[sjdbl]

Reference

Python objects as plain JSON files

version 0.6.0 · swindl.seecr.nl

seecr

Contents

Requirements 2
Defining models 2
Storing and loading 3
    Transactions 3
    Crash safety 3
Identity: id, qid and qrid 4
Revisions and conflicts 4
Archiving 4
References and lazy loading 5
Enumerators 5
Examples 5
Limits 5

SJDBL stores Python objects as plain JSON files: one file per object, one directory per class. It is a storage layer only. There is no query language; searching belongs in a separate index, which can be rebuilt from the store at any time.

Design choices:

- **Plain text.** Every record is a readable, indented JSON file. You can inspect it with `cat` / `jq`, `grep` it, diff it and fix it by hand.
- **No server.** A store is a directory. A backup is a copy of that directory.
- **Single process.** One `Store` per directory, used from one process. Threads within that process are safe.

Requirements

Python 3.11 or newer, lxml and pytest (the tests, `*_test.py`, are part of the package). Install with `pip install sjdbl`.

Defining models

A model is a subclass of `Base` with a list of `FIELDS`:

```
from sjdbl import Store
from sjdbl.base import Base
from sjdbl.field import Field


class Person(Base):
    FIELDS = [
        Field.str("name"),
        Field.date("dob"),
        Field.reference("parent"),
    ]
```

Available field types:

Field	Python value	Stored as
`Field.str(name, default="")`	`str`	string
`Field.text(name, default="")`	`str`	string
`Field.int(name, default=0)`	`int`	number
`Field.bool(name, default)`	`bool`	boolean
`Field.decimal(name, default=None)`	`Decimal`	string, e.g. `"9.95"`
`Field.date(name, default=None)`	`date`	ISO string
`Field.datetime(name, default=None)`	`datetime`	ISO string
`Field.enum(name, enum, default)`	`Enum` member	the member's value
`Field.json(name, default=list)`	any JSON-able structure	JSON encoded as a string
`Field.set(name)`	`set` of str or int	JSON list as a string
`Field.xml(name, default=None)`	lxml tree	XML string
`Field.uuid(name, default=new_uuid)`	`str` (read-only)	string
`Field.id(name, default=None)`	any	as is
`Field.reference(name)`	another object	its qid
`Field.collection(name)`	list of objects	list of qids

A callable default (such as `list`) is called for every new object.

The fields are checked when the class is defined: a `TypeError` for a name that is used twice, or that is an attribute of `Base` (`id`, `qid`, `update`, `create`, `__id__`, ...). Such a field would silently be hidden behind that attribute.

A subclass lists all its fields, including the inherited ones; they are not merged:

```
class Employee(Person):
    FIELDS = Person.FIELDS + [Field.str("department")]
```

Storing and loading

```
from pathlib import Path

store = Store(path=Path("data"), models=[Person])

john = Person.create(name="John Smith")
store.put(john)

again = store.get(john.qid)       # None if it does not exist
again.update(name="John Doe")     # returns True if something changed
store.put(again)

for person in store.list(Person):
    print(person.name)

store.delete(again)               # or store.delete(again.qid)
```

Transactions

Puts and deletes in a `transaction()` block are written together at its end, or not at all:

```
with store.transaction():
    order.update(state="paid")
    store.put(order, invoice)
    store.delete(basket)
```

- An exception in the block discards it: nothing is written, and the objects get their old revisions back.
- Inside the block `get()` sees its puts and deletes; `list()` does not, and neither does `get()` in other threads until the block is done.
- The store's lock is held for the whole block: other threads' puts and deletes wait. Blocks can be nested; the outermost one commits.
- It is also the fast way to load many objects: the files are synced together at the end, instead of each put waiting for its own sync.

Crash safety

Every change is a transaction (a single `put` or `delete` is one of its own), and a transaction is in the files completely or not at all, also when the process or the machine dies halfway:

1. The new versions are written to temporary files (`<id>.tmp`) and synced to disk.
2. When more than one file changes, `commit.log` (in the store's directory) lists them and is synced. From this moment the transaction is committed.
3. The temporary files are renamed into place and deleted files removed; the directories are synced. A single file needs no log: its rename is atomic.
4. `commit.log` is removed.

When a store is opened, a `commit.log` that is still there is finished (step 3), and temporary files of a transaction that never committed are removed; both are logged as a warning. This happens once per process for a directory: a second `Store` on the same directory does not touch a commit that is still going on.

`john` is stored in `data/Person/<id>.json`:

```
{
  "data": {
    "name": "John Doe",
    "dob": null,
    "parent": null
  },
  "meta": {
    "__id__": "335b6043-b6aa-4011-98aa-893271a23d97",
    "__rev__": "01064895-d3f3-41d1-825a-1f944ab3380d",
    "__archived__": false
  }
}
```

`list()` reads every file of a class. It is meant for full passes such as rebuilding an index, not for searching.

Identity: id, qid and qrid

Every object has three meta fields: `__id__`, `__rev__` and `__archived__`.

- `obj.id` – the uuid of the object, a random uuid4. It carries no meaning; store things like a creation time in an explicit field.
- `obj.qid` – qualified id, `"Person:<id>"`. Use it in URLs, forms and indexes.
- `obj.qrid` – qualified revision id, `"Person:<id>:<rev>"`. Identifies one specific version of the object.

Objects compare equal and hash by `qid`, so they can be used in sets and as dict keys.

Revisions and conflicts

`__rev__` changes every time an object is written by `put()`. `update()` only changes the data in memory.

`put()` checks that the version on disk is still the version the object was loaded from. If someone else saved the object in the meantime, it raises `HasChangedException` and writes nothing:

```
from sjdbl.base import HasChangedException

first = store.get(john.qid)
second = store.get(john.qid)

first.update(name="Jane Smith")
store.put(first)

second.update(name="Joe Smith")
try:
    store.put(second)
except HasChangedException:
    ...  # reload, merge or tell the user
```

`put(a, b, ...)` checks and serializes all objects before writing any of them, so a conflict or an unserializable value leaves every file untouched. It is one transaction (see Transactions): after a crash either all of them have the new version or none. Each file is replaced by a rename, so a reader always sees either the old or the new version of a file.

Archiving

```
john.archive(store)          # or store.archive(john)
john.is_archived             # True
john.mark_archived(store, archived=False)
```

Archiving is a normal `put()` with `__archived__` set, so it also gets a new revision. The store itself does not hide archived objects; that is up to the caller or the index.

References and lazy loading

A `Field.reference` or `Field.collection` is loaded lazily. After `get()`, the referenced object is a `Proxy` that only holds the qid. On first attribute access it loads the object and becomes it:

```
child = Person.create(name="Joe Smith", parent=john)
store.put(child)

child = store.get(child.qid)
child.parent                # Proxy, nothing loaded yet
child.parent.qid            # still not loaded
child.parent == john        # True, compared by qid, still not loaded
child.parent.name           # loads john from disk
```

Each proxy loads its own copy: two children with the same parent give two separate parent objects.

If the referenced object no longer exists, attribute access raises `DanglingReferenceError` (a `ValueError`). `hasattr()` checks for dunder names such as `__html__` do not load the proxy and simply return `False`.

Enumerators

For sequential numbers (invoice numbers, case numbers), declare an enumerator on the model:

```
from sjdbl.enumerate import EnumerateValue


class Invoice(Base):
    ENUMERATORS = [EnumerateValue("number", start=1000)]
    FIELDS = [Field.int("number")]


store = Store(path=Path("data"), models=[Invoice])
number = store.enumerator(Invoice, "number").increment()   # 1000, 1001, ...
store.put(Invoice.create(number=number))
```

The counter is stored as plain text in `data/Invoice/number.enumerator`. `increment()` is thread-safe and durable: a number is never handed out twice, also not after a crash.

Examples

The `examples/` directory contains runnable scripts:

- `basic.py` – models, put/get/update/list/delete and the file on disk
- `references.py` – references, collections and lazy loading
- `conflicts.py` – revisions, `HasChangedException` and archiving
- `enumerators.py` – sequential numbers

Run them from the project root against this checkout, e.g. `PYTHONPATH=. python examples/basic.py`.

Limits

- **One process per directory.** Threads are fine (one lock per `Store`); several processes on the same directory are not: the revision check and recovery assume one process.
- **`list()` reads every file** of a class: for rebuilding an index, not for searching. Searching is an index's job.
- **One file per object, one directory per class.** Fine for hundreds of thousands of objects per class on ext4, XFS or APFS.
- **Durability:** `fsync` of files and directories; on macOS `fsync` does not flush the drive's cache.