*)
Mark a document as deleted.

Will abort if the current revision doesn't match doc.rev. This will also set doc.content to None.

Parameters **doc** (*leap.soledad.common.document.Document*) – A document to be deleted.

Returns A deferred.

Return type twisted.internet.defer.Deferred

delete_index(*index_name*)
Remove a named index.

Parameters **index_name** (*str*) – The name of the index we are removing

Returns A deferred.

Return type twisted.internet.defer.Deferred

get_all_docs(*include_deleted=False*)
Get the JSON content for all documents in the database.

Parameters **include_deleted** (*bool*) – If set to True, deleted documents will be returned with empty content. Otherwise deleted documents will not be included in the results.

Returns A deferred which, when fired, will pass the a tuple containing (generation, [Document]) to the callback, with the current generation of the database, followed by a list of all the documents in the database.

Return type twisted.internet.defer.Deferred

get_count_from_index(*index_name*, **key_values*)
Return the count for a given combination of index_name and key values.

Extension method made from similar methods in u1db version 13.09

Parameters

- **index_name** (*str*) – The index to query
- **key_values** (*tuple*) – values to match. eg, if you have an index with 3 fields then you would have: get_from_index(index_name, val1, val2, val3)

Returns A deferred whose callback will be invoked with the count.

Return type twisted.internet.defer.Deferred

get_doc(*doc_id*, *include_deleted=False*)
Get the JSON string for the given document.

Parameters

- **doc_id** (*str*) – The unique document identifier
- **include_deleted** (*bool*) – If set to True, deleted documents will be returned with empty content. Otherwise asking for a deleted document will return None.

Returns A deferred whose callback will be invoked with a document object.

Return type twisted.internet.defer.Deferred

get_doc_conflicts(*doc_id*)
Get the list of conflicts for the given document.

The order of the conflicts is such that the first entry is the value that would be returned by "get_doc".

Parameters **doc_id** (*str*) – The unique document identifier

Returns A deferred whose callback will be invoked with a list of the Document entries that are conflicted.

Return type twisted.internet.defer.Deferred

get_docs(*doc_ids*, *check_for_conflicts=True*, *include_deleted=False*)
Get the JSON content for many documents.

Parameters

- **doc_ids** (*list*) – A list of document identifiers.
- **check_for_conflicts** (*bool*) – If set to False, then the conflict check will be skipped, and 'None' will be returned instead of True/False.

- **include_deleted** (*bool*) – If set to True, deleted documents will be returned with empty content. Otherwise deleted documents will not be included in the results.

Returns A deferred whose callback will be invoked with an iterable giving the document object for each document id in matching doc_ids order.

Return type twisted.internet.defer.Deferred

get_from_index(*index_name*, **key_values*)
Return documents that match the keys supplied.

You must supply exactly the same number of values as have been defined in the index. It is possible to do a prefix match by using '*' to indicate a wildcard match. You can only supply '*' to trailing entries, (eg 'val', '*', '*' is allowed, but '*', 'val', 'val' is not.) It is also possible to append a '*' to the last supplied value (eg 'val*', '*', '*' or 'val', 'val*', '*', but not 'val*', 'val', '*')

Parameters

- **index_name** (*str*) – The index to query
- **key_values** (*list*) – values to match. eg, if you have an index with 3 fields then you would have: get_from_index(index_name, val1, val2, val3)

Returns A deferred whose callback will be invoked with a list of [Document].

Return type twisted.internet.defer.Deferred

get_index_keys(*index_name*)
Return all keys under which documents are indexed in this index.

Parameters **index_name** (*str*) – The index to query

Returns A deferred whose callback will be invoked with a list of tuples of indexed keys.

Return type twisted.internet.defer.Deferred

get_or_create_service_token(**args*, ***kwargs*)
Return the stored token for a given service, or generates and stores a random one if it does not exist.

These tokens can be used to authenticate services.

get_range_from_index(*index_name*, *start_value*, *end_value*)
Return documents that fall within the specified range.

Both ends of the range are inclusive. For both start_value and end_value, one must supply exactly the same number of values as have been defined in the index, or pass None. In case of a single column index, a string is accepted as an alternative for a tuple with a single value. It is possible to do a prefix match by using '*' to indicate a wildcard match. You can only supply '*' to trailing entries, (eg 'val', '*', '*' is allowed, but '*', 'val', 'val' is not.) It is also possible to append a '*' to the last supplied value (eg 'val*', '*', '*' or 'val', 'val*', '*', but not 'val*', 'val', '*')

Parameters

- **index_name** (*str*) – The index to query
- **start_values** (*tuple*) – tuples of values that define the lower bound of the range. eg, if you have an index with 3 fields then you would have: (val1, val2, val3)
- **end_values** (*tuple*) – tuples of values that define the upper bound of the range. eg, if you have an index with 3 fields then you would have: (val1, val2, val3)

Returns A deferred whose callback will be invoked with a list of [Document].

Return type twisted.internet.defer.Deferred

list_indexes()
List the definitions of all known indexes.

Returns A deferred whose callback will be invoked with a list of [('index-name', ['field', 'field2'])] definitions.

Return type twisted.internet.defer.Deferred

local_db_file_name = 'soledad.u1db'

local_db_path

put_doc(*doc*)
Update a document.

If the document currently has conflicts, put will fail. If the database specifies a maximum document size and the document exceeds it, put will fail and raise a DocumentTooBig exception.

============================== WARNING ============================== This method converts the document's contents to unicode in-place. This means that after calling *put_doc(doc)*, the contents of the document, i.e. *doc.content*, might be different from before the call. ============================== WARNING ==============================

Parameters **doc** (*leap.soledad.common.document.Document*) – A document with new content.

Returns A deferred whose callback will be invoked with the new revision identifier for the document. The document object will also be updated.

Return type twisted.internet.defer.Deferred

raw_sqlcipher_operation(**args*, ***kw*)
Run a raw sqlcipher operation in the local database, and return a deferred that will be fired with None.

raw_sqlcipher_query(**args*, ***kw*)
Run a raw sqlcipher query in the local database, and return a deferred that will be fired with the result.

resolve_doc(*doc*, *conflicted_doc_revs*)
Mark a document as no longer conflicted.

We take the list of revisions that the client knows about that it is superseding. This may be a different list from the actual current conflicts, in which case only those are removed as conflicted. This may fail if the conflict list is significantly different from the supplied information. (sync could have happened in the background from the time you GET_DOC_CONFLICTS until the point where you RESOLVE)

Parameters

- **doc** (*Document*) – A Document with the new content to be inserted.
- **conflicted_doc_revs** (*list(str)*) – A list of revisions that the new content supersedes.

Returns A deferred.

Return type twisted.internet.defer.Deferred

secrets
Return the secrets object.

Returns The secrets object.

Return type Secrets

secrets_file_name = 'soledad.json'

sync()
Synchronize documents with the server replica.

This method uses a lock to prevent multiple concurrent sync processes over the same local db file.

Returns A deferred lock that will run the actual sync process when the lock is acquired, and which will fire with with the local generation before the synchronization was performed.

Return type twisted.internet.defer.Deferred

sync_lock
Class based lock to prevent concurrent syncs using the same local db file.

Returns A shared lock based on this instance's db file path.

Return type DeferredLock

sync_stats()

syncing
Return wether Soledad is currently synchronizing with the server.

Returns Wether Soledad is currently synchronizing with the server.

Return type bool

userid

2.6.2 Document Attachments API

Soledad Documents implement the `IDocumentWithAttachment` API that associates *Blobs* with documents.

class `leap.soledad.client._document.`**IDocumentWithAttachment**
Bases: `zope.interface.Interface`

A document that can have an attachment.

delete_attachment()
Delete the attachment of this document.

The pointer to the attachment will be removed from the document content, but the document has to be manually put in the database to reflect modifications.

Returns A deferred which fires when the attachment has been deleted from local storage.

Return type Deferred

download_attachment()
Download this document's attachment.

Returns A deferred which fires with the state of the attachment after it's been downloaded, or NONE if there's no attachment for this document.

Return type Deferred

get_attachment()
Return the data attached to this document.

If document content contains a pointer to the attachment, try to get the attachment from local storage and, if not found, from remote storage.

Returns A deferred which fires with a file like-object whose content is the attachment of this document, or None if nothing is attached.

Return type Deferred

get_attachment_state()
Return the state of the attachment of this document.

The state is a member of AttachmentStates and is of one of NONE, LOCAL, REMOTE or SYNCED.

Returns A deferred which fires with The state of the attachment of this document.

Return type Deferred

is_dirty()
Return whether this document's content differs from the contents stored in local database.

Returns A deferred which fires with True or False, depending on whether this document is dirty or not.

Return type Deferred

put_attachment(*fd*)
Attach data to this document.

Add the attachment to local storage, enqueue for upload.

The document content will be updated with a pointer to the attachment, but the document has to be manually put in the database to reflect modifications.

Parameters **fd** (*file-like*) – A file-like object whose content will be attached to this document.

Returns A deferred which fires when the attachment has been added to local storage.

Return type Deferred

set_store(*store*)
Set the store used by this file to manage attachments.

Parameters **store** (Soledad) – The store used to manage attachments.

upload_attachment()
Upload this document's attachment.

Returns A deferred which fires with the state of the attachment after it's been uploaded, or NONE if there's no attachment for this document.

Return type Deferred

2.6.3 Blobs Client-side Python API

Each Soledad Client has a property called `blobmanager` which is an instance of the `BlobManager` class and handles Blobs creation, retrieval, deletion and synchronization.

Blobs creation, retrieval, deletion and flagging

The `BlobManager` class is responsible for blobs creation, retrieval, deletion, flagging and synchronizing. For better code organization, the methods related to synchronization are implemented separatelly in a superclass (see *Blobs Synchronization*).

class leap.soledad.client._db.blobs.**BlobManager**(*local_path*, *remote*, *key*, *secret*, *user*, *token=None*, *cert_file=None*)
Bases: *leap.soledad.client._db.blobs.sync.BlobsSynchronizer*

The BlobManager can list, put, get, set flags and synchronize blobs stored in local and remote storages.

__init__(*local_path*, *remote*, *key*, *secret*, *user*, *token=None*, *cert_file=None*)
Initialize the blob manager.

Parameters

- **local_path** (*str*) – The path for the local blobs database.
- **remote** (*str*) – The URL of the remote storage.
- **secret** (*str*) – The secret used to encrypt/decrypt blobs.
- **user** (*str*) – The uuid of the user.
- **token** (*str*) – The access token for interacting with remote storage.
- **cert_file** (*str*) – The path to the CA certificate file.

close()

concurrent_transfers_limit = 3

concurrent_writes_limit = 100

count(*namespace=''*)
Count the number of blobs.

Parameters **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

Returns A deferred that fires with a dict parsed from the JSON response, which *count* key has the number of blobs as value. Eg.: {"count": 42}

Return type twisted.internet.defer.Deferred

delete(*blob_id*, *namespace=''*)
Delete a blob from local and remote storages.

Parameters

- **blob_id** (*str*) – Unique identifier of a blob.
- **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

Returns A deferred that fires when the operation finishes.

Return type twisted.internet.defer.Deferred

get(**args*, ***kwargs*)
Get the blob from local storage or, if not available, from the server.

Parameters

- **blob_id** (*str*) – Unique identifier of a blob.
- **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

get_flags(**args*, ***kwargs*)
Get flags from a given blob_id.

Parameters

- **blob_id** (*str*) – Unique identifier of a blob.
- **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

Returns A deferred that fires with a list parsed from JSON response. Eg.: [Flags.PENDING]

Return type twisted.internet.defer.Deferred

local_list(*namespace=''*)

local_list_status(*status*, *namespace=''*)

max_decrypt_retries = 3

put(*doc*, *size*, *namespace=''*, *local_only=False*)
Put a blob in local storage and upload it to server.

Parameters

- **doc** (*leap.soledad.client._document.BlobDoc*) – A BlobDoc representing the blob.
- **size** (*int*) – The size of the blob.
- **local_only** (*bool*) – Avoids sync (doesn't send to server).
- **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

remote_list(**args*, ***kwargs*)
List blobs from server, with filtering and ordering capabilities.

Parameters

- **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.
- **order_by** (*str*) – Optional parameter to order results. Possible values are: date or +date - Ascending order (older first) -date - Descending order (newer first)
- **deleted** – Optional paramter to return only deleted blobs.
- **filter_flag** (*leap.soledad.common.blobs.Flags*) – Optional parameter to filter listing to results containing the specified tag.
- **only_count** (*bool*) – Optional paramter to return only the number of blobs found.

Returns A deferred that fires with a list parsed from the JSON response, holding the requested list of blobs. Eg.: ['blob_id1', 'blob_id2']

Return type twisted.internet.defer.Deferred

set_flags(*blob_id*, *flags*, *namespace=''*)
Set flags for a given blob_id.

Parameters

- **blob_id** (*str*) – Unique identifier of a blob.
- **flags** (*[leap.soledad.common.blobs.Flags]*) – List of flags to be set.
- **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

Returns A deferred that fires when the operation finishes.

Return type twisted.internet.defer.Deferred

Blobs Synchronization

The synchronization part of the `BlobManager` class is implemented in the `BlobsSynchronizer` class, whose API can be seen below.

class `leap.soledad.client._db.blobs.sync.`**BlobsSynchronizer**

fetch_missing(**args*, ***kwargs*)
Compare local and remote blobs and fetch what's missing in local storage.

Parameters **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

refresh_sync_status_from_server(**args*, ***kwargs*)

send_missing(**args*, ***kwargs*)
Compare local and remote blobs and send what's missing in server.

Parameters **namespace** (*str*) – Optional parameter to restrict operation to a given namespace.

sync(**args*, ***kwargs*)

sync_progress

Blobs Errors

This module contains the different errors that can happen when dealing with blobs.

exception leap.soledad.client._db.blobs.errors.**BlobAlreadyExistsError**(*message=None*)
Raised on attempts to put local or remote blobs that already exist in storage.

exception leap.soledad.client._db.blobs.errors.**BlobNotFoundError**(*message=None*)
Raised on attemtps to get remote blobs that do not exist in storage.

exception leap.soledad.client._db.blobs.errors.**InvalidFlagsError**(*message=None*)
Raised on attempts to set invalid flags for remotelly stored blobs.

exception leap.soledad.client._db.blobs.errors.**MaximumRetriesError**
Raised when the maximum number of transfer retries has been reached.

exception leap.soledad.client._db.blobs.errors.**RetriableTransferError**
Raised for any blob transfer error that is considered retriable.

2.6.4 Blobs Server-side HTTP API

Soledad Server provides two different REST APIs for interacting with blobs:

- A *Public Blobs HTTP API*, providing the *Blobs* service for Soledad Client (i.e. actual users of the infrastructure).
- A *Local Incoming Box HTTP API*, providing the delivery part of the *Incoming Box* service, currently used for the MX mail delivery.

Authentication is handled differently for each of the endpoints, see *Authentication* for more details.

Public Blobs HTTP API

The *public endpoint* provides the following REST API for interacting with the *Blobs* service:

path	method	action	accepted query string fields
/blobs/{uuid}	GET	Get a list of blobs. filtered by a flag.	namespace, filter_flag, order_by
/blobs/{uuid}/{blob_id}	GET	Get the contents of a blob.	namespace
/blobs/{uuid}/{blob_id}	PUT	Create a blob. The content of the blob should be sent in the body of the request.	namespace
/blobs/{uuid}/{blob_id}	POST	Set the flags for a blob. A list of flags should be sent in the body of the request.	namespace
/blobs/{uuid}/{blob_id}	DELETE	Delete a blob.	namespace

When listing blobs, results can be filtered and ordered by using different query string parameters. See the table above for accepted query string fields for each action.

The Blobs service supports *namespaces*. All requests can be modified by the `namespace` query string parameter, and the results will be restricted to a certain namespace. When no namespace explicitelly given, the `default` namespace is used.

Listing results can be filtered by flags using the `filter_flag` query string parameter. Currently valid values for flags `PENDING`, `PROCESSING`, `PROCESSED`, and `FAILED`.

Also, results can be ordered by date using the `order_by` query string parameters. The possible values for `order_by` are `date` or `+date` for increasing order, or `-date` for decreasing order.

Local Incoming Box HTTP API

The *local endpoint* provides the following REST API for interacting with the *Incoming Box* service.

path	method	action
/incoming/{uuid}/{blob_id}	PUT	Create an incoming blob. The content of the blob should be sent in the body of the request.

All blobs created using this API are inserted under the namespace `MX` and flagged as `PENDING`.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

Symbols

B

C

D

F

G

I

L

M

P

R

S

U