pyramid_sqlalchemy_sessions Documentation

Release 0.1

Andrey Tretyakov

Sep 11, 2018

Contents:

CHAPTER 1

Getting Started

1.1 Introduction

`pyramid_sqlalchemy_sessions` is a Pyramid framework add-on library providing a *session* implementation using SQLAlchemy as a storage backend.

Session data is stored in the database and is fully transactional. Session cookie only contains randomly generated session ID required to refer the DB entry and is fully encrypted in AES-GCM mode using PyCryptodome library.

The library features are fully modularized, and you are only paying for what you are using.

The library aims to provide secure solution by default, and to use best security practices.

1.2 Why would you (not) need this library

You may need this library if:

- You need ability to store session data server-side, for security or any other reasons
- You need reliable session data storage. As we depend on SQLAlchemy, you can try to use any ACID-compatible DB engine if it's supported by SQLAlchemy.
- You want to store important data in the session. Valid usecases include things like: authentication data, online store shopping cart, multi-step form wizard state, user preferences, etc.
- You want (or don't mind) your session data to be transactional. The library will use same dbsession as your app and will automatically join all transactions. So you can be sure that any `ROLLBACK` for your main data will not leave inconsistent session data.

You may skip this library if:

- you prefer lightweight solutions even if it compromises security or features. In this case, cookie-based session backend is a better pick.
- you want to store not very important information or even throw-away data

- you require that session data should be always saved, regardless of transaction results, e.g. if you collect statistics. This is a big No to this library.
- you don't care about transactions, data reliability and don't mind to lose session data from time to time. In this case you could pick a memory-based session backend, like pyramid_redis_sessions. or pyramid_session_redis

---

Note: Without a server-side backend it's impossible to securely terminate any session, as cookie-based solutions rely on gentleman agreement to "forget" the cookie, which can't be enforced.

1.3 Before you begin

The library will assume the following:

- You are using SQLAlchemy as a data storage backend. The library tries to use portable solutions as much as possible, but the author does not have ability to test every engine out there, especially proprietary ones. So for a start we can say that PostgreSQL and MySQL-family (MariaDB, etc) are supported. SQLite works but it's main purpose is to run test cases as generally it has poor support for concurrency and transactions.
- Your database and SQLALchemy `engine` are configured to work in `SERIALIZABLE` transaction isolation mode. It's the best mode to avoid any data anomalies and if the DB implements optimistic locking such as MVCC, is also best for performance: avoid excessive locks but be ready to retry the transaction (basically what `pyramid_retry` is doing).
- You are using `pyramid_tm` to manage your transactions. Transaction will span the whole request, without any manual commits by the developer. It's important as breaking this workflow could break the whole library. Savepoints compatibility haven't been tested yet.
- You don't clear your session by running `dbsession.expunge_all()`, etc. As the library will share the DB session with your app, both your main data and the library data need to coexist peacefully.

  Note: It's possible to use a separate session for the library, as generally the library can't distinguish *right* sessions from *wrong* ones, but such configuration haven't been tested and is not supported at the moment.

- Since Pyramid 1.9, the library will assume you are using `pyramid_retry` to retry failed transactions. Retrying is not required technically, but in most cases you would want to retry instead of showing 500 page to the user, so it's a welcomed feature.
- Your code expects that your *session data* won't be always committed to the DB. For example, in Pyramid you can *raise* or *return* HTTP exceptions. For an app the difference between the two is not always significant, but for the library it is huge: *raising* a seemingly safe `pyramid.httpexceptions.HTTPFound` will always `ROLLBACK` the transaction, even while this type of response is successful. Inside `pyramid_tm` there are some tweaks for what is a success or not, but generally you want to avoid exceptions if you can, if you want your *session data* to be committed at all.

Make sure your app configuration includes the following line:

```
tm.annotate_user = False
```

Annotations can cause problems with the library, as it may start a premature transaction before `pyramid_tm` has begun.

Also using explicit transaction manager by setting `tm.manager_hook` as described in `pyramid_tm` docs is recommended.

1.4 Quick Start

Let's configure a minimal session. We will assume you created a project using `pyramid-cookiecutter-alchemy` cookiecutter, and your DB session is available as `request.dbsession`.

Create `session.py` file in your `models` subpackage and add the following lines:

```
from pyramid_sqlalchemy_sessions import BaseMixin
# Using default declarative Base provided by the cookiecutter.
from .meta import Base


class Session(BaseMixin, Base):
    __tablename__ = 'session'
```

Import your new model in the `__init__.py` of your models subpackage:

```
from .session import Session
```

and initialize the db using the script generated by the cookiecutter.

Then, start a python shell and run:

```
>>> from pyramid_sqlalchemy_sessions import generate_secret_key
>>> generate_secret_key()
```

Copy the generated key (without surrounding single quotes) to clipboard. Add the following settings to the `[app:main]` section of your configuration file:

```
session.secret_key = paste your generated key here
session.model_class = yourproject.models.session.Session
```

And finally, include the library configuration in your project main `__init__.py` file:

```
def main(global_config, **settings):
    config = Configurator(settings=settings)
    config.include('pyramid_sqlalchemy_sessions')
    config.scan()
    return config.make_wsgi_app()
```

Now unless you have some conflict in your configuration or you did a mistake, the session should be working.

CHAPTER 2

Detailed features guide

2.1 Basic Session Usage

You can always use basic session features dictated by the `pyramid.interfaces.ISession` API:

- store pickle-serializable data in the session dict
- store and fetch flash messages

Note: Unlike default cookie-based session implementations provided by the Pyramid, the library does not store flash messages in the main session dict. So for example `session.clear()` will not clear the messages.

2.2 Idle Timeout

The feature implements OWASP "Idle Timeout" security policy. With this feature enabled, user's session will expire after `idle_timeout` seconds has passed since his last session activity.

For this feature to work properly, every request accessing session has to *extend* it, i.e. to update it's expiration timestamp. This is one of the complaints related to DB-based session backends: traditional relational databases don't like too many writes because of locking, slow discs, replication, etc.

At minimum, any session write (when *session data* is put or changed inside the session) will extend it. But if the code processing the request only reads the session, that's when we can optimize the DB performance using dedicated settings:

- `extension_delay` allows to lower the frequency of extensions, i.e. db writes. Session reads will not extend the session sooner than `extension_delay` since last extension.
- `extension_chance` allows to randomize the extensions by session reads. It's a percentage-based chance to extend: every time an extension could happen (including the requirement to pass `extension_delay` if the latter was enabled) a dice will be rolled to decide if the extension should happen.

This setting is experimental. It's main purpose is to deal with very specific and not very common scenario: concurrent requests using same session. Concurrent writes to same rows can cause performance issues because of row locks and serialization conflicts. Note that this setting affects all requests, not only parallel ones, so it has to be applied very carefully (if at all).

- `extension_deadline` allows to limit the randomness of the extension, when `extension_chance` is lower than 100. Upon reaching the `extension_deadline` since last extension, next session read will always extend, as if `extension_chance` was set to 100.

The most important side effect of these 3 settings is they affect error margin when calculating idle timeout: sessions will be expired earlier than should have been. If, and to what extent it is acceptable is for you to decide.

2.3 Runtime-configurable Idle Timeout

Same as *Idle Timeout*, but allows to use different feature settings per session. See *Working with settings* for details.

2.4 Absolute Timeout

The feature implements OWASP "Absolute Timeout" security policy. With this feature enabled, user's session will expire after `absolute_timeout` seconds has passed since creation of the session, regardless of any session activity.

2.5 Runtime-configurable Absolute Timeout

Same as *Absolute Timeout*, but allows to use different feature settings per session. See *Working with settings* for details.

2.6 Renewal Timeout

The feature implements OWASP "Renewal Timeout" security policy. With this feature enabled, user's session will periodically run *renewal* procedure. The procedure can be described as following:

1. Generate random renewal ID in addition to the main ID on session creation
2. Wait until `renewal_timeout` seconds has passed since creation (or last renewal).
3. Upon reaching the timeout, try to *renew* the session by generating a candidate renewal ID and sending it to the user.
4. Wait until we receive acknowledgement - session cookie with the candidate ID. If there's no acknowledgement, try again after `renewal_try_every` seconds has passed since the last renewal try. Until acknowledgement is received, the old renewal ID is valid.
5. When user sends a session cookie containing the candidate ID, delete old renewal ID and use the candidate instead. At this moment renewal is finished.
6. After this, if an old (or any otherwise unknown) renewal id is received, invalidate the session.

The purpose of the renewal is to limit the time an attacker could use stolen session cookie (assuming the theft happened once and the attacker can't access newly issued cookies). Also this protocol allows to detect the fact of theft itself, when both the attacker and the user try to use the same session. We may not know who is who, but we certainly know that there are 2 versions of the same cookie and one of them is invalid.

2.7 Runtime-configurable Renewal Timeout

Same as *Renewal Timeout*, but allows to use different feature settings per session. See *Working with settings* for details.

2.8 Runtime-configurable cookie settings

The feature allows to use different cookie settings per session. See *Working with settings* for details.

2.9 Userid

Pyramid framework provides `pyramid.authentication.SessionAuthenticationPolicy` that stores user ID in the session. The problem is that the interaction between the policy and the session is not explicit: user ID is treated like any other session dict key. While we could always treat the user ID key as a special guest, explicit interaction is a much better idea:

```
# Read
who = request.session.userid
# Write
request.session.userid = 123
# What could happen when you "forget" the user.
request.session.userid = None
```

The feature allows to *explicitly* associate sessions with users:

1. User ID is stored in a dedicated session table column. This brings some important advantages:
   - you can query sessions by user. For example, you can invalidate all sessions of a user, or show the user his "login sessions".
   - you can eager-load additional data the current view may require. Just configure some eager-loading relationships on your model and some of your views will only run a single query per request.
2. The library provides *`UserSessionAuthenticationPolicy`* that uses the explicit API of this feature.

Note: The library will not register *`UserSessionAuthenticationPolicy`* as the authentication policy automatically. You have to do it yourself.

2.10 CSRF

The feature allows to store CSRF token in the dedicated session table column. You can work with it using `session.new_csrf_token()` and `session.get_csrf_token()` methods.

Note: CSRF session API has been deprecated since Pyramid 1.9, but in case you need it, you can still use this optional feature.

CHAPTER 3

Configuration guide

3.1 Overview

You can configure session factory in 2 steps:

1. Pick desired *session features* and add corresponding *SQL Alchemy ORM Classes (Mixins)* to the bases list of your *model*.

   The library will detect mixin combination and enable corresponding session features.

   Note: It's much better to exclude mixins of features that you are not planning to use:

   - your database will save some resources
   - the library will run a bit less code
   - if you accidentally try to enable a timeout setting that depends on a missing mixin, you will get explicit error at startup instead of a runtime error.

2. Apply session settings - globally, or per-session (if the setting is configurable at runtime). Global settings are provided at startup, and per-session settings use global settings as defaults. In case global settings are not provided, library defaults will be used as globals. (see *Configuration settings reference*)

Note: Timeout-related features have settings also deciding if the feature is enabled or not:

- `idle_timeout`
- `absolute_timeout`
- `renewal_timeout`

For these features, even if corresponding mixin is included, the feature will not work when the setting value is `None`.

3.2 Model configuration

In the following example we configure a session storing user ID, having non-runtime-configurable Absolute Timeout and runtime-configurable Idle Timeout. The example also showcases:

- ability to customize model columns - we use UUID as User ID instead of the default integer.
- eager-loading of the user object.

```
from sqlalchemy import (
  Column,
  ForeignKey,
)
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.orm import relationship
from pyramid_sqlalchemy_sessions import (
    BaseMixin,
    UseridMixin,
    AbsoluteMixin,
    ConfigIdleMixin,
)
# Using default declarative Base provided by the cookiecutter.
from .meta import Base

class Session(
    UseridMixin,
    AbsoluteMixin,
    ConfigIdleMixin,
    BaseMixin,
    Base,
    ):
    __tablename__ = 'session'

    userid = Column(UUID(as_uuid=True), ForeignKey('user.id'))
    # user instance loaded automatically when user is logged in.
    user = relationship('User', backref='sessions', lazy='joined')
```

Note: Runtime-configurable features mixins subclass their non-configurable versions, so you don't need to include both.

Tip: Don't forget to add DB indexes to your session table! The library doesn't provide one, as it's difficult to create universal index solution for all mixin configurations and different DB engines.

3.3 Working with settings

Some settings only meant to be set once and forgotten, such as `cookie_name` or `dbsession_name`. But most other settings are accessable and even configurable at runtime (if the corresponding session model mixin is enabled).

You can access current session settings using the settings object:

```
# Read
idle_timeout = request.session.settings.idle_timeout
```

(continues on next page)

(continued from previous page)

```
# Settings is also a dict-like object.
absolute_timeout = request.session.settings['absolute_timeout']
```

By default you can only read the settings. But when you enable a runtime-configurable feature, it's settings can be changed also:

```
# Suppose configurable cookie settings feature is enabled.
# You need to put settings in editable mode first.
request.session.settings.edit()
request.session.settings.cookie_max_age = 12345
# When you are done, you need to save it.
# Settings are always validated before saving.
request.session.settings.save()
# You can use settings as context manager so that edit and save
# is called automatically
with request.session.settings as s:
    s['cookie_max_age'] = 54321
```

Note: Currently changing of settings works only for a *new session*, otherwise you will get a *SettingsError* exception.

Note: The session implementation provided by the library is *lazy*, and will not persist *clean session*, so any changes of settings for such sessions also won't be persisted in the DB.

3.4 Configuration settings reference

3.4.1 Required settings

secret_key [str] This setting is required by default *serializer* when the library `includeme()` function runs.

Not meant to be accessible at runtime.

serializer [object] Controls what *serializer* to use. Only needed if you want to configure *session factory* manually and to skip the `includeme()`.

Not meant to be accessible at runtime.

model_class [class] Controls what *model* to use to store session data in the DB. Should be a dotted Python name referencing the class (if provided by default way of configuration, e.g. through the ini file) or the class object itself (may need to use this option if you are configuring the session factory manually).

Not meant to be accessible at runtime.

3.4.2 Optional settings (settings with library defaults)

dbsession_name [str] Session code will try to access SQLAlchemy session as an attribute of request using this name.

Not meant to be accessible at runtime.

Default: `dbsession`

cookie_name [str] Name of the session cookie (will appear in `Cookie` and `Set-Cookie` headers). See WebOb and **RFC 6265** for details.

Not meant to be accessible at runtime.

Default: `session`

cookie_max_age [int or None] How long the browser will store the cookie. `None` is for non-persistent cookie. See WebOb and **RFC 6265** for details.

Default: `None`

cookie_path [str] Path of the session cookie. Can be a valid path only (starting with `/`). See WebOb and **RFC 6265** for details.

Default: `/`

cookie_domain [str or None] Domain of the session cookie. See WebOb and **RFC 6265** for details.

Default: `None`

cookie_secure [bool] Boolean flag instructing the browser to send cookie in HTTPS mode only. See WebOb and **RFC 6265** for details.

Default: `False`

cookie_httponly [bool] Boolean flag instructing the browser to prevent scripts accessing the cookie. See WebOb and **RFC 6265** for details.

Default: `True`

idle_timeout [int or None] Controls idle timeout value. See *Idle Timeout* for detailed explanation.

Default: `None`

absolute_timeout [int or None] Controls absolute timeout value. See *Absolute Timeout* for detailed explanation.

Default: `None`

renewal_timeout [int or None] Controls renewal timeout value. See *Renewal Timeout* for detailed explanation.

Default: `None`

renewal_try_every [int] When *Renewal Timeout* feature is working, the library will try to *renew* the session every `renewal_try_every` seconds until success. See *Renewal Timeout* for detailed explanation.

Default: 5

extension_delay [int or None] When *Idle Timeout* feature is working, the library will not try hard to extend the session more often than every `extension_delay` seconds. See *Idle Timeout* for detailed explanation.

Default: `None`

extension_chance [int] When *Idle Timeout* feature is working, the library will *extend* the session randomly, using `extension_chance` chance (in percents). See *Idle Timeout* for detailed explanation.

Default: 100

extension_deadline [int] When *Idle Timeout* feature is working, and `extension_chance < 100` the library will *extend* the session after reaching the `extension_deadline` timeout, as if `extension_chance` was 100. See *Idle Timeout* for detailed explanation.

Default: 1

CHAPTER 4

DB maintenance

At the moment the library does not have any DB migrations code. You are responsible for taking care of your DB schema if your model changes.

The library will delete any expired or otherwise invalid session on the first encounter - when receiving the session cookie. However, a session could expire without the server encountering it again and it's a common situation with bots as they don't care about cookies at all. To deal with this problem the library has DB maintenance procedure that will remove expired sessions from the DB. It's provided in the form of **`pyramid_session_gc`** commandline script. You can run it as the following:

```
pyramid_session_gc <config_uri>
```

The script will load config provided by `config_uri` argument and use it's settings to access the DB.

You can run it as often as you want using a scheduler of your choice.

Note: Special care must be taken when switching global settings on and off without removing existing session rows - it's developer's duty to process the data so that the library code is not confused. It's recommended to delete existing sessions if possible, when changing global settings, unless you *really* know what you are doing.

CHAPTER 5

API Reference

Note: All library API is importable from the root level.

5.1 Configuration

pyramid_sqlalchemy_sessions.config.**factory_args_from_settings**(*settings*, *maybe_dotted*, *prefix='session.'*)

Convert configuration (ini) file settings to a defaults-applied dict suitable as *get_session_factory()* function arguments. Only validates secret key and model class settings - full validation happens inside the *get_session_factory()* function.

Arguments:

settings dictionary of Pyramid app settings (**required**)

maybe_dotted a callable to resolve dotted Python name to a full class (**required**)

prefix settings names prefix

Returns dictionary of settings, suitable as args for the *get_session_factory()* function.

Raises *ConfigurationError* if secret_key or model_class settings are invalid

pyramid_sqlalchemy_sessions.config.**generate_secret_key**(*size=32*)

Generate a random secret key as a string suitable for configuration files. Size is secret key size in bytes.

pyramid_sqlalchemy_sessions.session.**get_session_factory**(*serializer*, *model_class*, ***kw*)

Return *session factory* constructed using settings provided by the arguments.

Arguments:

serializer a *serializer* object (**required**)

model_class session *model* object (**required**)

Other keyword arguments are optional (using library defaults when not provided). See *Configuration settings reference* for details.

class pyramid_sqlalchemy_sessions.authn.**UserSessionAuthenticationPolicy**(*callback=None*, *debug=False*)

Authentication policy storing user ID in the *session*. Similar to `pyramid.authentication.SessionAuthenticationPolicy`, with some differences:

- uses explicit *Userid* feature and will only work with session storage implementation from the `pyramid_sqlalchemy_sessions` package
- doesn't need a prefix argument, as the ID is stored explicitly in a dedicated DB column

5.2 SQL Alchemy ORM Classes (Mixins)

class pyramid_sqlalchemy_sessions.model.**BaseMixin**
Base session ORM class mixin. Subclass this mixin to get a minimal working session without any extra features.

class pyramid_sqlalchemy_sessions.model.**FullyFeaturedSession**
Class providing all features of all mixins together. Use it if you are really using all features, or if you don't care about running dead code or having unused columns in the DB.

class pyramid_sqlalchemy_sessions.model.**UseridMixin**
Mixin that enables *Userid* feature.

class pyramid_sqlalchemy_sessions.model.**CSRFMixin**
Mixin that enables *CSRF* feature.

class pyramid_sqlalchemy_sessions.model.**IdleMixin**
Mixin that enables *Idle Timeout* feature.

class pyramid_sqlalchemy_sessions.model.**AbsoluteMixin**
Mixin that enables *Absolute Timeout* feature.

class pyramid_sqlalchemy_sessions.model.**RenewalMixin**
Mixin that enables *Renewal Timeout* feature.

class pyramid_sqlalchemy_sessions.model.**ConfigCookieMixin**
Mixin that enables *Runtime-configurable cookie settings* feature.

class pyramid_sqlalchemy_sessions.model.**ConfigIdleMixin**
Mixin that enables *Runtime-configurable Idle Timeout* feature.

class pyramid_sqlalchemy_sessions.model.**ConfigAbsoluteMixin**
Mixin that enables *Runtime-configurable Absolute Timeout* feature.

class pyramid_sqlalchemy_sessions.model.**ConfigRenewalMixin**
Mixin that enables *Runtime-configurable Renewal Timeout* feature.

5.3 Events

class pyramid_sqlalchemy_sessions.events.**InvalidCookieErrorEvent**(*request*, *exception=None*)

Pyramid event. Fired when InvalidCookieError is catched

class pyramid_sqlalchemy_sessions.events.**CookieCryptoErrorEvent**(*request*, *exception=None*)

Pyramid event. Fired when CookieCryptoError is catched

class pyramid_sqlalchemy_sessions.events.**RenewalViolationEvent**(*request*, *exception=None*)

Pyramid event. Fired when received cookie contains invalid renewal id, which could be a sign of a stolen session cookie or abnormal browser behavior such as using old cookies restored from a backup.

5.4 Exceptions

exception pyramid_sqlalchemy_sessions.exceptions.**ConfigurationError**

Raised when the session factory has been incorrectly configured.

exception pyramid_sqlalchemy_sessions.exceptions.**CookieCryptoError**

Raised by serializer when session cookie can't be decrypted and/or authenticated. Could be a sign of a system problem, user tampering with the cookie, or secret key mismatch.

The library will catch this exception to avoid breaking normal flow of the application. You can subscribe to *CookieCryptoErrorEvent* event if you want to run additional procedures when it happens.

exception pyramid_sqlalchemy_sessions.exceptions.**InconsistentDataError**

Raised when inconsistent session data has been found in the DB, which could be a sign of incorrect DB manipulations or misconfiguration.

exception pyramid_sqlalchemy_sessions.exceptions.**InvalidCookieError**

Raised by serializer when session cookie is invalid prior to decryption/deserializing. Could be a sign of a system problem or user tampering with the cookie.

The library will catch this exception to avoid breaking normal flow of the application. You can subscribe to *InvalidCookieErrorEvent* event if you want to run additional procedures when it happens.

exception pyramid_sqlalchemy_sessions.exceptions.**SettingsError**

Runtime settings errors not related to incorrect settings values. Incorrect settings values raise ValueError instead.

CHAPTER 6

Glossary of terms

session In the context of web applications, a temporary storage of information related to the current user. In the Pyramid framework it's usually an object implementing `pyramid.interfaces.ISession`

session factory A callable returning *session* object. In the Pyramid framework it's usually an object implementing `pyramid.interfaces.ISessionFactory`

serializer An object with `dumps` and `loads` methods, packing and unpacking data to/from a cookie value.

model SQLAlchemy ORM model class, subclassing *declarative Base* and selected model mixins. In the context of this library, a model is a class referenced by the `model_class` setting.

new session Session is new when it hasn't been saved (i.e. committed) to the database yet. You can get a new session when you start it, or after you invalidate a session.

session data The main purpose of session is to store useful data. Examples of such data in the library include:

- any session dict values
- flash messages
- user ID with *Userid* feature enabled
- CSRF token with *CSRF* feature enabled
- any internal data the libary may need to save (not exposed to the developer)

Note: Session settings are metadata, not the data.

---

lazy session Session is called lazy if it is not saved without any data. The library session is lazy: you need to store *session data* to make it *dirty* and to save it in the database.

clean session Session not containing any *session data*.

dirty session A new session having uncommitted *session data*, or an existing session having any uncommitted data.

session extension A process of updating session idle expiration timestamp, or in other words, applying idle timeout using the current time as a base. Happens once per request.

session renewal A procedure that includes rotating a separate randomly generated renewal ID, described in detail in *Renewal Timeout*

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

U