Meepo Documentation

Release 0.1.9

lxyu

March 24, 2015

Contents

1 Features **3**

2 Installation **5**

3 Usage **7**

4 Pub Concept **9**
4.1 MySQL Pub 9
4.2 SQLAlchemy Pub 10

5 Meepo Sub **13**
5.1 Dummy Sub 13
5.2 0MQ Sub 13

6 Applications **15**
6.1 EventSourcing 15
6.2 Replicator 20

Python Module Index **23**

Welcome to meepo's documentation. Meepo is a event sourcing and broadcasting platform for database.

This documentation consists of two parts:

1. Meepo PubSub (*meepo.pub* & *meepo.sub*). This part is enough if you only needs a simple solution for your database events.
2. Meepo Apps (*meepo.apps*). This part ships with eventsourcing and replicator apps for advanced use. You can refer to examples for demo.

Meepo source code is hosted on Github: https://github.com/eleme/meepo

- Features
- Installation
- Usage
- Pub Concept
  - MySQL Pub
  - SQLAlchemy Pub
- Meepo Sub
  - Dummy Sub
  - 0MQ Sub
- Applications
  - EventSourcing
  - Replicator

CHAPTER 1

Features

Meepo can be used to do lots of things, including replication, eventsourcing, cache refresh/invalidate, real-time analytics etc. The limit is all the tasks should be row-based, since meepo only gives `table_action` -> `pk` style events.

- Row-based database replication.

  Meepo can be used to replicate data between databases including postgres, sqlite, etc.

  Refer to `examples/repl_db` script for demo.

- Replicate RDBMS to NoSQL and search engine.

  Meepo can also be used to replicate data changes from RDBMS to redis, elasticsearch etc.

  Refer to `examples/repl_redis` and `examples/repl_elasticsearch` for demo.

- Event Sourcing.

  Meepo can log and replay what has happened since some time using a simple event sourcing.

  Refer to `examples/event_sourcing` for demo.

Note: Meepo can only replicate row based data, which means it DO NOT replicate schema changes, or bulk operations.

CHAPTER 2

Installation

Requirements **Python 2.x >= 2.7** or **Python 3.x >= 3.2** or **PyPy**

To install the latest released version of Meepo:

```
$ pip install meepo
```

CHAPTER 3

Usage

Meepo use blinker signal to hook into the events of mysql binlog and sqlalchemy, the hook is very easy to install.

Hook with MySQL's binlog events:

```
from meepo.pub import mysql_pub
mysql_pub(mysql_dsn)
```

Hook with SQLAlchemy's events:

```
from meepo.pub import sqlalchemy_pub
sqlalchemy_pub(session)
```

Then you can connect to the signal and do tasks based the signal:

```
sg = signal("test_write")

@sg.connect
def print_test_write(pk)
    print("test_write -> %s" % pk)
```

Try out the demo scripts in `example/tutorial` for more about how meepo event works.

CHAPTER 4

Pub Concept

Meepo's core concept is based event pubs, it follows mysql row-based binlog and sqlalchemy events system and shape them into `table_action pk` format signals.

Currently there are 2 pubs implemented: `mysql_pub` and `sqlalchemy_pub`.

The publishers and subscribers are connected with `blinker.signal`.

Publisher sends pk by:

```
signal("table_action").send(pk)
```

And subscriber can accept the pk by:

```
sg = signal("table_action")

@sg.connect
def dummy_print(pk):
    print(pk)
```

4.1 MySQL Pub

The mysql pub will use the `python-mysql-replication` binlog stream as the source of events.

The events pub flow:

```
                                                    +---------------------+
                                                    |                     |
                                                +--->  table_write event  |
                                                |   |                     |
                                                |   +---------------------+
                                                |
+---------------------+      +----------------+ |
|                     |      |                | |   +---------------------+
|        mysql        |      |   meepo.pub    | |   |                     |
|                     +------>                +--+--->  table_update event |
|  row-based binlog   |      |   mysql_pub    | |   |                     |
|                     |      |                | |   +---------------------+
+---------------------+      +----------------+ |
                                                |
                                                |   +---------------------+
                                                |   |                     |
                                                +--->  table_delete event |
```

```
                                   |                      |
                                   +----------------------+
```

meepo.pub.mysql.**mysql_pub**(*mysql_dsn*, *tables=None*, *blocking=False*, ***kwargs*)

MySQL row-based binlog events pub.

General Usage

Listen and pub all tables events:

```
mysql_pub(mysql_dsn)
```

Listen and pub only some tables events:

```
mysql_pub(mysql_dsn, tables=["test"])
```

By default the `mysql_pub` will process and pub all existing row-based binlog (starting from current binlog file with pos 0) and quit, you may set blocking to True to block and wait for new binlog, enable this option if you're running the script as a daemon:

```
mysql_pub(mysql_dsn, blocking=True)
```

The binlog stream act as a mysql slave and read binlog from master, so the server_id matters, if it's conflict with other slaves or scripts, strange bugs may happen. By default, the server_id is randomized by `randint(1000000000, 4294967295)`, you may set it to a specific value by server_id arg:

```
mysql_pub(mysql_dsn, blocking=True, server_id=1024)
```

Signals Illustrate

Sometimes you want more info than the pk value, the mysql_pub expose a raw signal which will send the original binlog stream events.

For example, the following sql:

```
INSERT INTO test (data) VALUES ('a');
```

The row-based binlog generated from the sql, reads by binlog stream and generates signals equals to:

```
signal("test_write").send(1)
signal("test_write_raw").send({'values': {'data': 'a', 'id': 1}})
```

Binlog Pos Signal

The mysql_pub has a unique signal `mysql_binlog_pos` which contains the binlog file and binlog pos, you can record the signal and resume binlog stream from last position with it.

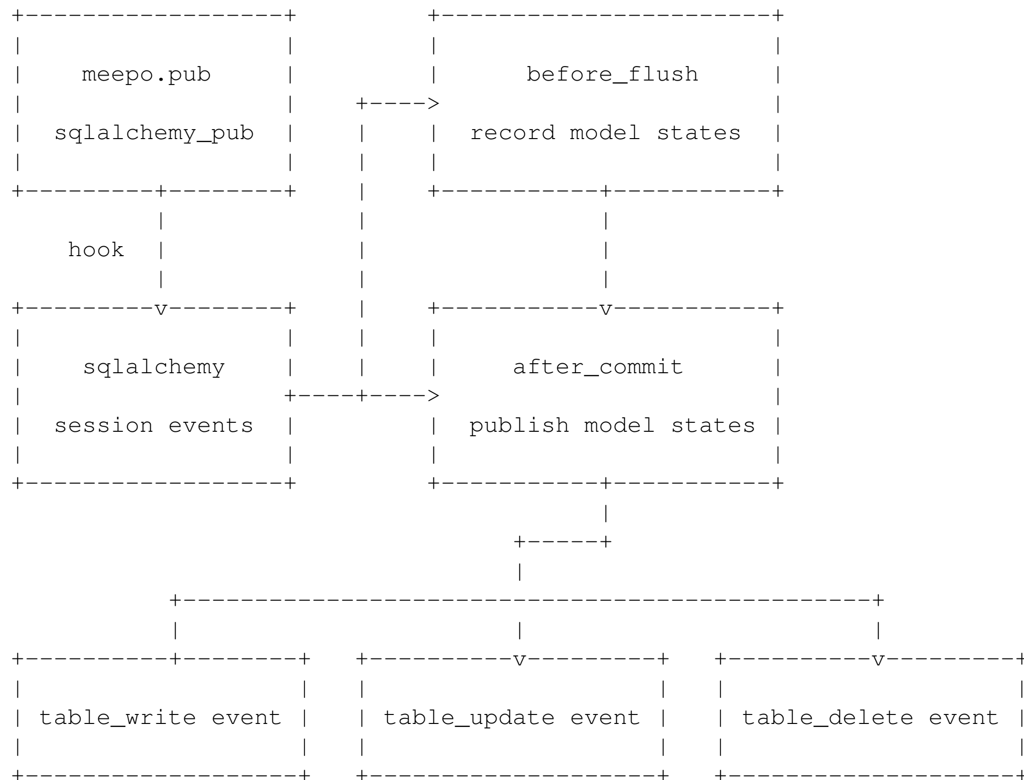
Parameters

- **mysql_dsn** – mysql dsn with row-based binlog enabled.
- **tables** – which tables to enable mysql_pub.
- **blocking** – whether mysql_pub should wait more binlog when all existing binlog processed.
- **kwargs** – more kwargs to be passed to binlog stream.

4.2 SQLAlchemy Pub

The sqlalchemy pub will hook into SQLAlchemy's event system, shape and publish events with `table_action pk` style.

The events pub flow:



class `meepo.pub.sqlalchemy.`**`sqlalchemy_pub`**(*session*, *tables=None*)

SQLAlchemy Pub.

The install method will add 2 hooks on sqlalchemy events system:

- `session_update` -> sqlalchemy - `before_flush`
- `session_commit` -> sqlalchemy - `after_commit`

The `session_update` method need to record the model states in sqlalchemy "before_flush" event, when the session records the status with `session.new`, `session.dirty` and `session.deleted`, these states will be deleted in "after_commit" event.

General Usage

Install the sqlalchemy pub hook by calling it on sqlalchemy session:

```
sqlalchemy_pub(session)
```

Only listen some tables:

```
sqlalchemy_pub(session, tables=["test"])
```

Tables can be added later, the duplicated tables will be automatically merged:

```
pub = sqlalchemy_pub(session)
pub(["table_a", "table_b"])
pub(["table_b", "table_c"])
pub.tables == {"table_a", "table_b", "table_c"}
```

Then use the session as usual and the events will be available.

Signals Illustrate

Sometimes you want more info than the pk value, the sqlalchemy_pub expose a raw signal which will send the original sqlalchemy objects.

For example, this code:

```
class Test(Base):
    __tablename__ = "test"
    id = Column(Integer, primary_key=True)
    data = Column(String)

t_1 = Test(id=1, data='a')
session.add(t_1)
session.commit()
```

Generates signals equal to:

```
signal("test_write").send(1)
signal("test_write_raw").send(t_1)
```

Parameters

- **session** – sqlalchemy session to install the hook
- **tables** – tables to install the hook, leave None to pub all.

Warning: SQLAlchemy bulk operation currently **NOT** supported, so this code won't work:

```
# bulk updates
session.query(Test).update({"data": 'x'})

# bulk deletes
session.query(Test).filter(Test.data == 'x').delete()
```

session_commit(*session*)

Pub the events after the session committed.

This method should be linked to sqlalchemy "after_commit" event.

session_update(*session*, *_)

Record the sqlalchemy object states in the middle of session, prepare the events for the final pub in session_commit.

CHAPTER 5

Meepo Sub

Meepo sub is where all the imagination comes true, all subs implemented here are only some simple demos. Customize your own sub for the real power.

To make use of a signal, just create a function that accepts a primary key.

For example, print an event with:

```
# use weak False here to force strong ref to the lambda func.
signal("test_write").connect(
    lambda pk: logger.info("%s -> %s" % event, pk),
    weak=False
)
```

For advanced use with sqlalchemy, you may also use the raw signal:

```
signal("test_write_raw").connect(
    lambda obj: logger.info("%s -> %s" % event, obj.__dict__),
    weak=False)
)
```

5.1 Dummy Sub

`meepo.sub.dummy.`**`print_sub`**`(`*tables*`)`

Dummy print sub.

Parameters **tables** – print events of tables.

5.2 0MQ Sub

`meepo.sub.zmq.`**`zmq_sub`**`(`*bind*, *tables*, *forwarder=False*, *green=False*`)`

0mq fanout sub.

This sub will use zeromq to fanout the events.

Parameters

- **bind** – the zmq pub socket or zmq device socket.
- **tables** – the events of tables to follow.
- **forwarder** – set to True if zmq pub to a forwarder device.

- **green** – weather to use a greenlet compat zmq

CHAPTER 6

Applications

6.1 EventSourcing

6.1.1 Concept

For basic concept about eventsourcing, refer to http://martinfowler.com/eaaDev/EventSourcing.html

Simple Eventsourcing

The eventsourcing implemented in meepo is a simplified version of es, it only records what has changed since a timestamp, but not the diffs.

So you only get a list of primary keys when query with a timestamp:

```
order_update 102 27 59 43
```

Because event sourcing is hard in distributed system, you can't give a accurate answer of events happening order. So we only keep a record of what happened since some time, then you know the data has gone stale, and you have to retrieve latest data from source and do the tasks upon it.

Why Eventsourcing

Why is eventsourcing needed? Let's check the sqlalchemy_pub events flow:

1. before flush -> record instances states
2. commit transaction in database
3. after commit -> pub signal

So it's possible that the process(or thread or greenlet) somehow being killed right between b and c, then the signal lost.

With prepare commit in event sourcing, the session will be recorded so it's possible to recover from this corrupt state.

But you should note this is a very rare, so in most cases, you don't need this 100% grantee on events, then just use the simple `sqlalchemy_pub()` is enough.

Pub & Sub

class `meepo.apps.eventsourcing.pub.`**`sqlalchemy_es_pub`**(*session*, *tables=None*)

SQLAlchemy EventSourcing Pub.

Add eventsourcing to sqlalchemy_pub, three more signals added for tables:

- `session_prepare`

- `session_commit` / `session_rollback`

The hook will use prepare-commit pattern to ensure 100% reliability on event sourcing.

Multi-Sessions Prepare Commit

The 3 additional signals were attached to sqlalchemy session factory in case of being used in multi-sessions environments.

If you only use one sqlalchemy session in your program, it's fine to use `session_prepare` / `session_commit` as other signals.

But if you use multiple sessions, you can separate the prepare-commit signals by:

- Separate sessions by settings `info` arg in session factory.

  Because the info is the only attributes copied from session factory to session instance.

  `meepo.signals` monkey patched the blinker `hashable_identity` func to use the `session.info` for session hash.

- Provide session as sender when signal receivers connects.

For example:

```
# setting 'info' in sqlalchemy session_factory
SessionA = sessionmaker(bind=engine_a, info={"name": "session_a"})
SessionB = sessionmaker(bind=engine_b, info={"name": "session_b"})

sqlalchemy_es_pub(SessionA)
sqlalchemy_es_pub(SessionB)

sg = signal("session_prepare")

def _sp_for_a(session, event):
    print(session.info)
sg.connect(_sp_for_a, sender=SessionA)
```

Then the `_sp_for_a` will only receive prepare-commit related events triggered by `SessionA`.

`session_commit`(*session*)
Send session_commit signal in sqlalchemy `before_commit`.

This marks the success of session so the session may enter commit state.

`session_prepare`(*session*, *_*)
Send session_prepare signal in session "before_commit".

The signal contains another event argument, which records whole info of what's changed in this session, so the signal receiver can receive and record the event.

`session_rollback`(*session*)
Send session_rollback signal in sqlalchemy `after_rollback`.

This marks the failure of session so the session may enter commit phase.

`meepo.apps.eventsourcing.sub.`**`redis_es_sub`**(*session*, *tables*, *redis_dsn*, *strict=False*, *namespace=None*, *ttl=259200*, *socket_timeout=1*)
Redis EventSourcing sub.

This sub should be used together with sqlalchemy_es_pub, it will use RedisEventStore as events storage layer and use the prepare-commit pattern in `sqlalchemy_es_pub()` to ensure 100% security on events recording.

Parameters

- **session** – the sqlalchemy to bind the signal

- **tables** – tables to be event sourced.
- **redis_dsn** – the redis server to store event sourcing events.
- **strict** – arg to be passed to RedisPrepareCommit. If set to True, the exception will not be silent and may cause the failure of sqlalchemy transaction, user should handle the exception in the app side in this case.
- **namespace** – namespace string or func. If func passed, it should accept timestamp as arg and return a string namespace.
- **ttl** – expiration time for events stored, default to 3 days.
- **socket_timeout** – redis socket timeout.

6.1.2 EventStore

class `meepo.apps.eventsourcing.event_store.`**`RedisEventStore`**(*redis_dsn*, *namespace=None*, *ttl=259200*, *socket_timeout=1*, ***kwargs*)

EventStore based on redis.

The event store use namespace and event name as key and store primary keys using redis sorted set, with event timestamp as score.

General Usage

Init event store with redis_dsn:

```
event_store = RedisEventStore("redis://localhost/", "store")
```

You can also pass a function to namespace, it'll accept timestamp as arg, this can be used to separate events store based on hour, day or week etc.:

```
event_store = RedisEventStore(
    "redis://localhost/", lambda ts: "store:%s" % d(ts, "%Y%m%d"))
```

Add a event with:

```
event_store.add("test_write", 1)
```

Or add a event with timestamp passed in:

```
event_store.add("test_write", 2, ts=1024)
```

Clear all records of an event within a namespace:

```
event_store.clear("test_write")
```

Events Replay

One important feature for eventsourcing is replay, it can replay what has changed and the latest update timestamp of events.

Replay all records of an event within a namespace:

```
event_store.replay("test_write")
```

Or replay all records since timestamp:

```
# all events since timestamp 1024
event_store.replay("test_write", ts=1024)

# all events between timestamp 1024 and now
event_store.replay("test_write", ts=1024, end_ts=time.time())
```

You can also replay all events with it's latest updating time:

```
event_store.replay("test_write", with_ts=True)
```

Events Query

You can query the last change timestamp of an event with query api.

Query records within current namespace:

```
event_store.query("test_write", 1)
```

The return value will either be int timestamp or None if record not exists.

Add a timestamp to query events within other namespace (assume you separate the event store namespace by day, you may want to query event happened yesterday.):

```
event_store.query("test_write", 1, ts=some_value)
```

Note: The redis event store class is compat with twemproxy.

Parameters

- **redis_dsn** – the redis instance uri
- **namespace** – namespace func for event key, the func should accept event timestamp and return namespace of the func. namespace also accepts str type arg, which will always return the same namespace for all timestamps.
- **ttl** – expiration time for events stored, default to 3 days.
- **socket_timeout** – redis socket timeout
- **kwargs** – kwargs to be passed to redis instance init func.

add (*event, pk, ts=None, ttl=None*)

Add an event to event store.

All events were stored in a sorted set in redis with timestamp as rank score.

Parameters

- **event** – the event to be added, format should be `table_action`
- **pk** – the primary key of event
- **ts** – timestamp of the event, default to redis_server's current timestamp
- **ttl** – the expiration time of event since the last update

Returns bool

clear (*event, ts=None*)

Clear all stored record of event.

Parameters

- **event** – event name to be cleared.

- **ts** – timestamp used locate the namespace

query (*event*, *pk*, *ts=None*)
Query the last update timestamp of an event pk.

You can pass a timestamp to only look for events later than that within the same namespace.

Parameters

- **event** – the event name.
- **pk** – the pk value for query.
- **ts** – query event pk after ts, default to None which will query all span of current namespace.

replay (*event*, *ts=0*, *end_ts=None*, *with_ts=False*)
Replay events based on timestamp.

If you split namespace with ts, the replay will only return events within the same namespace.

Parameters

- **event** – event name
- **ts** – replay events after ts, default from 0.
- **end_ts** – replay events to ts, default to "+inf".
- **with_ts** – return timestamp with events, default to False.

Returns list of pks when with_ts set to False, list of (pk, ts) tuples when with_ts is True.

6.1.3 PrepareCommit

Prepare Commit also known as Two-Phase Commit, for basic concept about it, refer to http://en.wikipedia.org/wiki/Two-phase_commit_protocol

The two phase commit feature implemented in meepo is used to make sure event 100% reliably recorded in eventsourcing, and it's not a strict traditional two-phase commit.

Only use it if you need a 100% grantee of not losing any events. The feature should only be used in combination of sqlalchemy_es_pub, which ships with session prepare-commit signals.

class meepo.apps.eventsourcing.prepare_commit.**RedisPrepareCommit** (*redis_dsn*, *strict=False*, *namespace=None*, *ttl=86400*, *socket_timeout=1*, ***kwargs*)
Prepare Commit session based on redis.

This prepare commit records sqlalchemy session, and should be used with `sqlalchemy_es_pub()`.

Parameters

- **redis_dsn** – the redis instance uri
- **strict** – by default the exceptions happened in middle of prepare-commit will only be caught and logged as error, but the process continue to execute. If strict set to True, the exception will be raised to outside.
- **namespace** – namespace string or namespace func. if func passed, it should accepts timestamp as arg and return string namespace.

- **ttl** – expiration time for events stored, default to 1 day.
- **socket_timeout** – redis socket timeout
- **kwargs** – kwargs to be passed to redis instance init func.

clear(*ts=None*)
Clear all session in prepare phase.

Parameters **ts** – timestamp used locate the namespace

commit(*session*)
Commit phase for session.

Parameters **session** – sqlalchemy session

phase(*session*)
Determine the session phase in prepare commit.

Parameters **session** – sqlalchemy session

Returns phase "prepare" or "commit"

prepare(*session*, *event*)
Prepare phase for session.

Parameters **session** – sqlalchemy session

prepare_info(*ts=None*)
Return all session unique ids recorded in prepare phase.

Parameters **ts** – timestamp, default to current timestamp

Returns set of session unique ids

rollback(*session*)
Commit phase for session.

Parameters **session** – sqlalchemy session

session_info(*session*)
Return all session unique ids recorded in prepare phase.

Parameters **ts** – timestamp, default to current timestamp

Returns set of session unique ids

6.2 Replicator

Meepo Replicators based on events.

class meepo.apps.replicator.**QueueReplicator**(**args*, ***kwargs*)
Replicator using Queue as worker task queue

This Replicator receives events from upstream zmq devices and put them into a set of python multiprocessing queues using ketama consistent hashing. Each queue has a worker. We use `WorkerPool` to manage a set of queues.

event(**topics*, ***kwargs*)
Topic callback registry.

callback func should receive two args: topic and pk, and then process the replication job.

Note: The callback func must return True/False. When passed a list of pks, the func should return a list of True/False with the same length of pks.

Parameters

- **topics** – a list of topics
- **workers** – how many workers to process this topic
- **multi** – whether pass multiple pks
- **queue_limit** – when queue size is larger than the limit, the worker should run deduplicate procedure

run()

Run the replicator.

Main process receive messages and distribute them to worker queues.

class meepo.apps.replicator.**RqReplicator**(**args*, ***kwargs*)

Replicator suitable for rq task queue.

For example: >>> rq_repl = RqReplicator("tcp://127.0.0.1:4000") >>> @rq_repl.event("a_table_update") >>> def job_test(pks): >>> q = rq.Queue("update_cache:a_table") >>> q.enqueue("module.jobs.func", pks) >>> rq_repl.run()

Rq queue should be created in the external code.

In fact this replicator can be generally used. It will pass pks as argument to the supplied callback func and the func can do anything you want.

The callback func should always accept a list of primary keys.

Python Module Index

m

meepo.apps.eventsourcing, 15
meepo.apps.eventsourcing.event_store, 17
meepo.apps.eventsourcing.prepare_commit, 19
meepo.apps.eventsourcing.pub, 15
meepo.apps.eventsourcing.sub, 16
meepo.apps.replicator, 20
meepo.pub, 9
meepo.pub.mysql, 9
meepo.pub.sqlalchemy, 10
meepo.sub, 13
meepo.sub.dummy, 13
meepo.sub.zmq, 13

A

add() (meepo.apps.eventsourcing.event_store.RedisEventStore method), 18

C

clear() (meepo.apps.eventsourcing.event_store.RedisEventStore method), 18
clear() (meepo.apps.eventsourcing.prepare_commit.RedisPrepareCommit method), 20
commit() (meepo.apps.eventsourcing.prepare_commit.RedisPrepareCommit method), 20

E

event() (meepo.apps.replicator.QueueReplicator method), 20

M

meepo.apps.eventsourcing (module), 15
meepo.apps.eventsourcing.event_store (module), 17
meepo.apps.eventsourcing.prepare_commit (module), 19
meepo.apps.eventsourcing.pub (module), 15
meepo.apps.eventsourcing.sub (module), 16
meepo.apps.replicator (module), 20
meepo.pub (module), 9
meepo.pub.mysql (module), 9
meepo.pub.sqlalchemy (module), 10
meepo.sub (module), 13
meepo.sub.dummy (module), 13
meepo.sub.zmq (module), 13
mysql_pub() (in module meepo.pub.mysql), 10

P

phase() (meepo.apps.eventsourcing.prepare_commit.RedisPrepareCommit method), 20
prepare() (meepo.apps.eventsourcing.prepare_commit.RedisPrepareCommit method), 20
prepare_info() (meepo.apps.eventsourcing.prepare_commit.RedisPrepareCommit method), 20
print_sub() (in module meepo.sub.dummy), 13

Q

query() (meepo.apps.eventsourcing.event_store.RedisEventStore method), 19
QueueReplicator (class in meepo.apps.replicator), 20

R

redis_es_sub() (in module meepo.apps.eventsourcing.sub), 16
RedisEventStore (class in meepo.apps.eventsourcing.event_store), 17
RedisPrepareCommit (class in meepo.apps.eventsourcing.prepare_commit), 19
replay() (meepo.apps.eventsourcing.event_store.RedisEventStore method), 19
rollback() (meepo.apps.eventsourcing.prepare_commit.RedisPrepareComm method), 20
RqReplicator (class in meepo.apps.replicator), 21
run() (meepo.apps.replicator.QueueReplicator method), 21

S

session_commit() (meepo.apps.eventsourcing.pub.sqlalchemy_es_pub method), 16
session_commit() (meepo.pub.sqlalchemy.sqlalchemy_pub method), 12
session_info() (meepo.apps.eventsourcing.prepare_commit.RedisPrepareCo method), 20
session_prepare() (meepo.apps.eventsourcing.pub.sqlalchemy_es_pub method), 16
session_rollback() (meepo.apps.eventsourcing.pub.sqlalchemy_es_pub method), 16
session_update() (meepo.pub.sqlalchemy.sqlalchemy_pub method), 12
sqlalchemy_es_pub (class in meepo.apps.eventsourcing.pub), 15
sqlalchemy_pub (class in meepo.pub.sqlalchemy), 11

Z

zmq_sub() (in module meepo.sub.zmq), 13