[swindl]

Reference

crash-safe storage and fast search for Python web apps

version 3.2.1 · swindl.seecr.nl

seecr

Contents

Setting up a store 2
Models 3
Storing 3
Transactions 4
After a crash 4
Searching 5
OAI-PMH 6
    The provider 6
    The downloader 7
Reindexing 8
Checking: swindl-check 8
Forms: the WebAdapter 9
Testing 9
Troubleshooting 9

swindl (Seecr Web Index and Database Lib) is the storage side of an SWL application. It keeps objects in sjdbl's JSON files and makes them searchable with sil's index, as one `Store`. It also has the WebAdapter, which turns the text of an HTML form into the typed values of an object and back. This page lists what swindl provides. For a guided introduction, run and read the examples first; this page is for looking things up:

```
PYTHONPATH=. python examples/store.py        # storing, searching, revisions, archive
PYTHONPATH=. python examples/webadapter.py   # forms: converting, checking, errors
```

Setting up a store

```
from swindl.store.store import Store
from swindl.store.indexes import Commit

store = Store(data_dir, models=[Customer, Order])     # once, at startup
...
store.disconnect()                                     # at shutdown: closes the indexes
```

`data_dir` is a `pathlib.Path`. The store keeps the objects in `data_dir/store/<Model>/<id>.json` (sjdbl) and an index per model in `data_dir/index/<Model>` (sil). Both directories are made when they are missing.

`commit` says when changes become searchable:

`Commit()` (default)	within 5 seconds (`Commit(timeout=...)`); many puts share a commit. Needs a running event loop when the store is made (the timer is an asyncio task)
`Commit(immediate=True)`	at the end of every put, delete or transaction: searchable at once; for tests, scripts and examples
`Commit(on_search=True)`	just before a search, when something changed since the last commit: a save costs no commit, a search finds everything saved before it, and many saves without a search in between share one commit. For writes in bursts (harvesting, many updates at once). The timer still commits what no search asks for (`timeout=0` : nothing)
`Commit(timeout=0)`	only at `store.commit()` and `disconnect()` ; for bulk imports

A `get` always reads the files, so it sees a change at once either way; only queries wait for the commit. A delete is committed at once (at the end of its transaction), except with `on_search`, where the next search commits it.

Which to choose: `immediate=True` when a save is almost always followed by a search (a page that saves and reloads its list: a commit is paid either way, so pay it at once); `on_search=True` when saves come in bursts, or without searches after them; the default timer when a query may be a few seconds behind.

Models

A model is a subclass of `swindl.store.Base`: sjdbl's `Base` with `INDEX` and `ADAPTATIONS` (see Forms). It has its fields and, for searching, its index fields and their values:

```
from sil.indexfield import IndexField
from sjdbl.field import Field
from swindl.store import Base

class Book(Base):
    FIELDS = Base.FIELDS + [
        Field.str("title"),
        Field.str("genre"),
        Field.int("year"),
        Field.str("note"),                              # stored, not indexed
    ]
    INDEX = Base.INDEX + [
        IndexField.unqualified("all"),                  # for a query without field name
        IndexField.text("title"),
        IndexField.untokenized("genre", drilldown=True),
        IndexField.int("year", sortable=True),
    ]

    def index_values(self):                             # called at every put
        values = super().index_values()
        values.update(all=f"{self.title} {self.genre}", title=self.title,
                      genre=self.genre, year=self.year)
        return values
```

The field types (`Field.str`, `date`, `datetime`, `decimal`, `bool`, `enum`, `json`, `reference`, ...) are sjdbl's; see its README. When the class is defined, a field name used twice or one of `Base`'s attributes (`id`, `qid`, `update`, ...) raises a `TypeError`. Every index also has `__id__` and `__archived__` (the text `"True"` or `"False"`), added by swindl.

Index field	For
`IndexField.untokenized(name)`	exact values: codes, ids, statuses; filters
`IndexField.text(name)`	words: searched per word, lowercased
`IndexField.int(name)`	numbers
`IndexField.unqualified(name)`	the field a query without `field:` searches

Options: `sortable=True`, `drilldown=True` (counts per value), `stored=True`, `suggest=True`.

Storing

```
book = store.create(Book, dict(title="Dune", year=1965))   # makes and puts
book = Book.create(title="Dune"); store.put(book)          # the same in two steps

book = store.get("Book:<id>")                  # None when not there or archived
book.update(year=1966)                         # True when something changed
store.put(book)                                # files and index
store.archive(book)                            # hidden from get and list
store.delete(book)                             # gone, from files and index
```

`create(cls, values=None)`	a new object from a dict, put; `cls` may be the name
`put(*objs)`	stores the objects and indexes them
`get(qid_or_qrid, include_archived=False, assert_type=None)`	the object, or `None`
`archive(obj)`	marks it archived and puts it
`delete(obj)`	removes it from the files and the index; `obj` may also be a reference to one (a `Proxy`, as references and collections give; nothing is loaded) or its qid

`enumerator(...)`	sjdbl's `Enumerator`, for sequential numbers
`transaction()`	a block whose puts and deletes are stored together, or not at all; see Transactions
`commit()`	commits the indexes now (only needed with `Commit(timeout=0)`)
`disconnect()`	commits and closes the indexes

qid and qrid. `obj.qid` is `"Book:<id>"`; `obj.qrid` adds the revision, `"Book:<id>:<rev>"`. The revision changes at every put. `get(qrid)` raises `HasChangedException` (from `swindl.store.store`) when the object was changed since: put the qrid in a form, and you know whether someone else saved in between.

`assert_type=Book` raises `TypeError` when the qid is of another model; use it for qids that come from a request.

A put whose indexed values did not change (only `note`, say) writes the file but leaves the index alone, and does not wait for a commit.

Transactions

Every put and delete is a transaction of its own. A `transaction()` block makes one of several:

```
with store.transaction():
    order.update(state="paid")
    store.put(order, invoice)
    store.delete(basket)
# stored, all of it; and indexed
```

- At the end of the block everything is stored together, or nothing is: an exception in the block discards it (files and index unchanged, the objects get their old revisions back), and a crash halfway is finished or undone when the store is opened again (see After a crash).
- Within the block, `get` sees its puts and deletes (in this thread; other threads see them when the block is done). `list`, `query`, `find_one` and `find_by_field` don't: they search the index, which follows at the end.
- The store's lock is held for the whole block: changes from other threads wait. Keep blocks short in a web application.
- Blocks can be nested; the outermost one commits.
- It is also the fast way to store many objects: one sync of the files and one commit of the index for the whole block, instead of one per put.

```
with store.transaction():
    for order in customer.orders:      # e.g. after a rename: their index changes
        store.put(store.get(order.qid))
```

To act on changes once they are done (stored and indexed), register a listener; it gets the changes of every transaction (qid to the object, or `None` for a delete):

```
store.on_transaction(lambda changes: print(sorted(changes)))
```

It runs in the thread of the transaction, with the store's lock held: keep it short (wake up something). An exception in it is logged; the transaction stays done.

After a crash

The files are the truth and the index is derived from them, so swindl makes sure of two things when the process (or the machine) dies:

1. **The files have a transaction completely or not at all.** sjdbl writes the new versions to temporary files first, and with more than one file it commits by writing `store/commit.log`. A commit that was interrupted is finished when the store is opened; temporary files of a transaction that never committed are removed.

2. **The index catches up.** Before the files are committed, the qids of the transaction go to `index/pending.log`; after the index has committed them they are removed. When the store is opened, the objects still in it are indexed again from the files (a qid without a file is removed from the index). With the default `Commit()` this also covers the changes of the last seconds before a crash that were not committed yet.

Both are logged as a warning and need nothing from the application. This is not a reindex: only what the crash left unfinished is done. `swindl-check` shows beforehand what opening will do. See `doc/guarantees.md` for what is promised, and what is not.

Searching

```
from swindl.store.store import QueryParams

result = store.query(QueryParams(
    "Book",
    query="*",                                  # everything; or e.g. "title:dune"
    filters={"genre": "fantasy"},               # exact terms, all must match
    sort=[{"field": "year", "reverse": True}],  # or "year", or a list of those
    page=0, size=20,
    drilldown=[{"name": "genre", "count": 10}],
))
result.objs                 # the objects of this page
result.total                # all hits
result.drilldown            # {"genre": [("fantasy", 2)]}
```

The query uses sil's syntax (tantivy): words, `field:word`, `"a phrase"`, `AND` / `OR`, `prefix*`. Words without a field search the `unqualified` index field. A query that is not valid syntax (a stray quote from a search box) is searched for literally.

Filters are exact terms on index fields; a list as value means one term per value, all of them must match (so for a field with one value, a list finds nothing). On a `text` field a term is one lowercased word (`title="dune"`). `Query` from `swindl.store.indexes` is the query with its terms, when you build it yourself:

```
q = Query("lord")
q.add_term("genre", "fantasy")
store.query(QueryParams("Book", query=q))
```

Sorting only on `sortable=True` fields; equal values (and no sort with `"*"`) keep the order in which the objects were indexed. With a query and no sort, the best matches come first.

The result (`QueryResult`):

`objs`	the objects of this page
`total`	the number of hits (in the index; a hit whose file is gone is left out of `objs` but still counted)
`start`, `stop`	the numbers of the first and last hit of this page, from 1
`pages.first`, `.prev`, `.next`, `.last`	page numbers, `None` where there is none
`is_last_page()`	
`params`	the `QueryParams`; change `page` and query again for the next page
`drilldown`	per drilldown field a list of `(value, count)`, most first

Shortcuts:

`list(cls, include_archived=False, **filters)`	a generator over all pages
`find_one(cls, **filters)`	the first hit or `None`
`find_by_field(cls, field, value)`	the objects with this term (one page)

`query`, `find_one` and `find_by_field` do **not** leave out archived objects; `list` does. Add `filters={"__archived__": "False"}` to a query that should not show them.

OAI-PMH

`swindl.oai` keeps the OAI-PMH administration of a repository in the store, as meresco-oai's OaiJazz did: per identifier the metadata prefixes and sets it is in, those it is deleted in, and a stamp (microseconds, unique, increasing) for every change. Add `swindl.oai.MODELS` to the store's models; one store can hold several repositories.

```
from swindl.oai import MODELS, Oai

store = Store(path, MODELS + [Article])
oai = Oai(store, "full", always_delete_in_prefixes=["didl"])

with store.transaction():              # with the application's own data, or neither
    store.put(article)
    oai.add(article.identifier, prefixes=["didl", "oai_dc"], sets=["genre:poetry"])
```

`oai.add(identifier, prefixes, sets=())`	in these prefixes and sets (besides those it was in), no longer deleted in them; a set `a:b` is in `a` too
`oai.delete(identifier, prefixes=(), sets=())`	deleted in all its prefixes and sets, and in `always_delete_in_prefixes` and those given (also when it was not in them)
`oai.delete_in_prefixes(identifier, prefixes)`	deleted in these prefixes only
`oai.delete_in_sets(identifier, sets)`	deleted in these sets only; not standard OAI-PMH, needs `Oai(..., delete_in_sets=True)`
`oai.update_format(prefix, schema, namespace)`, `oai.update_set(spec, name)`	the schema and namespace of a prefix, the name of a set
`oai.get(identifier)`	its `OaiRecord` (`prefixes`, `deleted_prefixes`, `sets`, `deleted_sets`, `tombstone`, `stamp`, `datestamp`), or `None`
`oai.select(prefix, set=None, continue_after=0, from_=None, until=None, size=200)`	`(records, more)`: the records of a list, in the order of their stamps
`oai.formats`, `oai.sets`, `oai.earliest_stamp()`	every prefix and set used, the first stamp

A record deleted in all its prefixes, or in all its sets, is deleted (`tombstone`), as in OaiJazz; so a record deleted with its sets and added again without them stays deleted.

The provider

`OaiProvider` answers OAI-PMH 2.0 requests for an `Oai`, as meresco-oai did: the same XML, error codes and messages, and resumption tokens (`m<prefix>|c<stamp>|f<from>|u<until>|s<set>|p`). It does not depend on a web framework; the application's route passes the arguments:

```
from swindl.oai.provider import CLIENT_HEADER, OaiProvider

provider = OaiProvider(
    oai,
    metadata=lambda record, prefix: stored_xml(record.identifier, prefix),
    name="Example", admin_email="beheer@example.org",
    repository_identifier="example.org", identifier_prefix="oai:example.org:",
    batch_size=200, supports_x_wait=True,
)

async def oai_endpoint(request):
    response = await provider.handle(
        list(request.query_params.multi_items()),
        base_url=str(request.url.replace(query="")),
        client=request.headers.get(CLIENT_HEADER),
    )
    return Response(response.body, status_code=response.status,
                    media_type=response.content_type)
```

`metadata(record, prefix)`	the metadata of a record in prefix: an lxml element or XML text (`None`: empty)
`about(record, prefix)`	XML text for `<about>` (provenance); `""` gives an empty `<about>` (as meresco), `None` none
`identifier_prefix`	put before the identifiers shown, taken off those asked for
`repository_identifier`, `descriptions`, `watermark`	Identify's oai-identifier description, more descriptions, a comment after `<OAI-PMH>`
`batch_size`	records per part (default 200)
`delete_in_sets`	mark the sets a record is deleted in with `status="deleted"` in its header (meresco's `OaiPmh(deleteInSets=...)`); default: when the `Oai` deletes in sets
`supports_x_wait`, `max_waiting`	x-wait, below (default off; at most 100 waiting)

`provider.respond(arguments, base_url)` answers without waiting (for tests, or without x-wait).

x-wait. A `ListRecords` or `ListIdentifiers` with `x-wait=True` that finds nothing waits until a change in its prefix (and set) is stored, then answers; with x-wait the last part of a list has a resumption token too, so a harvester goes on exactly where it was. One request per client waits (the `X-Meresco-Oai-Client-Identifier` header): a new request of the same client ends the waiting one with `204 No Content`, and above `max_waiting` a random one is ended so. Requests wake up after the transaction of the change, when it is searchable: the store needs `Commit(immediate=True)` or `Commit(on_search=True)` (else `ValueError`).

The downloader

`OaiDownloader` harvests another OAI-PMH repository into the application's store, as meresco's OaiDownloadProcessor did: part after part with `ListRecords`. Each part goes to `process(records)` in a store transaction, together with the position after it; so a part and its position are stored together, or neither, and after an error or a crash the same part comes again (`process` must not mind getting a record twice).

```
from swindl.oai import DOWNLOADER_MODELS, OaiDownloader

store = Store(path, DOWNLOADER_MODELS + [RichRecord])

def process(records):                # in a transaction of store
    for record in records:
        rich = RichRecord.for_identifier(store, record.identifier)
        if record.deleted:
            store.delete(rich)
        else:
            rich.update(data=convert(record.metadata))
            store.put(rich)

downloader = OaiDownloader(store, "conversion", "http://conversion:8000/oai",
                           metadata_prefix="document", process=process, x_wait=True)
downloader.start()                   # an asyncio task (in the app's lifespan)
await downloader.stop()
```

A `Record` has `identifier`, `datestamp`, `deleted`, `sets`, `metadata` (the element in `<metadata>`), `about` (the elements in the `<about>`s) and `element` (the `<record>`).

`x_wait=True`	the repository holds the request until there is something new (swindl's or meresco's provider with x-wait): changes come in at once. Sends a client identifier of its own (kept in the state)
without `x_wait`	a finished list is asked for again after `interval` seconds (default 60), from the `responseDate` of the request that began it

`set`, `from_`, `headers`	the set to harvest, the `from` of the first list, more HTTP headers
`retry_delay`	seconds to wait after an error (default 30); the error is in `downloader.state.error`
`downloader.state`	`OaiDownloadState`: `resumption_token`, `from_`, `records` (count), `last_request`, `error`
`downloader.reset()`	harvest everything again (an explicit step)
`fetch(url, headers)`	the HTTP GET, `(status, body)`; default urllib in a thread. For tests, an async function (e.g. calling an `OaiProvider` directly)
`parser`	the lxml `XMLParser` for the responses, e.g. `XMLParser(huge_tree=True)` for very large records

Reindexing

`store.reindex(models)` empties the indexes and indexes every object of the given models again from the files. It is needed after a change to a model's `INDEX` or `index_values`, and only then. swindl never does it by itself, and application code doesn't either: the application offers it as an explicit step, e.g. a file `<data>/actions/reindex` that the server checks at startup, and the CHANGELOG of the release says when to do it. When a model's index fields changed, sil starts that index empty and logs that it needs a reindex.

Checking: swindl-check

```
swindl-check <data dir> <module>:<list of models>     # or <module>: all models in it
swindl-check /data/myapp myapp.models:ALL_MODELS
```

Reports, one per line, whether the index matches the files, and changes nothing (it does not even open a `Store`, so it can run next to the application):

- leftovers of a crash: `store/commit.log`, `.tmp` files, objects in `index/pending.log` (opening the store finishes them);
- objects indexed without a file, files that are not indexed, objects indexed twice;
- index values that differ from the file (for `untokenized` and `int` index fields);
- files that are not valid JSON, references to objects that don't exist.

Exit code 0 and `ok` when all is well, 1 otherwise. In code: `swindl.check.check(data_dir, models)` gives the list.

Forms: the WebAdapter

`swindl.webadapter` converts and checks what comes in from a web form, and gives the values for the inputs of a form. The full description is in `doc/webadapter.md`; this is the summary.

```
from swindl.webadapter import Adapt, FormErrors, at_least, from_form, max_length, to_form

class Person(Base):
    FIELDS = [Field.str("name"), Field.date("dob"), Field.int("age"),
              Field.bool("active", default=False)]
    ADAPTATIONS = [                              # a field without an Adapt is text
        Adapt.str("name", checks=[max_length(100)]),
        Adapt.date("dob"),
        Adapt.int("age", checks=[at_least(0)]),
        Adapt.bool("active"),
    ]

fields = ["name", "dob", "active"]
try:
    person.update(**from_form(Person, form, fields, obj=person))
except FormErrors as e:
    values = to_form(person, fields, form=form)  # what was typed; e.errors per field
```

The rules, per field: not in `fields`, never touched; not sent, left alone (a checkbox: `False`); sent, stripped, and empty becomes `None` (text: `""`); then converted (`invalid_int`, `invalid_decimal`, `invalid_date`) and checked.

`from_form(model, form, fields, obj=None, adaptations=None)`	the values, or `FormErrors` with all errors
`to_form(obj, fields, form=None, adaptations=None)`	the values for the inputs (`None` is `""`); with `form=` what was typed
`Adapt.str`, `.int`, `.decimal(decimals=2)`, `.date`, `.bool`	with `checks=[...]`, and `from_web` / `to_web` to replace the conversion
`Adapt.custom(name, from_web(obj, form), to_web(obj, value))`	anything else; `obj` is `None` when creating
`at_least(n)`, `at_most(n)`, `max_length(n)`, `one_of(values)`	checks: `too_small`, `too_large`, `too_long`, `not_allowed`
`FieldError(code, **params)`	a wrong value; raise it in a check of your own
`FormErrors.errors`	field name → `FieldError` (`.code`, `.params`); the application translates

`adaptations` adds or overrides `Adapt`s for one form, e.g. `one_of` with a list from the store. swindl's `Base` checks at class definition that every `Adapt` names a field.

Testing

Use `Commit(immediate=True)` so queries see a put at once, and pytest's `tmp_path`:

```
def test_find(tmp_path):
    store = Store(tmp_path, [Book], commit=Commit(immediate=True))
    store.create(Book, dict(title="Dune"))
    assert store.find_one(Book, title="dune").title == "Dune"
```

The WebAdapter needs no store: a model with `ADAPTATIONS` and a dict as form will do (`swindl/webadapter_test.py`).

Troubleshooting

A query doesn't find what was just put	commit: within 5 seconds, or `Commit(immediate=True)`; within a `transaction()`, at its end

`RuntimeError: no running event loop` when making the store	the default `Commit()` starts a timer: make the store in the application's startup, or use `Commit(immediate=True)` / `Commit(timeout=0)` in scripts
A warning about `commit.log` or `pending.log` at startup	the process died during a transaction; it was finished, nothing to do
Archived objects in a query	`query` doesn't leave them out; filter on `__archived__: "False"`
A filter on a text field finds nothing	it is one lowercased word: `title="dune"`, not `"Dune"`
`RuntimeError: Unsortable field`	add `sortable=True` to the index field; then reindex
A changed `INDEX` gives old results	reindex: it is not done by itself
A field is not searchable	it needs an `INDEX` field and a value from `index_values`; then reindex
A checkbox is saved as off by a form that doesn't show it	it is in `fields`; list only what the form has
`TypeError` at import: Adapt for ..., which is not a field	a typo in `ADAPTATIONS`, or the field is missing from `FIELDS`
A model gets no index	it is built on sjdbl's `Base`; use `swindl.store.Base`
`HasChangedException`	someone saved the object since the qrid was taken; get it again