— ACT III · LIVE UPDATES · CHAPTER 15

The BFF served it. Next.js rendered it. Now keep it live.

Chapter thirteen built the BFF. Chapter fourteen rendered a Patient, an Encounter, an Observation — each a snapshot, rendered once. The moment another clinician edits that patient on the server, the open view goes silently stale. How does the screen learn the resource just changed — without hammering the server with polling?

What this chapter answers

- The live-update problem — and why the obvious answers fall short
- The 5-layer loop — datastore to DOM
- A typed event contract — one schema, both ends

— COMPARISON · PICK THE TRANSPORT — A FIT DECISION

Three ways to learn about a change. Only one fits a one-way push.

The browser needs to know when a resource changed. Why not just ask again every few seconds? Because this is one-directional — the server pushes, the client only renders. That shape picks the transport.

— POLLING · THE BLUNT ONE

Re-request on a timer

You waste requests when nothing changed, and you are always stale by up to one interval. No push — just repeated questions.

— WEBSOCKET · THE HEAVY ONE

Full-duplex protocol upgrade

Real two-way channel, but you own the reconnect logic, the heartbeats, and there is no built-in replay if the connection drops.

— SSE · THE LEAN FIT

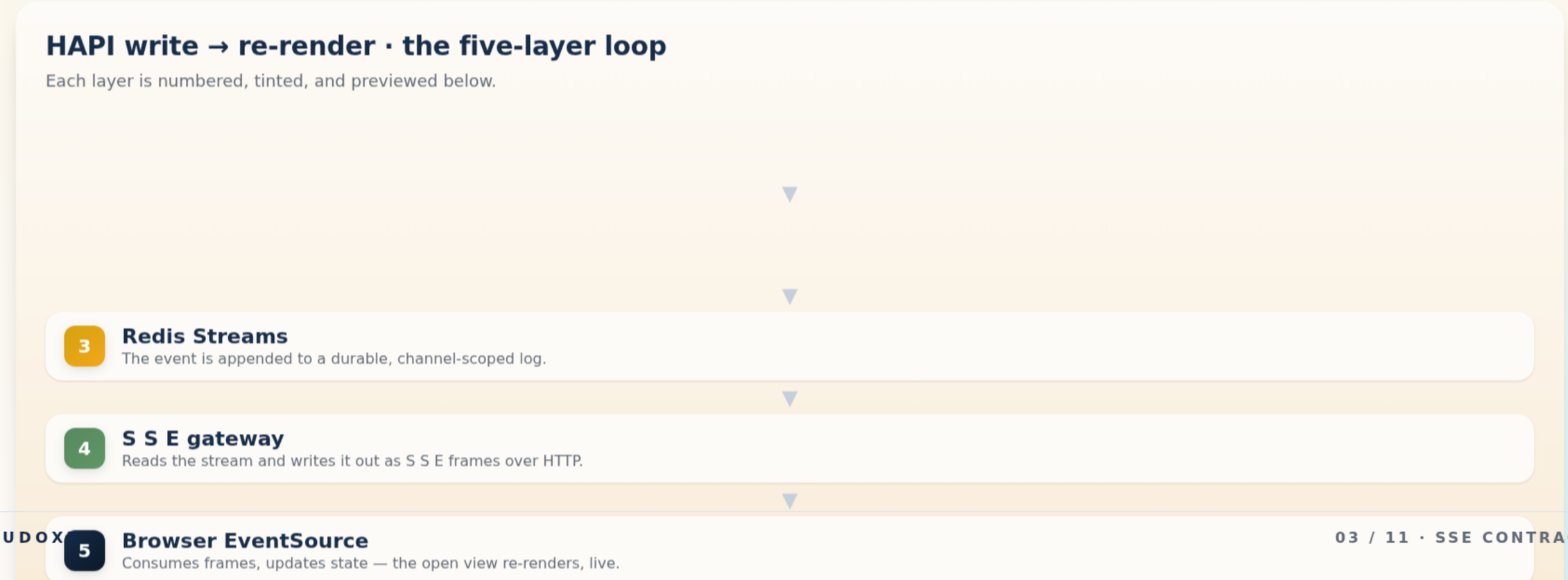
One-way over plain HTTP

Server to client, so proxies and CDNs already understand it. The browser auto-reconnects, and replay is native through Last-Event-ID.

— DIAGRAM · THE SPINE OF THE WHOLE TOPIC

One mutation travels five layers — datastore to DOM.

This is the map. A change at the datastore ends up re-rendering in the open view, and the loop only closes when layer five updates the chapter-fourteen component. Every later slide zooms into exactly one of these nodes — hold this picture.



— LAYER 1 · WALKTHROUGH — ILLUSTRATIVE SKELETON

A write becomes an event — at the edge of the datastore.

Same machinery as chapter ten: an annotated class, a method bound to a Pointcut in HAPI's lifecycle. Here it is scoped to mutations — a write Pointcut reads the changed resource and emits a domain event onto the BFF bus.

— ILLUSTRATIVE · @INTERCEPTOR — WRITE HOOK (JAVA)

@Hook(Pointcut.STORAGE_PRECOMMIT_RESOURCE_UPDATED) void onWrite(IBaseResource r) { bus.emit(toDomainEvent(r)); }

Read the changed resource, turn it into a domain event — patient.updated, observation.created — and hand it to the BFF's event bus.

- Same interceptor + Pointcut-enum pattern as Ch 10 — here, mutation-scoped
- Open question — exact Pointcut (precommit vs subscription path) reconciled with Ch 10 / Ch 12 in live code
- Representative write hook · not a hard claim — the takeaway is the shape, not the signature

— LAYER 3 · CONCEPT — THE DURABLE BUS

Why not push straight to the browser? Because connections drop.

Redis Streams is an append-only log built for exactly this. The missed events are still there when the client reconnects — that is what a stream buys you over a fire-and-forget push.

— ILLUSTRATIVE · STREAM COMMANDS

XADD sse:{clinic}:patient * type patient.updated id 123 — entry gets a monotonic ID

MAXLEN ~ N caps the log into a ring buffer. XRANGE replays a window of past entries. Consumer groups fan the same stream across gateway instances.

• XADD

Append

Each entry gets a monotonic, time-ordered ID — always increasing.

• MAXLEN

Ring buffer

Caps the log so it never grows without bound — memory stays predictable.

• XRANGE

Replay

Read back a window of past entries from any ID forward.

• GROUPS

Fan-out

Several gateway instances consume the same stream horizontally.

• **The stream entry ID == the S S E id == Last-Event-ID — the entire replay contract**

• **Size MAXLEN to a realistic reconnect window — long enough to cover a drop, short enough to bound memory**

— FRAMEWORK ANCHOR · TYPED CONTRACT — CONCEPTUAL

One Zod schema. Producer and consumer can't drift.

— CONCEPTUAL · A CONTRACT PACKAGE LIKE @DDX/SSE-CONTRACT

defineEvent('patient.updated', z.object({ id, version, clinicId })) — EVENTS enumerates the valid set — SseClient consumes with inference

defineEvent pairs an event name with a Zod payload schema. EVENTS registers every valid event. A typed SseClient consumes them with end-to-end type inference. This is the pattern — the exact v3 API surface is illustrative, confirmed against the real package before any code ships.

Ava

One shared package centralizes a typed registry. There is literally one definition of each payload.

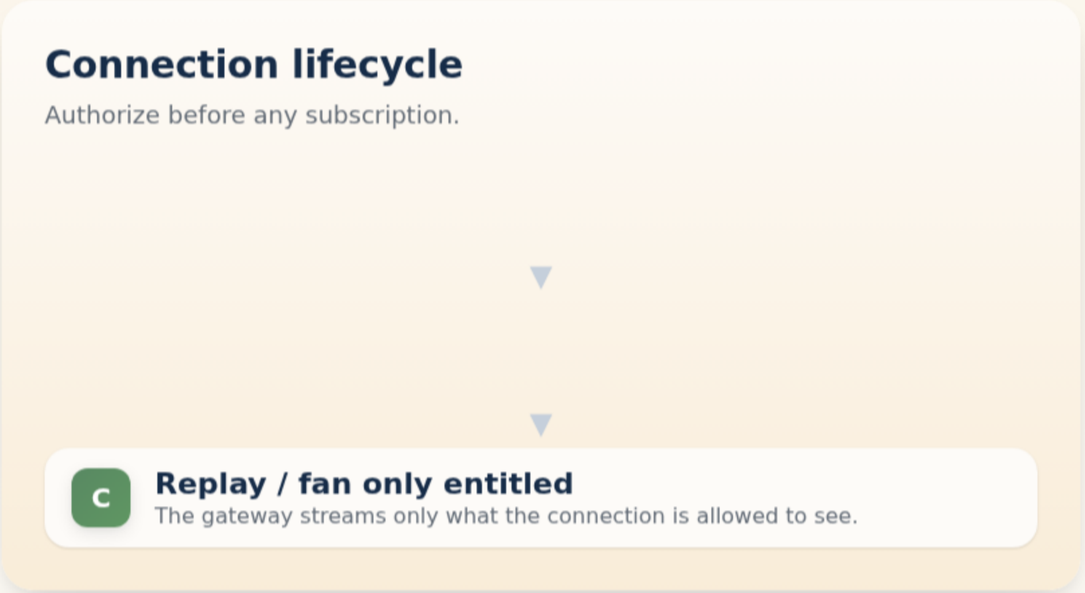
Marcus · one source of truth

Both ends import the same schema · payload validates at the boundary · TypeScript types flow from one Zod source. The server can't emit a shape the client didn't expect.

— LAYER 4 · DIAGRAM — PARTITION ISOLATION REACHES THE PUSH CHANNEL

Authorize first. Then subscribe — to your tenant's streams only.

In a platform with many clinics, an event for one tenant must never reach another. The same X-Clinic-ID that picks a partition in chapter eleven scopes the live channel here.



— FORBIDDEN · STRUCK OUT

Tenant A → tenant B's stream

Partition isolation does not stop at the database — it runs all the way to the browser. A connection authorized for clinic A can never subscribe to clinic B's stream key.

- Ch 11 — X-Clinic-ID picks the partition
- Ch 13 — where the auth lives

— LAYER 5 · WALKTHROUGH — THE CLIENT PRIMITIVE

EventSource consumes the frame. The chapter-fourteen view re-renders.

Construct an EventSource at the stream route. Handle messages — the generic onmessage or named-event listeners that match the contract. Parse each frame, validate against the same shared Zod schema, then update state or invalidate the cache.

— ILLUSTRATIVE · BROWSER (TYPESCRIPT)

```
const es = new EventSource('/events'); es.addEventListener('patient.updated', e => {
const p = PatientUpdated.parse(JSON.parse(e.data)); cache.invalidate(p.id); });
```

Validation at the boundary reuses the slide-six schema — the resource re-renders with fresh data, no refresh.

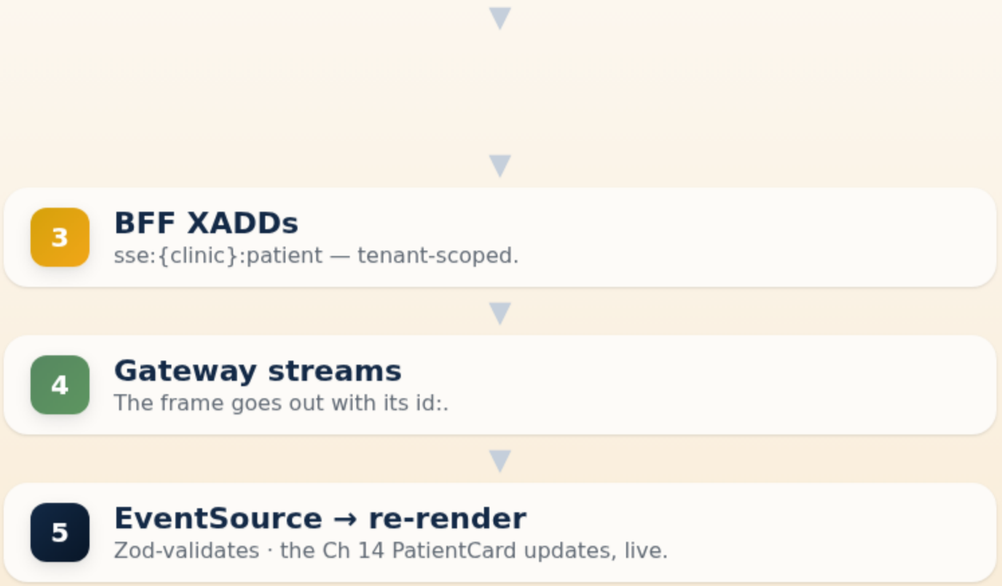
- **Auto-reconnect on drop — interval tunable by the stream's retry: field**
- **Browser stores each id: and resends it as Last-Event-ID on reconnect**
- **Gateway XRANGEs forward from that ID — replays exactly the events the client missed**
- **The browser primitive + the Redis ring buffer — two halves of one replay contract**

— SYNTHESIS · ONE PATIENT EDIT · ALL FIVE LAYERS

One mutation, five layers, three guarantees.

A single Patient edit, end to end

Trace one concrete change down the rail.



— TYPED

Zod guards both ends

The schema validates at the emit and at the receive — no shape the client didn't expect.

— TENANT-SCOPED

The clinic rides the key

The stream key and the authorized connection both carry the clinic — isolation to the browser.

— REPLAYABLE

Entry ID == Last-Event-ID

On reconnect the gateway XRANGEs forward from the last id — nothing silently missed.

— PRODUCTION · THE SCARS — DEMO VS TRUST AT 3AM

Four things that bite in production. Each has a mitigation.

The difference between a demo that works and a system you trust at three in the morning is honesty about these four.

AT-LEAST-ONCE

Not exactly-once

You will sometimes see an event twice.

Handlers MUST be idempotent — re-applying patient.updated is harmless

BACKPRESSURE

Slow client + cap

Events can age out of the buffer before they are read.

On a gap — Last-Event-ID older than the buffer head — fall back to a full refetch

BUFFERING

Proxy / CDN stall

Response buffering can hold an S S E stream silent.

Disable buffering on the stream route

CONN LIMIT

~6 per host on h1

Browsers cap connections per host on HTTP/1.1.

Multiplex over h2 — or share one stream

Failure modes

Each one has a mitigation — never silently miss a change

— TRY IT YOURSELF · RECAP · NEXT

Open a stream. Drop it. Watch it replay.

The BFF served it, Next.js rendered it — and now it stays live. That closes the app-layer arc of Act III.

— RUN THIS — THEN SKETCH THE CONTRACT

const es = new EventSource('/events'); es.onmessage = e => console.log(e.lastEventId, e.data); — then kill the connection and watch the reconnect + Last-Event-ID replay

Log each event's id and parsed payload, drop the connection mid-stream, confirm the browser auto-reconnects and replays what you missed. Then sketch one defineEvent('patient.updated', ZodSchema) entry and a matching SseClient handler so producer and consumer share one typed contract.

Recap

- S S E transport
- The 5-layer loop
- Redis Streams replay
- Typed Zod contract
- Tenant scoping