# A Zotero Write-and-Mirror Pipeline, and a Lossless Database Recovery



INDEPENDENT SOFTWARE ENGINEER · 2026

## Context

A research project needed its sources mirrored into a reference manager as a single, clean collection, with each item carrying its own document, and it needed a local catalogue resilient enough to survive loss of the original files. The two together are the difference between a bibliography that is a list of titles and one that is a working archive a reviewer could actually open. The work was engineering against a real external API and a real local store, so it is described here in full, without the publication sensitivity that constrains the research case studies.

## What I built

I extended a Python tool to write to the Zotero Web API. It creates a project collection, creates bibliographic items idempotently, and attaches the source document to each item.

Idempotency was the property I designed around first, because a mirror is something you run more than once. On each run the tool reads what the collection already contains and skips any item already present by digital object identifier, then by URL, then by title, so a re-run reconciles rather than duplicates. That check is what lets the pipeline be run repeatedly and safely as the source base grows, which is the normal case, not the exception.

quest upload authorisation, upload the bytes to storage, then register the upload as complete. I implemented the whole sequence, plus a check for whether an item already carries an attachment so that re-runs do not re-upload.

I generated the item payload by joining two sources of truth rather than one: the authoritative, hand-checked bibliography supplied titles, authors, dates, and identifiers, and the local catalogue supplied item types. Keeping the human-verified metadata as the source for citations, and the machine catalogue only for structure, is the same separation that protects the reference list from unverified data elsewhere in the project. I then mirrored the full set of sources into one collection and attached every document, and confirmed the result by reading the collection back and checking that each item carried an attachment, rather than trusting that the writes had succeeded.

## Bugs found and fixed

Two real defects surfaced only under real load, which is where this kind of integration is actually tested.

Item creation posted every record in a single request and hit the API's fifty-item cap, which returned an error rather than a partial success. I changed it to send batches of fifty, and gave each batch its own idempotency token so that a later batch can never be mistaken by the server for a retry of an earlier one. Getting the token scoping right mattered as much as the batching: a single shared token would have reintroduced exactly the duplication the batching was meant to avoid.

Separately, the routine that reads a collection to avoid duplicates fetched only the first page of results. That was invisible while the collection was small, but once it grew past a hundred items, counting the items and their attachments together, a re-run would have started creating duplicates of everything on the later pages, because the deduplication check could not see them. I changed it to page through the whole collection. Both fixes are covered by the test suite, which passes.

## A diagnosis worth keeping

At one point the local catalogue database appeared corrupt: every read returned zero bytes, and both the tooling and a direct connection reported that the file was not a database. The obvious move, rebuilding it from scratch, would have re-numbered every record and broken the link between records and their stored copies, so I refused to take it until I understood the cause.

reporting an empty file. Reading each file through triggered its download and materialised it, and the database came back whole, which I confirmed with an integrity check rather than assuming. I recovered it without loss, refreshed the derived exports, and flagged the offloading as a standing risk to the project's local-copy strategy, with the specific setting to change so that it does not recur. The lesson worth keeping is that the destructive fix and the correct fix looked identical at the symptom level, and only understanding the cause told them apart.

## One more environment quirk

The pipeline runs from a Python virtual environment, and its first attempts to reach the Zotero endpoints failed on certificate verification, because the interpreter was not resolving a trusted certificate-authority bundle. Rather than disable verification, which would have been the quick and wrong fix, I pointed the process at a real bundle so the connections verify properly. Small as it is, it is the kind of environment detail that separates a design that works on one machine from one that works when deployed, and the temptation to silence the error instead of fixing it is exactly the temptation to resist.

## What it demonstrates

Integration against a real external service, its documented protocol, and its quirks; idempotent design so an operation can be repeated safely; methodical debugging when the symptoms are ambiguous and the obvious diagnosis is the wrong one; a preference for lossless recovery over destructive rebuilds; security-conscious defaults under pressure; and fixes backed by tests rather than by hope.

## Reference


- Zotero Web API, version 3. Official documentation, including the write API and the multi-step file-upload authorisation flow. [zotero.org/support/dev/web_api/v3/start](zotero.org/support/dev/web_api/v3/start).


Download PDF ↓