— CHAPTER 05 · BUNDLES & TRANSACTIONS · BEAT 1 — ONE CALL, MANY RESOURCES

So far, one resource at a time. A clinical event arrives all at once.

A patient, the encounter, the readings — they all reference each other, and they need to land together. This chapter is how you post the whole thing in one call. We read the Bundle types, compose a transaction that creates several linked resources atomically, and use the conditional operations that keep your writes idempotent.

What you will be able to do

- Bundle types — transaction · batch · document · message · searchset
- Transaction atomicity — all-or-nothing vs best-effort
- Conditional ops — create · update · delete

— CONCEPT · BUNDLE.TYPE — A CLOSED CODE SET

A Bundle is a container. Its type tells the server what to do with it.

Type is not cosmetic — it is a closed code set, and it changes everything the server does with the Bundle. Today we live in two of them and read a third.

MULTI-ACTION

transaction

Several actions, atomic. All commit or all roll back.

Taught deeply

MULTI-ACTION

batch

Several actions, independent. Best-effort, no shared rollback.

Taught deeply

SIGNED PAYLOAD

document

A frozen, signed package. First entry is a Composition.

Composition first

EVENT PAYLOAD

message

Drives an event. First entry is a MessageHeader.

MessageHeader first

QUERY RESULT

searchset

What a query returns. We read one later this chapter.

Read-only

LOOSE GROUPING

collection

A grouping with no processing promise at all.

AUDIT TRAIL

history

A resource's version history, oldest to newest.

THE POINT

Type drives processing

Pick the type and you pick the semantics. We focus on three.

Closed code set

— COMPARISON · ATOMICITY — THE REAL DIFFERENCE

Both send many actions. Only one rolls them all back.

TRANSACTION

All-or-nothing

Server accepts every action and returns success, or rejects every action and rolls the whole thing back. Nothing half-applied.

Returns transaction-response

BATCH

Best-effort

Each entry processed independently. One entry can fail four-twenty-two and its siblings still commit.

Returns batch-response

Ava

Atomicity. In a transaction, one entry failing fails everything.

Marcus · the rule of thumb

One entry's failure does not touch its siblings. Pick transaction when the resources only make sense together.

— CONCEPT · INSIDE ONE BUNDLE.ENTRY

The request says what to do. The response says what happened.

Every entry has two halves. On the way in, a request — a method and a url. On the way back, the response Bundle mirrors each entry with a result.

— ENTRY.REQUEST — WHAT TO DO

{ "method": "POST", "url": "Patient" }

The method is the familiar verb — POST to create, PUT to update, DELETE, PATCH, even GET. The url is the target — Patient to create one, Observation slash one-two-three to address an existing one.

— ENTRY.RESPONSE — WHAT HAPPENED

{ "status": "201 Created", "location": "Patient/abc", "etag": "W/\"1\"" }

A status like two-oh-one Created, a location header pointing at the new resource, and an ETag carrying its version. One response object per entry, in order.

- **method + url — the action, per entry**
- **status + location + etag — the result, per entry**

— FRAMEWORK ANCHOR · ONE TRANSACTION, THREE LINKED RESOURCES

None of them have ids yet. urn:uuid is the placeholder that stitches them.

— TRANSACTION BUNDLE — PATIENT + ENCOUNTER + OBSERVATION

```
{ "resourceType": "Bundle", "type":
"transaction",
  "entry": [
    { "fullUrl": "urn:uuid:p1",  // (1)
      "resource": { "resourceType":
"Patient" },
      "request": { "method": "POST", "url":
"Patient" } },
    { "fullUrl": "urn:uuid:e1",
      "resource": { "resourceType":
"Encounter",
        "subject": { "reference":
"urn:uuid:p1" } },  // (2)
      "request": { "method": "POST", "url":
"Encounter" } },
    { "fullUrl": "urn:uuid:o1",
      "resource": { "resourceType":
"Observation",
```

The four moves

- (1) fullUrl carries the urn:uuid placeholder for an unborn id
- (2) another entry references that exact urn:uuid
- (3) on POST the server assigns real ids and rewrites every matching link, Bundle-wide
- (4) circular references are allowed — all ids resolve together

These are the same relative references from chapter three — they just start life as a urn:uuid until the server hands out the real id.

— CONCEPT · CONDITIONAL CREATE — IF-NONE-EXIST

Run that transaction twice. Get one patient, not two.

Conditional create checks first. You attach a search; the server creates only if nothing already matches. It is create-if-not-there.

— STANDALONE HEADER

If-None-Exist: identifier=http://hospital.org/mrn|12345

Sent alongside a POST. The server runs the search before it writes.

— INSIDE A BUNDLE ENTRY

```
"request": { "method": "POST", "url": "Patient",
  "ifNoneExist": "identifier=http://hospital.org/mrn|12345"
}
```

The same condition, carried on entry.request.ifNoneExist.

- 0 matches — create the resource
- 1 match — 200, no-op, return the existing one
- >1 match — 412, the condition was ambiguous
- This is how a seeder stays idempotent — re-run it, never get duplicates (Ch 16)

— COMPARISON · CONDITIONAL UPDATE & DELETE · IF-MATCH

Four-twelve is the recurring "your condition did not hold".

CONDITIONAL UPDATE

PUT [type]?search

1 match updates it. 0 matches creates it — an upsert. More than one is 412.

Upsert on a search

VERSION-AWARE UPDATE

If-Match: W/"vid"

Send the version you read. If someone changed it since, the server refuses — optimistic locking.

412 on version mismatch

CONDITIONAL DELETE

DELETE [type]?search

The mirror image — delete the resource matching a search query instead of an id.

By search, not by id

THE RECURRING CODE

412 Precondition Failed

Ambiguous match, or a stale version. Same status, different cause.

Your condition did not hold

— THE 412 GUARD

If-Match guards against a stale write — the four-twelve means someone changed the resource since you read it. Re-read, re-apply, retry.

— WALKTHROUGH · READING A SEARCHSET BUNDLE

A query returns a searchset. Three things to read.

— BUNDLE.TYPE — SEARCHSET


```
{ "resourceType": "Bundle", "type": "searchset", "total": 42,
  "link": [ { "relation": "self", "url": "..." },
            { "relation": "next", "url": "...&_offset=20" } ],
  "entry": [
    { "search": { "mode": "match" },   "resource": { "resourceType": "Observation" } },
    { "search": { "mode": "include" }, "resource": { "resourceType": "Patient" } } ] }
```


- total — the match count, can exceed the page in front of you
- link.next — walk through a paged result set
- search.mode — match (primary hit) vs include (pulled in alongside)
- Those include entries come from _include — full search is its own chapter (Ch 09)

— DIAGRAM · THE MANDATORY PROCESSING ORDER

This order is why slide 5's forward references resolve.

How HAPI processes a transaction

Always in this order, every time.

4 **GET / HEAD**
Reads, after the writes have landed.

5 **Resolve conditional & urn:uuid references**
Links rewrite to the real ids assigned in step 2.

Why the order is the integrity

Ids exist before anything tries to point at them — that ordering is what gives you referential integrity. Remember the scar from chapter three: a single POST lets you point at a patient who does not exist. Inside a transaction, the order is what protects you.

POST before PUT before reference resolution

— THE ONE CATCH

A conditional reference like Patient?identifier=12345 must match exactly one resource. Zero or more than one, and the whole transaction fails.

— TRY IT YOURSELF · 60 SECONDS

POST the slide-5 Bundle to the public HAPI sandbox.

— RUN THIS IN YOUR TERMINAL

curl -X POST https://hapi.fhir.org/baseR4 -H 'Content-Type: application/fhir+json' -d @transaction-bundle.json | jq '.entry[].response.status'

Three resources, one atomic call, references resolved. You should see a row of two-oh-one Created.

Ava · recap

And we have been hitting a public HAPI server the whole time.

Marcus · bridge

Next chapter we open the hood and see how HAPI is actually built.