

— WELCOME · COURSE INTRO

FHIR R4 & HAPI — from resources to production.

From your very first Patient document to a multi-tenant healthcare platform running in production. If you are a developer comfortable with REST and JSON, and you want to build something real in healthcare, this course is for you. You bring the engineering — we bring FHIR.

Who this is for

● Developers fluent in REST + JSON

● No clinician or hospital background needed

— THE PAYOFF · WHAT YOU'LL BUILD

Four things you ship — mapped onto five acts.

● HAPI SERVER

Multi-tenant FHIR in Java

A production HAPI FHIR server — the spine every other piece talks to.

● NESTJS BFF

FHIR shaped for the UI

A backend-for-frontend that adapts raw FHIR for your front end.

● NEXT.JS UI

Real clinical data

A front end rendering live Patients, Conditions, and Observations.

● MCP SERVER

FHIR for LLM agents

A Model Context Protocol server exposing FHIR to AI agents.

The five-act spine

● I · Spec — resources · datatypes · terminology · bundles

● II · HAPI — config · profiling · search · interceptors · subscriptions

● III · App layer — BFF · Next.js · SSE · seeders

● IV · Security + agents — SMART · RBAC · MCP · telemedicine

● V · The honest pitfalls — scars from real production

— HOW TO FOLLOW ALONG

About 150 minutes — hands-on, in two voices.

~150m

TOTAL RUNTIME

Watch straight through for the full arc, or jump to the chapters that matter to your role.

hapi.fhir.org

FREE PUBLIC SANDBOX

Everything is hands-on against a free, public HAPI server. No install needed for the first taste.

Try it

EVERY CHAPTER ENDS WITH A RECAP

One concrete command you run against the sandbox. Pause, try it, come back.

**Ava · host**

And every chapter is hands-on against the public HAPI sandbox.

**Marcus · expert**

Each chapter closes with one command to run yourself. Pause, try it, return.

— CHAPTER 01 · COLD OPEN

FHIR in 2026 — and why *R4* is still the answer.

Every pre-2024 tutorial tells you to ship on the latest FHIR version. In 2026 that advice is wrong. Epic and Oracle Health both declined to commit to R5, the industry is waiting for R6, and R4 is the only realistic production target through roughly 2031. That is the story we unpack.

What this topic covers

● Spec lineage

● The 2026 surprise

● Mental model

● What FHIR is NOT

— CONCEPT · SPEC LINEAGE

From HL7 v2 in 1989 to FHIR R6 ballot in 2026.

Thirty-seven years of healthcare interoperability standards — and one inflection in 2019.

● 1989

HL7 v2

Pipe-delimited messages. Still runs most lab feeds in production today.

● 2005

HL7 v3 / CDA

XML-heavy RIM model. Adopted narrowly; CDA survives for clinical documents.

● 2014

FHIR DSTU

Grahame Grieve's REST + JSON reboot. The first standard that web developers could read.

● OCT 2019 · ACTIVE

FHIR R4

First normative release. The version every regulator, EHR vendor, and SMART app speaks today.

● MAR 2023

FHIR R5

Published — but Epic and Oracle Health declined to commit. Effectively skipped at vendor scale.

● JAN 2026 · BALLOT

FHIR R6

Currently in ballot. Industry consensus: R4 → R6 direct path. Production migration ~2028 onward.

● You ship on R4

● R6 ballot in flight · 2026

— THE 2026 SURPRISE

R5 shipped. Then the two largest EHRs said no.

0

MAJOR EHR VENDORS COMMITTING TO R5

Of the top five US EHRs by installed base — Epic, Oracle Health (Cerner), Meditech, Allscripts, Athena — none has published an R5 commitment.

~2031

EARLIEST REALISTIC R6 PRODUCTION MAJORITY

R6 ballot January 2026 · normative ~2027
· early adopters 2028 – 2029 · majority production ~2031.

R4

THE ONLY PRODUCTION TARGET TODAY

Every EHDS exchange, every TEFCA FAST endpoint, every US Core 7 profile — all R4. Build on R4.



Ava



Marcus · production scar

Both. Epic said no to R5 — that decided it for the rest of us.

— COMPARISON · THE VERSION DECISION

R4 · R4B · R5 · R6 — which one ships today?

● R4 · 2019 · NORMATIVE

Use this. Always.

Every regulator (ONC, EHDS, TEFCa) writes R4 conformance. Every EHR vendor speaks R4. Every HAPI release supports R4 as first-class.

● R4B · 2022 · MAINTENANCE

Subscriptions only.

Patch release that ships the new Subscriptions framework. Use R4B only if you need server-sent FHIR subscriptions; otherwise stay on R4.

● R5 · 2023 · SKIPPED

Do not target.

Epic and Oracle Health declined to commit. No EHDS or TEFCa mandate references R5. Building greenfield on R5 in 2026 strands the deployment.

● R6 · 2026 · BALLOT

Watch, don't ship.

Ballot opened January 2026. Normative likely 2027. Plan a migration path from R4 — but production runs on R4 through ~2031.

● Ship · R4

● Niche · R4B (subscriptions)

● Skip · R5

● Watch · R6 ballot Jan 2026

— CONCEPT · FIVE-PILLAR MENTAL MODEL

FHIR in five pillars — everything else is detail.

• TRANSPORT

REST + JSON

Plain HTTP verbs on resource URLs.
Anything that speaks HTTP and JSON can read or write FHIR.

• DOMAIN

~150 Resources

Patient, Observation, Encounter,
Condition, Medication and ~145 more.
Typed, versioned, with defined cardinality.

• COMPOSITION

Bundles

Transaction, batch, document, message,
searchset. Atomic writes and grouped
reads — same envelope shape every time.

• VOCABULARY

Terminology

CodeSystem, ValueSet, ConceptMap.
Where SNOMED, LOINC, ICD-10 and
RxNorm hang off the resource model.

• CONSTRAINT

Profiles

StructureDefinition narrows base
resources to your jurisdiction — US Core,
IPS, EHDS profiles. Validation is the
contract.

Every chapter that follows lives inside one of these pillars. Chapter 2 dissects a resource. Chapter 4 builds bundles. Chapter 7 wires terminology. Chapter 9 ships profile validation.

— WHY NOW · REGULATORY PRESSURE

Four regulators, four deadlines — all writing R4.

• EU · EHDS 2025/327

European Health Data Space

In force March 2026. Cross-border patient-summary exchange mandatory by March 2029. Conformance profile: FHIR R4 + IPS.

• March 2026 · EHDS in force

• US · TEFCA FAST

FHIR At Scale Taskforce

January 2026 production milestone. QHIN-to-QHIN FHIR exchange via TEFCA. R4 endpoints only.

• January 2026 · TEFCA FAST

• US · ONC HTI-1

Certified Health IT

Decision support, USCDI v3, and SMART-on-FHIR endpoints required for certification. R4 is the baseline.

• ONC HTI-1 certified

• US · US CORE 7.0.0

Profile pack · mandatory

Current mandatory US implementation guide for ONC certification. Built on R4. No US Core release exists for R5.

• US Core 7 mandatory

— SCOPE · HONEST BOUNDARIES

What FHIR IS — and what it is NOT.

FHIR IS

A REST API SPECIFICATION

Verbs, URLs, status codes

An exchange contract — how systems read, write, and search clinical data over HTTP.

A DATA INTERCHANGE STANDARD

JSON · XML · Turtle

A typed wire format with versioning, references, and a defined resource model.

A CLINICAL RESOURCE MODEL

~150 typed Resources

Shared semantics for Patient, Observation, Encounter and the rest of the clinical domain.

FHIR is NOT

• A DATABASE

Bring your own storage

HAPI stores FHIR in Postgres. Aidbox uses Postgres + JSONB. Firely uses SQL Server. You choose the engine.

• A WORKFLOW ENGINE

No state machine

FHIR carries Task and ServiceRequest resources but does not orchestrate them. BPMN, Camunda, or your service layer drive workflow.

• A SECURITY MODEL

SMART- on-FHIR sits on top

FHIR defines no auth. OAuth 2.0 + SMART-on-FHIR scopes + UDAP layer above. The spec is intentionally transport-agnostic on identity.

— RECAP · THE COURSE SPINE

Four frameworks. One FHIR-native stack.

What you will build across the next chapters: a production-shaped FHIR system, end to end.

• SERVER

HAPI FHIR

Open-source Java reference server.
Postgres-backed JPA storage, validation,
terminology, subscriptions.

● Ch 2 – 5 · HAPI

• BFF

NestJS

Backend-for-frontend in TypeScript.
Authentication, aggregation, audit logging,
SMART scopes.

● Ch 6 – 8 · NestJS BFF

• UI

Next.js

React clinical UI with server components,
streaming, and FHIR-typed forms.

● Ch 9 – 11 · Next.js UI

• AGENT

MCP server

Model Context Protocol bridge that
exposes FHIR resources as typed tools for
LLM clients.

● Ch 12 · MCP agent

— TRY IT YOURSELF · 30 SECONDS

One curl. One jq. A real FHIR R4 server.

The HAPI public sandbox at hapi.fhir.org runs a live R4 endpoint. Hit its capability statement to see the version, the software name, and how many resource types it serves.

— RUN THIS IN YOUR TERMINAL

```
curl https://hapi.fhir.org/baseR4/metadata | jq '.fhirVersion, .software.name, (.rest[0].resource | length)'
```

You should see `"4.0.1"`, the HAPI server name, and a count near 150 — the live resource catalogue. That is the surface area we spend the rest of the course mastering.

— NEXT CHAPTER

Chapter 02 — Resource anatomy: Patient, Observation, and the typed model in detail.

● hapi.fhir.org · public R4 sandbox

● Next · Ch 2 · Resource anatomy

— CHAPTER 02 · FOUNDATIONS · BEAT 1 — OPEN THE BOX

Every FHIR resource has the same *skeleton*.

By the end of this chapter you will name the six anatomical parts every FHIR resource shares, tell extension apart from modifierExtension — and know why one of those can break clinical safety — and never again confuse id with identifier.

Learning goals

● Six anatomical parts

● extension vs modifierExtension

● id vs identifier

— CONCEPT · ANATOMY

Six anatomical parts — then the clinical content.

Every FHIR R4 resource — Patient, Observation, Encounter, all hundred and fifty of them — opens with the same six fields. Memorize these and you can read any FHIR resource on first sight.

● 1 · IDENTITY

resourceType

The type tag. Patient, Observation, Encounter — tells every parser what shape to expect.

● 2 · IDENTITY

id

Server-assigned logical key. Opaque. Lives in the URL: /Patient/abc-123.

● 3 · PROVENANCE

meta

versionId, lastUpdated, profile, security, tag — metadata about the resource itself.

● 4 · HUMAN

text

XHTML narrative. The human-readable safety net every clinician trusts.

● 5 · EXTENSIBILITY

extension

Adds locale, specialty, payer fields the core spec deliberately omits.

● 6 · MUST-UNDERSTAND

modifierExtension

Same shape — but CHANGES meaning. Ignore one you do not understand and you misread the resource.

● THEN · DOMAIN

name, gender, birthDate ...

The actual clinical content. Different per resource type. Patient has name; Observation has value.

● MENTAL MODEL

Skeleton + organs

Six bones are universal. Organs are domain-specific. Same skeleton; different organs.

— CONCEPT · NARRATIVE · TEXT

Every resource carries a clinician-readable paragraph.

The text element is a chunk of XHTML — wrapped in `<div xmlns="http://www.w3.org/1999/xhtml">` — that renders the resource as prose a doctor can scan. If the structured data and the narrative disagree, that is a patient-safety incident.

— PATIENT.TEXT · EXCERPT

```
"text": {  
  "status": "generated",  
  "div": "<div xmlns=\"http://www.w3.org/1999/xhtml\"><p><b>Peter James  
Chalmers</b>, male, born 1974-12-25, MRN 12345.</p></div>"  
}
```

Servers render the div; clients display it verbatim. Always render the narrative — clinicians trust it before they trust your structured fields.

text.status — four values

● **generated** — derived from structured data

● **extensions** — includes extension data

● **additional** — info NOT in structured

● **empty** — no narrative

— CONCEPT · CONTAINED · THE EXCEPTION

Inline a whole resource — only when it has no independent life.

contained[] holds resources that have no independent existence on the server. No URL of their own. Referenced locally with a #fragment. The rule of thumb — almost never use it.

— CAN-BE · INLINE VIA CONTAINED

One-off context · no independent existence

A visiting consultant Practitioner attached to a single Observation. Not in your directory. Will never be referenced again.

```
"contained": [{ "resourceType":  
  "Practitioner", "id": "prac1", "name": [...]  
}],  
"performer": [{ "reference": "#prac1" }]
```

● contained can't be searched

● contained can't be referenced elsewhere

— SHOULD-BE · USE A REFERENCE

Anything with a chance of independent existence

Real Practitioners, Organizations, Locations — make them first-class server resources and reference by URL.

```
"performer": [{ "reference":  
  "Practitioner/abc-123" }]
```

● Reference — deep dive in Ch 3

— CONCEPT · EXTENSION · THE ESCAPE HATCH

Core stays minimal. Everything country-specific lives in extensions.

FHIR core resources cover what the whole world agrees on. Everything country-specific, specialty-specific, payer-specific goes here. Each extension has two parts — a url that is the contract, and a value[x] that is the data.

— PATIENT.EXTENSION · US CORE RACE

```
"extension": [{  
  "url": "http://hl7.org/fhir/us-core/StructureDefinition/us-core-race",  
  "extension": [{  
    "url": "ombCategory",  
    "valueCoding": {  
      "system": "urn:oid:2.16.840.1.113883.6.238",  
      "code": "2106-3",  
      "display": "White"  
    }  
  }  
}]
```

That is how a US Patient declares race — without the core Patient spec needing a race field. The OMB category code is the canonical US race vocabulary.

— CONCEPT · MODIFIEREXTENSION · THE DANGEROUS COUSIN

Some extensions don't add — they *change meaning*.

MUST

REJECT IF YOU DON'T UNDERSTAND IT

A processor that encounters a modifierExtension it does not recognize MUST reject the resource. Not warn. Reject.

1

RULE THAT SEPARATES SAFE FROM UNSAFE

This is the single hardest rule in FHIR — and the line between a safe client and a clinical-safety bug.

mg/dL

EXAMPLE · ISO21090-PQ-TRANSLATION

A Quantity carries a translated unit. Ignore the modifierExtension and you ship the wrong unit downstream.



Ava



Marcus · production scar

— CONCEPT · META · FOUR FACETS

Where did this come from, what does it claim to be, who can see it, how do I find it?

meta is the resource's metadata about itself. Four facets — provenance, conformance, security, indexing. Every FHIR server populates these, every client should read them.

• PROVENANCE

versionId · **lastUpdated** · **source**

versionId increments on every write.
lastUpdated is the server timestamp.
source says where the data originated.

• CONFORMANCE

profile[]

Array of profile URLs this resource claims to conform to — US Core Patient, IPS Patient, your custom profile.

• SECURITY

security[]

Security labels — confidentiality, sensitivity, break-the-glass markers that gate access decisions.

• INDEXING

tag[]

Free-form labels you use for workflow, filtering, batch operations. Server-side or client-side conventions.

● **"meta.versionId": "3"**

● **"meta.profile": ["http://hl7.org/fhir/us-core/StructureDefinition/us-core-patient"]**

● **"meta.security": [{ "code": "R" }]**

● **"meta.tag": [{ "code": "needs-review" }]**

— CONCEPT · THE ROOKIE TRAP

id is for the database. identifier is for the real world.

They sound the same. They are not the same. id is singular, lower-case, server-assigned, opaque. identifier is a plural array of business identifiers — MRN, SSN, NPI — each with system, value, use, period.

— ID · SERVER LOGICAL KEY

"id": "abc-123"

Singular. Lower-case. Server-assigned. Opaque.
Appears in the URL path — /Patient/abc-123.
Has no meaning outside that one server.

● One per resource

● Lives in the URL

— IDENTIFIER · BUSINESS KEYS

**"identifier": [{
 "system":
 "http://hospital.example.org/mrn",
 "value": "MRN-44721",
 "use": "official",
 "period": { "start": "2018-03-01" }
}]**

Plural array. Each entry has a system (namespace), value, use (official · usual · temp · old), and optional period of validity.

● MRN

● SSN

● NPI

— WALKTHROUGH · [HAPI.FHIR.ORG/BASER4/PATIENT/EXAMPLE](https://hapi.fhir.org/baser4/patient/example)

Six anatomical parts, then the clinical content — in one real document.

— PATIENT/EXAMPLE · TRIMMED

```
{
  "resourceType": "Patient",                // 1
  "id": "example",                          // 2
  "meta": {
    "versionId": "1",
    "lastUpdated": "2024-09-12T08:14:22Z",
    "profile": ["http://hl7.org/fhir/us-core/StructureDefinition/us-core-patient"]
  },                                         // 3
  "text": {
    "status": "generated",
    "div": "<div xmlns=\"http://www.w3.org/1999/xhtml\">Peter Chalmers, male,
b.1974</div>"
  },                                         // 4
  "extension": [{ "url": "...us-core-race", "extension": [...] }], // 5
  // no modifierExtension on this Patient — but the slot is always there // 6
  "identifier": [{ "system": "urn:oid:1.2.36.146.595...", "value": "12345", "use": "usual" }],
  "name": [{ "family": "Chalmers", "given": ["Peter", "James"] }],
  "gender": "male",
  "birthDate": "1974-01-01",
  "address": [{ "line": "123 Main St", "city": "Anytown", "state": "CA", "postalCode": "90210" }],
  "contact": [{ "name": "John Doe", "relationship": "C", "telecom": [{"system": "phone", "value": "555-123-4567"}]}],
  "communication": [{ "language": "en", "text": "Patient history and clinical notes." }],
  "communicationRequest": [{ "request": "Request for a second opinion on the patient's condition." }],
  "communicationRequestResponse": [{ "response": "Response to the request for a second opinion." }],
  "communicationRequestResponseText": "Response to the request for a second opinion."
}
```


— TRY IT YOURSELF · 60 SECONDS

Slice the anatomy out of a real Patient.

Hit the public HAPI sandbox, pull Patient/example, and project just the anatomical parts — id, identifier, meta, narrative status, and how many extensions and modifierExtensions the resource carries.

— RUN THIS IN YOUR TERMINAL

```
curl https://hapi.fhir.org/baseR4/Patient/example | jq '{id, identifier, meta, narrative_status: .text.status, extension_count: (.extension // [] | length), modifier_extension_count: (.modifierExtension // [] | length)}'
```

You will see the logical id, the business identifiers, the meta block, the narrative status, and counts of both extension flavors. Then go hunt — find a Patient on the sandbox that has a modifierExtension and tell yourself in one sentence what semantic it flips.

— NEXT CHAPTER

Chapter 03 — Datatypes & References: CodeableConcept, Reference, Quantity, Period, Identifier. The building blocks inside the building blocks.

● hapi.fhir.org · public R4 sandbox

● Hunt a modifierExtension in the wild

● Next · Ch 3 · Datatypes & References

— CHAPTER 03 · FOUNDATIONS · BEAT 1 — THE CELLS AND THE GRAPH

Datatypes are the cells. References are the graph.

Last chapter we cracked open a resource and named its six anatomical parts. Inside those parts lives another layer — the datatypes. By the end of this chapter you can read FHIR's five workhorse datatypes, wire resources into a graph with References, and dodge the two classic traps that quietly corrupt real systems.

What you will be able to do

- The five datatypes — the spine of FHIR
- References — the graph that connects them
- The UCUM and MRN gotchas

— CONCEPT · WHY FHIR SHIPS 50+ DATATYPES

Primitives are the atoms. Complex datatypes are built from them.

The core resources stay small precisely because they reuse a fixed set of typed building blocks everywhere instead of reinventing fields. We are not learning all fifty — just the five you meet in almost every payload.

— PRIMITIVES — THE ATOMS

● string ● boolean ● dateTime

● decimal ● uri ● code

Single, indivisible values. Every richer datatype is assembled from these.

— COMPLEX — BUILT FROM PRIMITIVES

● Quantity ● CodeableConcept ● Reference

● Period ● Identifier

The five workhorses for this chapter. Master these and you can read ninety percent of real FHIR.

— DATATYPE 1 · QUANTITY — A NUMBER DONE CAREFULLY

A number, a unit, and the code the math actually trusts.

— A REAL QUANTITY

```
"valueQuantity": { "value": 70, "unit": "kg", "system": "http://unitsofmeasure.org",  
"code": "kg" }
```

`value` is the number. `unit` is a human display string. `system` points at UCUM, unitsofmeasure.org. `code` is the UCUM symbol — like `kg` or `mm[Hg]`. Sub-flavors: Money carries a currency, Age and Duration fix the time dimension, SimpleQuantity drops the comparator.

● `code` = the math — servers compare on this

● `unit` = the display — for humans only

— THE SCAR · 70 KG VS 70 LBS

Display says "70 kg". The code underneath says "lb".

Servers compare on `code`, never on the display `unit`. Your screen says one thing, your math does another. Wrong display with right data is still a patient-safety bug. Always trust the `code`.

— DATATYPE 2 · CODEABLECONCEPT — CODED PLUS HUMAN

One concept, possibly many vocabularies, plus the words a human typed.

— A REAL CODEABLECONCEPT

```
"code": { "coding": [ { "system":  
  "http://loinc.org", "code": "29463-7",  
  "display": "Body weight",  
  "userSelected": true }, { "system":  
  "http://snomed.info/sct", "code":  
  "27113001", "display": "Body weight" }  
], "text": "Body weight" }
```

A **coding** array holds zero or more entries; each is a **system**, a **code**, and a **display**. The free-text **text** sits beside them.

— HOW TO READ IT

One Coding when a single terminology is unambiguous. Multiple Codings when the same concept lives in LOINC and SNOMED and a local code — a translation.

userSelected: true marks the one the clinician actually picked; the rest are machine-added mappings.

The **text** field is the legal fallback — it wins for display when no code resolves.

Where these codes come from, and how binding strengths bind them, is chapter four.

● **coding[]** — zero or more typed codes

● **text** — the human fallback, legally meaningful

— DATATYPE 3 · REFERENCE — THE TYPED POINTER

A pointer from one resource to another. This is what makes a graph.

rel

RELATIVE

"reference": "Patient/123" — resolved against the same server. The everyday form.

abs

ABSOLUTE

"reference":
"https://other/fhir/Patient/123" — a full URL pointing at another server entirely.

#

FRAGMENT

"reference": "#prac1" — points at a contained resource inside the same document.

— TWO SAFETY NETS

display is a human-readable label — an unresolvable reference can still say "Dr. Chen". And **Patient/123/_history/2** pins an immutable version, essential for audit. Resolution order: URL, then fragment, then **identifier** logical match, then **display**.

— THE SCAR · REFERENTIAL INTEGRITY

The server does NOT check the target exists.

FHIR will happily let you reference a Patient that was never created. Dangling references are your problem to catch, not the server's.

— THE ANCHOR · ENCOUNTER + OBSERVATION LINKING

One Observation. Every datatype clicks into one graph.

— OBSERVATION — A BODY-WEIGHT READING

```
{ "resourceType": "Observation", "status": "final", "code": { "coding": [ { "system":  
"http://loinc.org", "code": "29463-7", "display": "Body weight" } ] }, [4] "subject": {  
"reference": "Patient/123" }, [1] "encounter": { "reference": "Encounter/456" }, [2]  
"performer": [ { "reference": "Practitioner/789", "display": "Dr. Chen" } ], [3]  
"effectivePeriod": { "start": "2026-05-24T09:00:00Z", "end": "2026-05-24T09:05:00Z" },  
[6] "valueQuantity": { "value": 70, "unit": "kg", "system": "http://unitsofmeasure.org",  
"code": "kg" } } [5]
```

● 1 · subject — Reference(Patient)

● 2 · encounter — Reference(Encounter)

● 3 · performer — Reference(Practitioner)

● 4 · code — CodeableConcept (LOINC)

● 5 · valueQuantity — Quantity (UCUM)

● 6 · effectivePeriod — Period

Three References stitch four resources into a graph — Observation, Patient, Encounter, Practitioner. The datatypes carry the clinical payload. References are the graph; datatypes are the cells. Read this one document and you have read FHIR.

— COMPARISON · PERIOD VS RANGE VS RATIO

Three shapes that look alike and mean very different things.

Period

AN INTERVAL IN TIME

start + end. An Encounter's duration, a coverage window.

Range

A BAND OF MEASUREMENT

low + high. A target blood-pressure window.

Ratio

A RELATIONSHIP

numerator / denominator. A titer of one to sixty-four, or a dose per volume.

— THE PERIOD GOTCHA

A start with no end does not mean missing data. It means ongoing.

The encounter is still open; the medication is still active. Never auto-close an open period — you would be inventing an end that never happened.

— DATATYPE 5 · IDENTIFIER — BEYOND SYSTEM AND VALUE

The facets that make a business key production-grade.

— THREE IDENTIFIERS · TWO DISTINCT SYSTEMS

```
"identifier": [ { "use": "official", "type": { "coding": [ { "code": "MR" } ] }, "system":
"urn:oid:2.16.840.1.113883.3.42", "value": "MRN-44721", "assigner": { "reference":
"Organization/hosp-a" } }, { "use": "old", "system": "urn:oid:2.16.840.1.113883.3.42",
"value": "MRN-10093", "period": { "end": "2024-01-01" } }, { "use": "usual", "system":
"https://hosp-b.example.org/mrn", "value": "B-55120", "assigner": { "reference":
"Organization/hosp-b" } } ]
```

● use — official, usual, temp, secondary, old

● type — a CodeableConcept (MR, SSN, DL)

● period — a validity window

● assigner — Reference to the issuing Organization

— THE ANTI-PATTERN · MRN PER FACILITY

Three MRNs across two hospitals = three Identifiers, two distinct systems.

Each issuing facility needs its own **system** namespace — an OID like **urn:oid:2.16.840...** or a URI. Collapsing them under one system merges namespaces that should never touch. When an

MRN changes, never delete the old one — demote it to **use: old** with a **period.end**, and add the

new one as official. Identity has history.

— SYNTHESIS · OBSERVATION + THE ENCOUNTER IT REFERENCES

Every datatype from the chapter, living in two real documents.

— OBSERVATION

```
{ "resourceType": "Observation",  
  "encounter": { "reference":  
    "Encounter/456" }, [1] "code": {  
    "coding": [ { "system": "http://loinc.org",  
      "code": "29463-7" }, { "system":  
        "http://snomed.info/sct", "code":  
          "27113001" } ] }, [2] "performer": [ {  
      "reference": "Practitioner/789",  
      "display": "Dr. Chen" } ], [6]  
  "valueQuantity": { "value": 70, "code":  
    "kg", "system":  
      "http://unitsofmeasure.org" } } [3]
```

— ENCOUNTER/456

```
{ "resourceType": "Encounter", "id":  
  "456", "status": "finished", "subject": {  
    "reference": "Patient/123" }, "period": {  
      "start": "2026-05-24T08:30:00Z", "end":  
        "2026-05-24T09:30:00Z" } } [4] //  
Patient/123 link "identifier": [ { "use":  
  "official", "system":  
    "urn:oid:2.16.840...", "value": "MRN-  
44721" } ] [5]
```

● 1 · Reference relative form

● 2 · CodeableConcept — two Codings

● 3 · Quantity — UCUM code

● 4 · Period — start + end

● 5 · Identifier — use + system

● 6 · display fallback on a Reference

— TRY IT YOURSELF · 60 SECONDS

Pull a live Observation. Read its concept and its value.

Run this against the public HAPI sandbox. Pull one Observation and slice out the CodeableConcept code with its system, plus the Quantity.

— RUN THIS IN YOUR TERMINAL

```
curl https://hapi.fhir.org/baseR4/Observation?_count=1 | jq '.entry[0].resource | {code: .code.coding[0].code, system: .code.coding[0].system, value: .valueQuantity}'
```

That gives you the CodeableConcept's first code and its system, plus the Quantity. Then go hunt — find an Observation whose **code** carries two Codings, and name both systems.

— NEXT CHAPTER

Chapter 04 — Terminology.

We read the codes today; next we learn where they come from, and how binding strengths decide which ones are even allowed.

● hapi.fhir.org · public R4 sandbox

● Next · Ch 04 · Terminology

— CHAPTER 04 · TERMINOLOGY BINDINGS · BEAT 1 — WHICH CODES?

Last chapter we built the Coding. Now — which systems fill it.

A CodeableConcept holds a system, a code, and a display. This chapter answers the question that leaves open. We map the five code systems clinicians actually use, learn the four binding strengths that decide what is allowed, and run the three terminology operations that make codes queryable.

What you will be able to do

● Code systems — SNOMED · LOINC · RxNorm · ICD-10 · UCUM

● Binding strengths — required to example

● Tx operations — expand · validate-code · translate

— COMPARISON · FIVE CODE SYSTEMS AT A GLANCE

Five systems carry almost all of FHIR's meaning.

Each has a clinical purpose and an owner — and four of the five are free worldwide. SNOMED CT is the one exception. That single fact reshapes a lot of deployments.

• FINDINGS · PROCEDURES

SNOMED CT

Clinical findings, problems, body sites.
Owner — SNOMED International.

● LICENSED

• LAB · OBSERVATIONS

LOINC

Laboratory and clinical observations.
Owner — Regenstrief.

● FREE

• MEDICATIONS

RxNorm

Normalized medications. Owner — US
National Library of Medicine.

● FREE

• BILLING

ICD-10-CM

Billing and reimbursement codes.
Owner — the CDC / NCHS.

● FREE

• UNITS OF MEASURE

UCUM

Machine-parseable units. Owner — the
UCUM project.

● FREE

• THE CATCH

Four free, one not

SNOMED CT needs a per-country
national license. We come back to it
next slide.

● Deploy-time blocker

— CONCEPT · SNOMED CT — THE BIG ONE, AND THE LICENSING SCAR

Chest pain is one code. Pointing HAPI at SNOMED is not free.

— CONDITION.CODE — SNOMED CODING

```
{ "system": "http://snomed.info/sct", "code": "29857009", "display": "Chest pain" }
```

The same Coding shape from last chapter — filled in. Hundreds of thousands of concepts cover findings, procedures, and body sites. The content is excellent.



Ava

Here is the scar — no. SNOMED CT requires a national member license.



Marcus · production scar

At deploy time, not design time. NHS Digital in the UK, ANS in France, BfArM in Germany. A public tx server cannot legally hand you all of it.

— CONCEPT · LOINC — WHAT CODES A LAB RESULT

Observation.code is almost always a LOINC code.

If SNOMED is findings, LOINC is measurements — Logical Observation Identifiers, Names and Codes. Drop a LOINC code into Observation.code and any system on earth knows exactly what was measured.

— OBSERVATION.CODE — LOINC CODING

```
{ "system": "http://loinc.org", "code": "2823-3", "display": "Potassium [Moles/volume] in Serum or Plasma" }
```

Serum potassium, two-eight-two-three-dash-three. The same Coding we built last chapter — now carrying the canonical lab identifier.

The relief after SNOMED

● Default for lab and clinical observations

● FREE worldwide — Regenstrief license, attribution only

● No national gate

— COMPARISON · CLINICAL VS BILLING — A LINE WORTH INTERNALIZING

RxNorm normalizes medicines. ICD-10 codes the claim.

A single Condition often carries two Codings in one CodeableConcept — a SNOMED code for clinical reasoning and an ICD-10 code for the bill. Clinical truth and billing truth, side by side. That is why the Coding array from last chapter is an array.

— CLINICAL · RXNORM

Medication.code

```
{ "system":
"http://www.nlm.nih.gov/research/umls/rxnorm",
"code": "857005", "display": "lisinopril 10 MG
Oral Tablet" }
```

Owned by the US National Library of Medicine.
Free in the US. Also fills
MedicationRequest.medicationCodeableConcept.

● RxNorm — FREE (US, NLM)

● ICD-10-CM — FREE (US)

— BILLING · ICD-10-CM

Condition.code

```
{ "system": "http://hl7.org/fhir/sid/icd-10-cm",
"code": "I20.9", "display": "Angina pectoris,
unspecified" }
```

For reimbursement, not primary clinical
reasoning. Free in the US under public domain.

● One CodeableConcept can hold both clinical and billing Codings

— DIAGRAM · UCUM — THE UNIFIED CODE FOR UNITS OF MEASURE

The Quantity we met last chapter — now its code field gets a real system.

Seventy kilograms is not just value seventy, unit kg. The full Quantity carries a system and a machine-parseable code. Always populate the code, not just the unit.

— OBSERVATION.VALUEQUANTITY — UCUM

```
{ "value": 70, "unit": "kg", "system": "http://unitsofmeasure.org", "code": "kg" }
```

For kilograms, unit and code look the same. The difference shows for compounds — millimoles per litre is unit "mmol/L" but the UCUM code is "mmol/L" the machine can parse and convert.

Two fields, two jobs

● unit — the human display string

● code — the literal a machine can compute on

● UCUM is free and open

— FRAMEWORK ANCHOR · VALUESET BINDING STRENGTHS

Four strengths. Two of them can fail validation — two only warn.

Every coded element binds to a ValueSet at one of four strengths. The strength decides whether HAPI's validator raises an error or merely warns. That distinction saves hours of debugging.

● STRENGTH · 1

required

You MUST use a code from the set. HAPI raises a validation ERROR otherwise.

● **HAPI — ERROR · blocks**

● STRENGTH · 2

extensible

Use a code from the set if one fits; extend only when nothing matches.

● **HAPI — ERROR if a fitting code was ignored**

● STRENGTH · 3

preferred

Encouraged but not enforced. Use the set where you reasonably can.

● **HAPI — WARNING · never blocks**

● STRENGTH · 4

example

Purely illustrative. Shows the kind of code expected, nothing more.

● **HAPI — informational only**

● **required + extensible — can fail validation**

● **preferred + example — only ever a warning**

— WALKTHROUGH · THE THREE TERMINOLOGY OPERATIONS

Codes are data. Three operations make them queryable.

\$expand

ENUMERATE THE MEMBER CODES

Run on a ValueSet. Returns the full enumerated list of member codes, paged with offset and count.

\$validate-code

MEMBERSHIP — YES OR NO

Asks whether a Coding is a member of a ValueSet. One question, one boolean answer.

\$translate

MAP ACROSS SYSTEMS

Uses a ConceptMap to map a code from one system to another — say SNOMED to ICD-10.

And where do these run — tx-server topology

• PUBLIC

tx.fhir.org

Conformant and free. The default external terminology server.

• IN-PROCESS

HAPI tx-server

Upload your own CodeSystem, ValueSet, and ConceptMap resources.

• LICENSED

Snowstorm edition

How you legally serve real SNOMED at scale. Back to that license again.

• NOTE

ConceptMap

The input to \$translate only — it defines the cross-system mapping.

— TRY IT YOURSELF · 60 SECONDS

One curl, one jq — a live terminology operation.

Run \$expand on the observation-status ValueSet against the public HAPI sandbox, pipe to jq, and read expansion.total — that is how many codes are in the set.

— RUN THIS IN YOUR TERMINAL

```
curl 'https://hapi.fhir.org/baseR4/ValueSet/$expand?url=http://hl7.org/fhir/ValueSet/observation-status' | jq '.expansion.total'
```

One line, and you have run \$expand against a conformant tx server — no SDK, no setup.

Recap

● 5 code systems

● 4 binding strengths

● 3 tx operations

● 1 licensing trap — SNOMED

— 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.

— CHAPTER 06 · ACT II OPENS — FROM THE SPEC TO A RUNNING SERVER

For five chapters we learned a language. Now we run a server that speaks it.

Resources, datatypes, terminology, Bundles — that was the FHIR spec. From here the course turns. HAPI FHIR is the reference Java implementation, the one most production systems are built on. Three goals: the modules HAPI is made of, how it stores resources in a real database, and the pattern that keeps our own server upgradeable.

Where this chapter goes

- Modules — the Maven module stack
- Storage — Hibernate and the HFJ tables
- Overlay — the upgradeable deployment pattern

— DIAGRAM · WHAT HAPI IS, UNDERNEATH

HAPI is a stack of composable Maven modules.

Read it bottom-up. Each layer depends on the one below it, and an application caps the stack by assembling them into a server.

● LAYER 1 · CORE

hapi-fhir-base

Version-independent core — parsers, the client, the FHIR context.

● LAYER 2 · MODEL

structures-r4

The R4 resource model — the classes for Patient, Observation, Condition.

● LAYER 3 · STORAGE API

hapi-fhir-storage

Storage abstractions — the DAO and persistence interfaces.

● LAYER 4 · PERSISTENCE

jpaserver-base

The concrete JPA implementation that talks to a database via Hibernate.

— VALIDATION + ASSEMBLY CAP

FhirInstanceValidator · RestfulServer + IResourceProvider beans

A validation module supplies the instance validator. To become a running server you assemble these modules — a RestfulServer with your resource-provider beans on top. That is the whole shape.

— DIAGRAM · WHERE THE RESOURCE ACTUALLY LANDS

The transaction Bundle you posted becomes rows, through Hibernate.

— CURRENT STATE

HFJ_RESOURCE

Holds the current state of every resource — one row per live resource, the version you read on a plain GET.



Ava

Into a relational database, through Hibernate. The resource you POST becomes rows.

— EVERY VERSION

HFJ_RES_VER

Holds every historical version. The serialized body lives in the version row — that is what powers _history reads and versioned reads.



Marcus · the JPA server

Hibernate ORM maps the entities onto whatever engine you run — Postgres, H2, SQL Server, Oracle.

— DIAGRAM · HOW SEARCH AVOIDS SCANNING EVERY BODY

Search parameters are extracted at write time into typed index tables.

When you save a resource, HAPI writes typed index rows. A REST search compiles straight to SQL against those tables — never a scan of the serialized bodies.

• STRINGS

HFJ_SPIDX_STRING

name, address, free-text params.

• TOKENS

HFJ_SPIDX_TOKEN

coded values — system plus code.

• DATE · QUANTITY · URI

HFJ_SPIDX_DATE / _QUANTITY / _URI

typed columns for range and unit search.

• REFERENCES

HFJ_RES_LINK

resolved links between resources for chaining.

— FULL-TEXT IS A SEPARATE ENGINE

Hibernate Search — Lucene (single-node) or Elasticsearch (scale-out)

Finding a word inside a note — `_content`, `_text`, `:contains` — is powered by Hibernate Search. Lucene on disk is the starter default; Elasticsearch is the production scale-out option. We push search hard in Chapter 09.

— CONCEPT · NOBODY WIRES THOSE MODULES BY HAND

You start from hapi-fhir-jpaserver-starter.

It is a runnable Spring Boot application that wraps jpaserver-base, configured almost entirely through one application.yaml file — the hapi.fhir.* keys. We point at those keys here; Chapter 07 walks them.

— CONFIGURED BY APPLICATION.YAML

hapi.fhir.* — fhir_version · storage · partitioning · batch jobs · validation · CORS

You customize by adding your own application class, interceptors, and datasource beans on top. The keys exist now; turning the knobs is the next chapter.

2026 baseline

● HAPI 8.x

● Spring Boot 3

● Java 17+

● Jakarta namespace

— CONCEPT · FRAMEWORK ANCHOR — THE PATTERN WE USE

Do not edit the starter in place. Vendor it as an overlay.

— 1 · PULL UPSTREAM MERGEABLE

git subtree

We bring the upstream starter in with git subtree, so we can pull new HAPI releases as clean three-way merges.

— 2 · OUR CODE, OUR PACKAGE

com.dudoxx.fhir.r4 — package convention

All of our own code lives in an overlay source tree, under our own package convention — never mixed into vendored files.

— 3 · BRACKET UNAVOIDABLE EDITS

// DDX-OVERLAY-BEGIN ... // DDX-OVERLAY-END

When we must touch a vendored upstream file, we bracket the change with two comment markers. A future subtree pull merges cleanly, and every local change is one grep away.



Ava → Marcus

That is the trap. The overlay also re-enables Clinical Reasoning, which the stock starter ships disabled.

— COMPARISON · WHY NOT JUST FORK IT?

Upstream-mergeable beats forked-and-frozen.

Many teams fork — and then they freeze. The overlay flips that trade. Here is the difference, side by side.

— HARD FORK — THE TRAP

You own the entire tree

Every upstream release becomes a manual reconciliation. The drift compounds, merges get painful, and eventually you stop upgrading at all.

— VENDORED OVERLAY — THE PATTERN

git subtree pull stays mergeable

Upstream three-way-merges against your subtree. Your edits stay isolated and auditable behind the markers, and you ride HAPI 8.x point releases instead of fearing them.

● **Fork — drift compounds, upgrades stall**

● **Overlay — clean merges, edits a grep away**

● **The takeaway for a server that moves as fast as HAPI**

— WALKTHROUGH · ONE REQUEST, END TO END

Every slide so far is one stop on this path.

A GET or a POST enters at the top and exits as serialized FHIR. The modules dispatch, the schema persists — this is the spine.

● STEP 1 · DISPATCH

RestfulServer

The dispatcher receives the HTTP request.

● STEP 2 · ROUTE

IResourceProvider

Routes to the provider for that resource type.

● STEP 3 · PERSIST

DAO → Hibernate

The operation becomes SQL against the schema.

● STEP 4 · LAND

HFJ_* tables

Rows written; FHIR serialized back out.

— ON THE WAY THROUGH

Interceptor Pointcuts fire — auth · partitioning · audit

Pointcuts are the hook surface where you inject cross-cutting behavior between dispatch and the DAO. We write our own in Chapter 10.

— DIAGRAM · THE WHOLE PICTURE, ON ONE SCREEN

None of it is magic once you see the layers.

A client sends FHIR over HTTP. Every box here is a slide we just did — assembled into a production FHIR server.

● CLIENT

FHIR over HTTP

REST request from any conformant client.

● SLIDES 05 · 06

Spring Boot starter + DDX overlay

jpaserver-starter with our overlay on top.

● SLIDES 02 · 08

HAPI module stack

RestfulServer dispatches; modules handle it.

● SLIDES 03 · 04

Hibernate → HFJ_*

Mapped down and persisted.

— PRIMARY STORE · SLIDE 03

Postgres — HFJ_RESOURCE + HFJ_RES_VER

Current state plus full version history, mapped by Hibernate ORM.

— FULL-TEXT SIDECAR · SLIDE 04

Lucene index

Hibernate Search alongside the relational store for _content and _text.

— TRY IT YOURSELF · UNDER A MINUTE

Clone the starter, boot it, and read its own software block.

In under a minute you have a live FHIR server on localhost. Then GET /fhir/metadata and pipe to jq — the running server names its own HAPI version in the CapabilityStatement software block.

— RUN THIS IN YOUR TERMINAL

```
git clone https://github.com/hapifhir/hapi-fhir-jpaserver-starter
cd hapi-fhir-jpaserver-starter && mvn spring-boot:run -Pboot
curl http://localhost:8080/fhir/metadata | jq '.software'
```

Recap

● 4 module layers

● HFJ_* schema + Hibernate

● Lucene full-text

● The overlay pattern

— CHAPTER 07 · SERVER / STACK · WE BUILT THE OVERLAY — NOW WE CONFIGURE IT

One yaml key can decide whether the server even boots.

Configuration is where HAPI is most powerful and most dangerous. In Chapter 06 we vendored the overlay and saw the module layout. Today we open the control panel — and learn to read it without breaking it.

What you will be able to do

- Read the hapi.fhir config tree with confidence
- Tune storage_settings without losing data
- Recognise three 8.x pitfalls before they cost a debugging afternoon

— CONCEPT · APPLICATION.YAML — ONE TREE, EVERY KNOB

The starter binds all of its config under one root: **hapi.fhir**.

Navigate this tree and you can configure HAPI. Almost every switch you will ever touch hangs off the same single root — including the storage settings we tune next.

— HAPI.FHIR.* — THE CONFIG ROOT

hapi.fhir.server_address · server.path · cors.* · mdm_enabled · cr.enabled · cdshooks.enabled

Server base URL and path, plus the feature flags that switch whole subsystems on and off — master data management, clinical reasoning, and C D S hooks.

● BRANCH · 1

Server

server_address, path, CORS — how clients reach the FHIR endpoint.

● BRANCH · 2

Feature flags

mdm_enabled, cr.enabled, cdshooks.enabled — whole subsystems on or off.

● BRANCH · 3

Binary + hooks

binary_storage_enabled and _mode, plus custom-interceptor-classes and custom-provider-classes.

● BRANCH · 4

storage_settings

Nested in the same tree — the day-one data-behavior knobs, next slide.

— CONCEPT · STORAGE_SETTINGS — FOUR SETTERS THAT MATTER DAY ONE

Each of these changes data-loss behavior. Change them deliberately.

These nest under the same tree and map to JpaStorageSettings setters. Every one alters whether data survives — flip them in version control, never live by hand.

• SETTER · 1

allow_multiple_delete

Off by default — HAPI refuses to delete more than one resource per operation. A safety rail; flip it only when you mean it.

• SETTER · 2

expunge_enabled

Hard, permanent removal of history and deleted rows. Ships off; enabling brings an expunge batch size that bounds each chunk.

• SETTER · 3

allow_external_reference

Lets a resource point at a URL on another server. Pairs with `treat_base_urls_as_local` — which bases HAPI treats as its own.

• SETTER · 4

enforce_referential_integrity

On write and on delete — decides whether HAPI allows a dangling reference, or a delete of a still-referenced resource.

• **CascadingDeleteInterceptor + `_cascade=delete` walks references**

• **Pair with `_expunge=true` to truly erase**

— PITFALL 1 OF 3 · THE 8.X SURPRISE — A SERVER THAT WON'T START

Enable the email subscription channel — and the server may not boot.

— PATTERN THEN SYMPTOM

PATTERN — enabling the email channel makes autoconfig wire a mail-sender component.

SYMPTOM — with no S M T P configuration, the application context fails to build and the server dies on boot. No FHIR endpoint, just a stack trace about a missing mail-sender bean.



Ava

Exactly. One feature flag, and there is no endpoint to debug against.



Marcus · the fix

Supply the mail settings — host, port, from-address — or do not enable the email channel until you need it.

— PITFALL 2 OF 3 · THE SUBTLE ONE — A SILENT SEMANTIC FLIP

Same yaml, different behavior after a minor bump. No error, no warning.

— PATTERN THEN SYMPTOM

PATTERN — a boolean in the partitioning config changed its meaning across 8.x minor versions.

SYMPTOM — the same unchanged file produces different behavior after a minor version bump. The server now partitions, or stops partitioning, differently than you expected — the kind of bug you only find in production, weeks later.



Ava

Right — which is exactly why the rule from slide one matters most here.



Marcus · the fix

Re-read the flag against your pinned release's own application.yaml. Treat anything you read elsewhere as a pointer to check.

— PITFALL 3 OF 3 · UPSTREAM OF HAPI — THIS ONE IS SPRING

You edit a config that was never being read.

The starter ships named configuration profiles. Which one is active is decided by a precedence between an environment variable and what the yaml declares.

● PATTERN

Profile activation

Named profiles, activated by a precedence between an environment variable and the yaml-declared default.

● SYMPTOM

Wrong profile, silently

A stale env var or a yaml default quietly activates a different profile — wrong datasource, wrong flags, all applied without a word.

● FIX

Be explicit + log it

Be singular about where activation comes from, and log the active profile on startup so you can see it.

● COST

An hour, gone

You spend an hour editing config the server never reads — then doubt the key, not the profile.

● **UNVERIFIED** — the exact shipped profile names and the precise activation precedence. Confirm both against your pinned tag before scripting the demo.

— DIAGRAM · WHEN AN OPERATION IS TOO BIG FOR ONE REQUEST

The heavy, long-lived operations run on the Batch2 framework.

Built on Spring Batch — chunked, persisted to the database, asynchronous, and resumable. A reindex over millions of resources survives a restart and picks up where it left off, instead of blocking a request thread and timing out.

● BATCH2 · JOB 1

Reindex

Rebuilds search indexes after you add or change a search parameter. Surfaced via \$reindex.

● BATCH2 · JOB 2

Bulk export

The \$export operation — streams large datasets out as N-D-J-S-O-N.

● BATCH2 · JOB 3

Delete-expunge

Batched permanent removal from slide three — using the same expunge batch size.

● Chunked

● DB-persisted

● Asynchronous

● Resumable after restart

— COMPARISON · TWO OPT-IN TOGGLES — SAFETY AND PRESENTATION VS COST

Both are off by default, because both cost you something.

You turn each on where its benefit outweighs its price. Validation is the doorway to Chapter 08 — today, just know the toggle exists and what it costs.

— TOGGLE 1 — VALIDATION

RequestValidatingInterceptor · validation.requests_enabled

HAPI can validate every incoming request against the spec and your profiles — and responses too. Opt-in because validation is expensive per call. Turn it on where correctness matters more than throughput.

— TOGGLE 2 — NARRATIVE

narrative_enabled · DefaultThymeleafNarrativeGenerator

Auto-generates the human-readable narrative block inside resources. It costs C P U and adds to startup. If your clients render their own display, switch it off and reclaim that.

● Next · Chapter 08 goes deep — profiling, StructureDefinitions, the instance validator

— CONCEPT · HAPI.FHIR.PARTITIONING.* — TURNING IT ON

Enabling is one switch. The semantics are version-sensitive.

There is a partitioning section in the tree where you enable it, declare your allowed partitions, and govern whether duplicate identifiers are allowed across a conditional create.

— HAPI.FHIR.PARTITIONING.*

enabled · allowed_partitions · conditional_create_duplicate_identifiers_enabled

Enabling partitioning is genuinely one flag. Understanding exactly what each related boolean does in your release is the hard part — and getting it wrong is how tenants leak into each other.



Ava

One switch — but, callback to slide five, the switch is easy and the semantics are version-sensitive.



Marcus · scope for today

Exactly. The production strategy — tenant registry, two-database wiring — is all Chapter 11.

— WALKTHROUGH · ONE SCREENFUL THAT BOOTS CLEAN AND DOES NOTHING DANGEROUS

The safe shape — not a promise of the exact key.

— APPLICATION.YAML — MINIMAL

```
spring.datasource.url: jdbc db url
hapi.fhir.server_address: base url
hapi.fhir.storage_settings.allow_multiple_delete:
false
hapi.fhir.narrative_enabled: false ^verify
hapi.fhir.partitioning.enabled: false ^verify
subscription_email_channel: off ^verify
```

• CALLBACK · SLIDE 02

The tree

Server block and base path — every line hangs off hapi.fhir.

• CALLBACK · SLIDE 03

One storage setter

Left at its safe default, stated explicitly so we own it.

• CALLBACK · SLIDES 04 TO 06

Three things off

Email channel, partitioning, narrative — the three pitfalls, deliberately disabled so it boots.

• Any line marked ^verify is one to confirm against your pinned tag — I am showing the safe shape, not asserting the exact key.

— TRY IT YOURSELF · SEE THE SYMPTOM BEFORE PRODUCTION DOES

Boot it clean. Then break it on purpose.

— THE EXERCISE — THREE STEPS

- 1 · Boot your pinned HAPI image with the minimal yaml from slide 10 — confirm it comes up clean.
- 2 · Flip `narrative_enabled` and watch the startup logs change.
- 3 · Enable the email subscription channel with no S M T P settings — reproduce the slide-04 boot failure with your own eyes.

Recap — and the discipline

● One config tree

● Four storage setters

● Three Batch2 jobs

● Pin the tag · verify the email path, the partitioning boolean, the profile precedence

● Next · Chapter 08 · Validation becomes a discipline — profiling, StructureDefinitions, slicing, the instance validator

— CHAPTER 08 · PROFILING & VALIDATION · BEAT 1 — THE CONTRACT

The server is standing. Now — how do we say what counts as valid?

Resources flow in and out, but nothing yet enforces that in OUR system a Patient must carry an identifier, or an Observation must use a LOINC code. That is profiling. A profile constrains a base resource into a contract — and the validator is where the contract bites.

Three goals for this chapter

- **Constrain** — a base resource with a `StructureDefinition`
- **Slice + must-support** — split repeats, declare capability
- **Validate** — a real instance against a profile in HAPI

— CONCEPT · A PROFILE IS A CONSTRAINED BASE RESOURCE

A StructureDefinition with derivation constraint narrows a base.

Take Patient and tighten it. Three legal moves on one element — narrow cardinality within the base limits, fix a value, and re-bind terminology. Remember the four binding strengths from chapter four — you can only tighten, never weaken. A base required binding stays required.

— ELEMENTDEFINITION · THREE MOVES

Patient.identifier

"min": 1, "max": "1" — was 0..1, now required
"fixed[x]" / "pattern[x]" — element must equal exactly this

"binding": { "strength": "required" } — re-bound, tighter only

— THE HARD LIMIT

Constrain — never extend

A profile can narrow what already exists. It cannot rename a base element or add a brand-new one.

New elements need extensions — a different mechanism we met earlier. Profiling is subtraction, not addition.

● CAN narrow cardinality within base limits

● CAN fix or re-bind — strength only stronger

● CANNOT rename or add new elements

— DIAGRAM · SLICING A REPEATING ELEMENT

Slicing splits a repeat into named, mutually-exclusive lists.

Observation.component — sliced

discriminator: type value · path code



slice: diastolic

component.code = 8462-4

☒ open — extras allowed

☐ closed — named only

☐ openAtEnd — extras after



Marcus & Ava

• discriminator

How does the validator tell them apart?

— CONCEPT · THE DISTINCTION THAT TRIPS EVERYONE

Must-support is capability. Cardinality is presence.

The single biggest misconception in profiling — `mustSupport` true does not mean `must-populate`. They are two different mechanisms. Confuse them and you build the wrong validation, or fail a certification for the wrong reason.

— MUSTSUPPORT — CAPABLE OF HANDLING

Must-SUPPORT

The implementing system must be CAPABLE of handling the element — storing it, displaying it, processing it.

It says nothing about whether any given instance contains it. And a profile declaring `must-support` must also state what kind of support it expects.

● `mustSupport` — implementer capability

— CARDINALITY — MUST BE PRESENT

Must-POPULATE

Cardinality forces presence. `1..1` means the element must actually appear in the instance. This is the trap — people read `mustSupport` and assume required. Presence is governed by `min`, not by `support`.

● `cardinality 1..1` — instance must contain it

— WALKTHROUGH · PROFILES YOU CAN ADOPT — US CORE

US Core profiles the core US clinical data set on FHIR R4.

In the US you rarely write from scratch — you adopt US Core. Three foundational profiles to know, each declaring the must-support elements a conformant system has to handle.

• DEMOGRAPHICS

US Core Patient

Demographics and identifiers, with must-support elements a conformant system must handle.

● **must-support**

• A FAMILY OF PROFILES

US Core Observation

Not one profile — a family: vital signs, lab results, and screening and assessment.

● **must-support**

• SPLIT IN TWO

US Core Condition

condition-encounter-diagnosis and condition-problems-health-concerns.

● **must-support**

● **Current — US Core 8.0.1 · STU 8 · 2026 (curriculum index written at v7; numbering stable, content current)**

— CONCEPT · CROSSING BORDERS — THE INTERNATIONAL PATIENT SUMMARY

IPS is a portable summary built from a Composition.

US Core is American. When care moves between countries, the International Patient Summary — IPS — is the cross-border answer. Think of it as a portable document pulling together the themes that matter when a clinician has never seen you before.

— THE RECURRING SECTION THEMES

Allergies · Medications · Problems

A Composition-driven document that gathers a patient's key clinical context into a single, shareable summary.

We keep IPS at the concept level — know that it exists, and why it matters when care crosses a border.

— WHY THIS MATTERS IN THE EU

The cross-border payload format

IPS is the payload format behind Europe's cross-border health-data exchange — and the regulatory clock is real, with a phased deadline through the end of this decade.

● **Portable patient summary**

● **Composition-driven document**

● **Cross-border / EU motivation**

— WALKTHROUGH · HOW PROFILES REACH THE SERVER

A profile is just data until you load the package that carries it.

Profiles like US Core ship as an Implementation Guide package — NPM-style. Load it into the server we stood up, and the validator can finally reach the profiles.

— IG PACKAGE ANATOMY — PACKAGE.TGZ

package.json + conformance resources

StructureDefinitions · ValueSets · CodeSystems · examples

One tarball bundles everything the validator needs to enforce the guide.

— LOAD IT INTO HAPI

hapi-fhir-cli upload-definitions

```
./hapi-fhir-cli upload-definitions -t  
http://localhost:8080/fhir -d ./us.core.tgz
```

Or pre-load the package at startup. Either way, the profiles now live in HAPI's validation and terminology store.

● **package.tgz — conformance bundle**

● **CLI upload or startup pre-load**

● **Profiles now reachable by the validator**

— FRAMEWORK ANCHOR · HOW HAPI CHECKS A RESOURCE

FhirInstanceValidator walks an instance against its profile.

You build a FhirValidator, give it the FhirInstanceValidator module, and back it with an IValidationSupport chain. Input — an instance plus the profile from meta.profile or an explicit URL. Output — an OperationOutcome, one issue per violation.

The validation chain

instance + profile to OperationOutcome

vs

IValidationSupport chain

in-memory + uploaded IG packages + terminology

oo

OperationOutcome

one issue per violation, each with a severity

— WHAT EACH ISSUE CAN FLAG

The contract, enforced in code

Cardinality too low. A code outside a required binding. A slice that does not match. A missing must-support capability.

Each becomes an issue with a severity. That OperationOutcome IS your profile being enforced — not documentation, executable contract.

● cardinality

● binding

● slicing

● must-support

— CONCEPT · A CUSTOM ORG PROFILE — DERM4HAIR

Where the contract bites — at write time.

Derm4Hair Patient — three constraints

derivation: constraint · base: Patient

- Enforced via FhirInstanceValidator in the create path



Ava & Marcus

● write time

So where does this profile actually bite?

— CODE · SEE THE ACTUAL ARTIFACT

The differential is just your changes against the base.

Here is the StructureDefinition itself — compact, reviewable, and exactly what you ship inside the IG package.

— STRUCTUREDEFINITION · DERM4HAIR PATIENT

"derivation": "constraint" — a profile, not a new resource

"baseDefinition": "http://hl7.org/fhir/StructureDefinition/Patient"

"differential": { "element": [

{ "path": "Patient.identifier", "min": 1 } — now required

{ "path": "Patient.identifier.system", "fixedUri": "https://derm4hair.example/mrn" } — fixed system

{ "path": "Patient.communication.language", "mustSupport": true } — capability declared

] }

● **derivation: constraint**

● **baseDefinition is Patient**

● **min narrowed to 1**

● **fixed[x] system**

● **mustSupport true**

— CODE · TRY IT YOURSELF · 60 SECONDS

Validate a Patient against US Core — one call.

Post a Patient to the dollar-validate operation with a profile parameter pointing at US Core Patient, then read the issues. Severity and diagnostics tell you exactly what failed.

— RUN THIS AGAINST THE PUBLIC HAPI SANDBOX

```
curl -s 'https://hapi.fhir.org/baseR4/Patient/$validate?
profile=http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient' -H 'Content-Type:
application/fhir+json' -d @patient.json | jq '.issue[] | {severity, diagnostics}'
```

One call, and you have validated against a real published profile — no SDK, no setup.

Recap

● Constrain a base resource

● Slice a repeating element

● Declare must-support

● Package profiles into an IG

● Enforce with the validator

— CHAPTER 09 · SEARCH OPERATIONS · BEAT 1 — NOW WE ASK

Every chapter built resources. This one asks them questions.

FHIR search is a query language that rides on plain URL parameters, and it goes far deeper than most people ever use. We compose queries across the parameter types and modifiers, traverse references with chaining and includes, and — the lesson that bites — keep all that power from blowing up your server.

What you will be able to do

- Params & modifiers — types, prefixes, refinements
- Chains & includes — query the reference graph
- The blowup — bound _include and \$everything

— COMPARISON · EVERY PARAMETER HAS A TYPE — AND THE TYPE DECIDES THE MATCH

Nine types, three families. Get the type right, search falls into place.

Get it wrong and your query silently matches nothing. Group them: a value-shape family, an ordered family, and composite — the combiner.

— VALUE-SHAPE FAMILY

string · token · reference · uri

string — case- and accent-insensitive prefix match by default; "smi" matches "Smith". token — a coded value as system|code, the Coding shape. reference — a literal reference, the links we built in Ch 03. uri — an absolute address.

● string = prefix match, no case

● token = system|code

— ORDERED FAMILY

date · number · quantity

Values with a natural ordering — so they accept prefixes like greater-than and less-than. quantity also carries a unit. composite combines two values in one expression — for example a component code paired with its value.

● ordered types take prefixes

— COMPARISON · APPEND A COLON AND A KEYWORD TO REFINE THE MATCH

Most missed-result bugs are a forgotten — or wrong — modifier.

Land the universal one first: `:missing` works on every type. After that, modifiers behave differently per type. Reach for one deliberately — the default is almost always a prefix or exact-code match.

`:missing`

UNIVERSAL · ALL TYPES

`gender:missing=true` finds resources where gender was never set. true or false.

● `:missing` — the one that works everywhere

`string`

`:EXACT` · `:CONTAINS` · `:TEXT`

`:exact` turns off the prefix match.
`:contains` does a substring. `:text` searches the display text.

● `:below` — subsumption, the Ch 04 hierarchies

`token`

`:NOT` · `:ABOVE` · `:BELOW`

`:not` excludes a code. `:above` and `:below` walk a terminology hierarchy — `:below` pulls in the more-specific children.

● Wrong modifier = silent zero results

— DIAGRAM · DATES AND NUMBERS NEED MORE THAN EQUALS

Nine prefixes — and they apply to the ordered types only.

One rule to remember: prefixes never apply to string, token, or uri — only date, number, and quantity. eq is the default.

— WORKED EXAMPLE — BORN ON OR AFTER

GET Patient?birthdate=ge2010-01-01

ge is greater-than-or-equal, so this returns patients born on or after the first of January, twenty-ten.

— THE EVERYDAY SIX

eq · ne · gt · lt · ge · le

eq equals (the default), ne not-equals, gt and lt for greater-than and less-than, ge and le for the or-equal versions.

— THE THREE PEOPLE FORGET

sa · eb · ap

sa starts-after and eb ends-before — both for ranges. ap is approximately, a fuzzy window around the value.

— WALKTHROUGH · CROSS A REFERENCE INSIDE A SINGLE QUERY

The dot hops from the Observation into the Patient it points at.

— FORWARD CHAIN — ONE REQUEST, NO ROUND TRIPS

GET Observation?subject:Patient.name=Smith

Observations whose subject resolves to a Patient whose name is Smith. subject is the reference, :Patient types it because subject can point at several resource kinds, and .name reaches into that patient.



Ava

That is the whole reference graph, queryable from a URL.



Marcus · mechanism

Don't fetch the patients first — chain straight through. One request.

— WALKTHROUGH · THE INVERSE OF CHAINING — WHO POINTS BACK AT ME

Forward chain follows a reference out. `_has` finds who points at you.

— REVERSE CHAIN — READ IT BACKWARDS

GET Patient?_has:Observation:patient:code=1234-5

Give me patients that are referenced — through the patient field — by some Observation whose code is one-two-three-four-dash-five. The arrow points backward, from the Observation to the Patient.



Ava

Reach for `_has` when the question is "which parents have a matching child".



Marcus · the subtlety

One trap — stack separate criteria and each chain evaluates independently. They need not be satisfied by the same target resource.

— CONCEPT · THE BLOWUP, PART 1 — WHEN ONE QUERY DRAGS IN THOUSANDS

_include pulls the referenced. _revinclude pulls the referencing.

— **_INCLUDE** — REFERENCED RIDE ALONG

Observation?

_include=Observation:subject

The referenced patients arrive in the same bundle. **_revinclude** is the mirror — pull resources that reference your matches, like Provenance. Included entries carry `search.mode=include`, not `match`, so you can tell what you asked for from what came along.

— **THE BLOWUP** — **_INCLUDE=* / :ITERATE**

_include=*&_include:iterate=*

Wildcard every reference, or `:iterate` to recurse, and one innocent query drags in thousands of resources — a bundle that hammers the server and the client. The fix is discipline: name the exact includes you need, and always cap with `_count`.

12

BOUNDED · NAMED INCLUDE

One explicit `_include` plus `_count=10` — a small, predictable bundle.

8,400

UNBOUNDED · `_INCLUDE=* :ITERATE`

Wildcard recursion sweeps the reference graph — the server pays for every hop.

_count

THE BRAKE · ALWAYS PRESENT

Power, bounded. Name the includes, cap the page, and the blowup never happens.

— CONCEPT · THE BLOWUP, PART 2 — THE WHOLE COMPARTMENT IN ONE CALL

\$everything is wonderful for an export and brutal as a casual query.

It returns the entire patient compartment — every resource that references the patient. That is a whole-compartment sweep, not a targeted search. On a patient with years of history it is a very large response.

— WHOLE-COMPARTMENT OPERATION

GET Patient/123/\$everything?_count=50

HAPI bounds it with _count and paging so it does not fall over. Same lesson as _include: convenience plus a count limit equals safe. Convenience alone equals an incident.

1

CALL

\$everything is a single convenient request.

ALL

COMPARTMENT RESOURCES RETURNED

Every Observation, Encounter, Condition, MedicationRequest that references the patient.

_count

PAGING · THE ONLY SAFE WAY

Bound the page size and walk the next-links — never pull it unbounded in a live query.

— WALKTHROUGH · WHEN THE BUILT-IN PARAMETERS DON'T COVER YOUR FIELD

Define your own SearchParameter — then reindex, or get zero results.

— SEARCHPARAMETER RESOURCE — FOUR THINGS MATTER

```
{ "resourceType": "SearchParameter", "code": "ejection-fraction", "base":  
["Observation"], "type": "number", "expression": "Observation.valueQuantity.value" }
```

code — the name you query with. base — which resource types it applies to. type — one of the nine. expression — a FHIRPath that points at the element to index.



Ava

New writes, yes — they get indexed as they come in.



Marcus · production scar

I shipped a custom param once, got zero results, and panicked. The rows were there — just not indexed for the new path. New param ... then reindex ... then it works.

— DIAGRAM · WHY A NEW PARAMETER NEEDS A WHOLE REINDEX

Index at write. Query the index — never the stored JSON.

On the JPA server we toured in Ch 06, search never reads your stored document at query time. The mental model is what matters; the exact table layout is a version-specific implementation detail, so don't memorize it.

Write path — HAPI extracts, then indexes

One indexed entry per searchable parameter value



3

Index entries written — organized by type

Strings in one place, tokens in another, dates in another. References resolve through a link relationship between resources.



4

Search hits the index

A query reads those compact indexes, not the full document — that is why search is fast. A new parameter means new index entries, which is exactly what reindex builds.

— TRY IT YOURSELF · SEE MATCH AND INCLUDE SIDE BY SIDE

One chained query, one include — read back the search modes.

On the public HAPI sandbox, run a chained query with an include and ask for just the search modes. You will see match and include side by side, exactly as Marcus described.

— RUN THIS IN YOUR TERMINAL

```
curl 'https://hapi.fhir.org/baseR4/Observation?
subject:Patient.name=Smith&_include=Observation:subject&_count=5' | jq
'.entry[].search.mode'
```

The matched Observations come back as match; the included Patients come back as include — the distinction, live.

Recap

● 9 parameter types

● 7 modifiers

● 9 ordered prefixes

● chain + _has

● _include + the blowup

— CHAPTER 10 · CUSTOM INTERCEPTORS + OPERATIONS

So far the server obeyed FHIR. Now you teach it your own rules.

Interceptors reach into the request lifecycle so you can act at the right moment. Operations add verbs beyond create, read, update, and delete. Write a concern once — auth, tenant, redaction — instead of copy-pasting it into every provider.

Three things you will be able to do

- The interceptor model — `@Interceptor` + `@Hook`
- The Pointcut request lifecycle
- Custom operations — your own \$verbs

— CONCEPT · THE INTERCEPTOR IS JUST A PLAIN CLASS

An interceptor is a POJO. Two annotations, and HAPI wires it in.

No base class to extend. You mark the class with `@Interceptor`, and you mark each reacting method with `@Hook`, passing the Pointcut you care about. The framework reads your method signature and injects the right context object by type — declare only what you need.

— CANONICAL EXAMPLE · TAGTRIMMINGINTERCEPTOR

`@Interceptor`

```
public class TagTrimmingInterceptor {  
    @Hook(Pointcut.STORAGE_PRESTORAGE_RESOURCE_CREATED)  
    public void trim(IBaseResource resource) { /* clear tags + security labels before storage */ }  
}
```

The framework injects context by parameter type

● RequestDetails — the parsed inbound request

● IBaseResource — the resource being saved

● Ask for it, and HAPI hands it to you

— DIAGRAM · A POINTCUT IS A NAMED MOMENT IN A REQUEST'S LIFE

One request can be touched at three different moments.

Two families. SERVER pointcuts fire on every HAPI server; the one you reach for most is SERVER_INCOMING_REQUEST_PRE_HANDLED — parsed, but before any provider runs. STORAGE pointcuts wrap persistence: PRESTORAGE just before a write, PRESHOW just before resources go back out.

One request, five touch-points

SERVER_* fire on every server · STORAGE_* auto on JPA, fired manually on a plain server



3

Provider method runs@Read / @Search / @Create handler



4

STORAGE_PRESTORAGE_RESOURCE_CREATEDmutate or validate just before write — UPDATED variant too



5

STORAGE_PRESHOW_RESOURCES

— WALKTHROUGH · HOW THE SERVER LEARNS ABOUT YOUR INTERCEPTOR

You register it. And the order you register in matters.

On a plain server you call `registerInterceptor` on the interceptor service. On the JPA Boot overlay we toured in Ch 06 — wired through the `application.yaml` from Ch 07 — you declare it as a Spring bean and it is picked up automatically.

— PLAIN SERVER

`server.getInterceptorService()`

```
.registerInterceptor(new  
ApiTokenAuthInterceptor());
```

Add the instance to the server's interceptor registry yourself.

— JPA BOOT OVERLAY

@Bean

```
ApiTokenAuthInterceptor apiTokenAuth() {  
    return new ApiTokenAuthInterceptor();  
}
```

Declare it as a Spring bean — the overlay's configurer registers it.

Why order matters

- Two interceptors mutate the same resource — order decides who runs first
- Put auth ahead of everything — never run a business rule before the gate

— FRAMEWORK ANCHOR · THE AUTH SEAM — A PATTERN, NOT A FILE PATH

An API-token auth interceptor says no at the edge, before any provider.

— PATTERN · APITOKENAUTHINTERCEPTOR (CONCEPTUAL)

@Hook(Pointcut.SERVER_INCOMING_REQUEST_PRE_HANDLED)

Read the bearer token off the inbound RequestDetails. If it is missing or invalid, reject the call right there — the edge is the cheapest place to say no.



Marcus · the mechanism

If the token is bad, it rejects before anything. No read handler, no search, no storage hook ever sees an unauthenticated request.



Ava · the payoff

That is the cross-cutting move — one concern, one place, not a check pasted into fifty providers. The exact class is an implementation detail; the seam and behavior are what matter.

— FRAMEWORK ANCHOR · THE TENANT SEAM — SAME POINTCUT, DIFFERENT JOB

A clinic-partition interceptor binds the request to a tenant — at the same seam.

— PATTERN · CLINICPARTITIONINTERCEPTOR (CONCEPTUAL)

@Hook(Pointcut.SERVER_INCOMING_REQUEST_PRE_HANDLED)

Right after auth, read the tenant signal off the request — an X-Clinic-ID header — and assign the request to the matching HAPI partition. Once bound, storage and the search we just covered in Ch 09 are scoped to that tenant. One clinic can never see another's data.



Ava · what keeps tenants apart

Yes — once the request is bound to a partition, storage and search are automatically scoped to that tenant.



Marcus · the Ch 11 seed

That is exactly Ch 11 — the tenant registry, the slug-to-partition mapping, the two-database wiring. Here, just hold onto WHERE it happens: the assignment lands at the request seam, in an interceptor.

— CONCEPT · FROM INTERCEPTING REQUESTS TO DEFINING NEW HANDLING

IResourceProvider is the home for everything specific to one resource type.

Interceptors touch requests on the way through. To add genuinely new handling for a resource, you implement IResourceProvider: declare getResourceType — say, Patient — and carry annotated handler methods, one per operation you support.

— PATIENTRESOURCEPROVIDER — ONE RESOURCE TYPE

```
public class PatientResourceProvider implements IResourceProvider {  
    public Class<Patient> getResourceType() { return Patient.class; }  
    @Read public Patient read(@IdParam IdType id) { ... } }
```

Where handlers and operations live

• HANDLER

**@Read · @Search ·
@Create**

Annotated methods, one per operation the provider supports.

• SCOPE · TYPE / INSTANCE

**on the resource
provider**

Type-level and instance-level custom operations live here.

• SCOPE · SERVER

on a plain provider

Server-wide operations sit on a plain provider, not a resource one.

• THE BRIDGE

intercept → define

From shaping requests to adding new verbs — which is next.

— WALKTHROUGH · THE DOLLAR-PREFIXED VERBS BEYOND CRUD

@Operation defines a new verb. You already met one — Patient/\$everything.

FHIR is not only create, read, update, delete — it has operations. To write your own, annotate a method with @Operation naming it \$custom-op, and bind each input with @OperationParam. Add @IdParam to scope it to an instance.

— A CUSTOM INSTANCE-SCOPED OPERATION

@Operation(name="\$custom-op", idempotent=true)

```
public Parameters customOp(@IdParam IdType id, @OperationParam(name="since") DateType
since) { ... }
```

• DEFINE

@Operation(name="\$custom-op")

Names the verb; @OperationParam binds each input.

• SCOPE

@IdParam

Targets a specific instance — the \$everything shape from Ch 09.

• HTTP VERB

idempotent=true → GET

Primitive params only; otherwise it is a POST.

• DISCOVERABILITY · CONVENTIONAL

OperationDefinition

Advertise in the CapabilityStatement so the verb is not invisible — the recommended convention.

— CODE · READ ONE CLASS TOP TO BOTTOM — CONCEPTUAL SKELETON

Annotation on the class. Annotation on the method. The resource handed to you.

— TEACHING SKELETON · NOT A VERBATIM OVERLAY QUOTE

@Interceptor

```
public class StampingInterceptor {  
    @Hook(Pointcut.STORAGE_PRESTORAGE_RESOURCE_CREATED)  
    public void stamp(IBaseResource resource) {  
        resource.getMeta().addTag().setCode("ingested"); // before it lands in the database  
    }  
}
```



Marcus · the shape

Its single parameter is the resource itself, injected by type. The body stamps one tag, then it lands in the database.



Ava · reflecting it back

And the parameter says "give me the resource". The model, the lifecycle, registration — all of it collapses into these few lines.

— DIAGRAM · ONE REQUEST, SHAPED FOUR TIMES — THE WHOLE TOPIC ON ONE CANVAS

Four small concerns at the seams. The business logic stays clean.

The request arrives. At the pre-handled seam, two interceptors fire in order. The provider runs. On the way to storage and on the way back out, two more hooks fire. None of them lives inside the provider.

Cross-cutting concerns at the seams

auth gate → partition assign → prestorage mutate → preshow redact



3

Provider runs the handler does its work



4

Prestorage mutate · PRESTORAGE_CREATED mutate or validate the resource before it is written



5

Preshow redact · PRESHOW_RESOURCES redacts the very searchset from Ch 09 before it leaves the server

— TRY IT YOURSELF · FEEL A REAL DOLLAR-VERB LIVE

POST a Patient to \$validate on the public sandbox — no custom server needed.

— HAPI.FHIR.ORG · A BUILT-IN OPERATION

```
curl -X POST 'https://hapi.fhir.org/baseR4/Patient/$validate'
```

```
-H 'Content-Type: application/fhir+json' --data @patient.json | jq '.issue[].severity'
```

The server runs validation as an operation and hands back an OperationOutcome. Read the issue severities and you see the dollar-verb shape live.

What we covered

● **Interceptor model** · @Interceptor + @Hook

● **Pointcut lifecycle**

● **Registration & order**

● **Auth + partition seams**

● **IResourceProvider**

● **\$custom-op**

● **Next: Ch 11 — Multi-tenant partitioning** · the machinery behind WHERE the assignment happens

— CHAPTER 11 - MULTI-TENANT HAPI - COLD OPEN

Last chapter we sketched the interceptor. Now we wire it for production.

One HAPI server, many clinics. Multi-tenancy is the feature that turns a single FHIR server into a platform - and the one place where a single bug silently shows Clinic A another clinic's patients. So we treat isolation as the thing we must never get wrong.

Three goals for this chapter

- Choose a HAPI partitioning mode that isolates clinics
- Wire a tenant registry - clinic slug to partition id
- Recognise two cross-tenant footguns before they leak

— CONCEPT - THE PRODUCTION REALITY

One server. Dozens of clinics. One requirement that can never break.

We don't run one HAPI server per clinic. In Dudoxx HMS we run a single server, and every clinic is a tenant living inside it. That buys operational simplicity - and takes on the hardest correctness requirement in the system.

● FORCE - 1

One deployment

A single server, one ops surface, one backup, one upgrade. The whole reason to do this instead of N servers.

● FORCE - 2

N tenants

Each clinic is a tenant, onboarded without standing up new infrastructure.

● FORCE - 3

Isolation

The non-negotiable. Clinic A's queries must never return Clinic B's patients - under any circumstance.

● HAPI's native mechanism for this is partitioning

● Isolation is the one requirement we must never get wrong

— COMPARISON - HOW HAPI ASSIGNS A REQUEST TO A PARTITION

Two shapes. They differ only in where the tenant identity is read.

HAPI assigns every request to a partition, and the assignment is interceptor-driven - the same mechanism from Chapter 10. The interceptor is the seam; the mode just changes what it reads.

— BUILT-IN - PATH-TENANT

POST /fhir/P1/Patient

RequestTenantPartitionInterceptor paired with UriBaseTenantIdentificationStrategy. The tenant becomes a URL path segment - P1 is the tenant. Clean, built-in, no custom code.

— OUR PATH - HEADER-TENANT

X-Clinic-ID: northgate

Keep the partition interceptor, but read the tenant from a request header instead of the path. Our clinic identity already lives in a header the gateway sets - and we keep tenants out of every FHIR URL.

● We take the header path - gateway already sets X-Clinic-ID

● Either way, the interceptor is the seam

— DIAGRAM - PARTITIONSETTINGS - NAME TO ID

Partitioning is off by default. Once on, every row gets a partition id.

You enable partition-aware storage through PartitionSettings, before the server initialises. Once on, HAPI adds a PARTITION_ID integer column to every resource table and stamps each row with the partition it belongs to.

Partition identity - one partition, two views

A textual name (up to 200 chars) maps one-to-one to an integer id.

(name = id)

(down)



DEFAULT partition

where an unassigned resource lands - seed of footgun one

● **Verify - exact enable keys live in the Chapter 7 config dossier - pin your tag, we don't quote yaml here**

— COMPARISON - THE READ POINTCUT DECIDES EVERY TIME

The whole isolation guarantee rests on one return value.

Reads go through STORAGE_PARTITION_IDENTIFY_READ - it returns the partition or partitions a search is allowed to touch. Return one, return many: same hook, opposite outcomes.

— RETURNS ONE NAME - PARTITION-SCOPED

The clinic sees only its own data

The caller's partition, and nothing else. This is the true tenant isolation we promised - what a clinic user must always get.

— RETURNS MANY - CROSS-PARTITION

HAPI searches across all of them

A catastrophic leak for a clinic user - but the legitimate feature for an internal admin or a reporting job that spans tenants. Both, depending on who's asking.

● Isolation = READ hook returns exactly the caller's partition, nothing more

● Verify - cross-partition reference behaviour is version-sensitive

— WALKTHROUGH - CALLBACK TO CHAPTER 10

Remember the ClinicPartitionInterceptor? This is where it earns its keep.

It overrides the two Pointcuts we just described. Both hooks do the same small job - produce a partition id - then hand it back to HAPI, which does the actual stamping or scoping. The interceptor itself stays thin.

— HOOK - STORAGE_PARTITION_IDENTIFY_CREATE

Fires on every write

Answers one question - which partition does this new resource belong to? Reads the tenant identity off the request, returns a partition id.

- The hook does NOT know how to turn a slug into an id - it delegates

— HOOK - STORAGE_PARTITION_IDENTIFY_READ

Fires on every read and search

Answers - which partition or partitions may this query see? Same shape, returns a partition id to scope the search.

- Two hooks, one job each, both return a partition id

— CONCEPT - THE PATTERN WE BUILD - NOT A HAPI CLASS

A slug to id lookup on every request needs to be fast. So we cache.

The interceptor needs to turn a clinic slug into a partition id on every call. A database hit per request is too slow - so we put a registry in front of it. This is our pattern; HAPI gives the hooks, the registry is the glue we write.

TenantRegistryService - slug to id with a ~60s TTL

Almost every request is a cache hit with no database cost.

(down)

(down)

3

partitionId returned

60s TTL - a newly onboarded clinic is visible within a minute

● The registry is the bridge from a human slug to HAPI's integer id

● Our pattern to build - HAPI gives the hooks

— DIAGRAM - ONE WRITE, ALL THE WAY THROUGH

Header in. Partition id out. The Patient belongs to Northgate, at the storage layer.

A client posts a new Patient to /fhir/Patient with header X-Clinic-ID set to "northgate". Follow it through the machine.

POST /fhir/Patient - X-Clinic-ID: northgate

CREATE hook to registry to Pointcut to stamped row.

(next)

(next)

3

HAPI stamps PARTITION_ID = 7

the Patient now belongs to Northgate, permanently

- Header missing or slug unknown - REJECT the request, return an error
- Never fall back to DEFAULT - that silently drops it in a shared bucket

— COMPARISON - TWO LEVELS OF ISOLATION - REAL SYSTEMS MIX THEM

Isolate physically where you must. Logically everywhere else.

Partitions inside one database are not the only way to isolate tenants. There are two levels, with very different operational bills.

— LEVEL 1 - SHARED-SCHEMA PARTITIONING

One schema, isolated by PARTITION_ID

Operational simplicity - one connection pool, one set of migrations, one backup. The cost is shared blast radius: a noisy tenant affects everyone, and a buggy READ hook can leak across tenants.

● **Production middle ground - partitions for routine tenancy**

● **Dedicated DB only for tenants with an explicit residency mandate**

— LEVEL 2 - PER-TENANT DATABASE

Physical separation per clinic

Hard separation for data-residency mandates, independent scaling and backup. You pay for it - N migrations, N pools, and no native cross-tenant search, because the data genuinely lives apart.

— CONCEPT - TWO FOOTGUNS - BOTH SILENT

No error. No warning. Just leaked data, discovered weeks later.

Both are dangerous precisely because nothing throws. You find them by auditing - or by a customer noticing someone else's patient.

— FOOTGUN 1 - DEFAULT-PARTITION LEAK

Forget to assign - it lands in DEFAULT

Forget a partition, or let a registry miss fall through instead of rejecting, and the resource lands in DEFAULT. Anything in DEFAULT is visible to any query that isn't strictly partition-scoped. You've effectively published it. Reject, never default.

— FOOTGUN 2 - CROSS-TENANT QUERY BUG

READ hook returns more than the caller's partition

Return a list, return null, return "all" by mistake - and every clinic can suddenly see every other clinic. One wrong return value in one hook, and the entire isolation guarantee is gone.

● **Verify - confirm HAPI 8.x default and null-partition behaviour against your pinned tag**

● **Write a test proving a scoped read can't see the default bucket**

— DIAGRAM - THE WHOLE MACHINE IN ONE PATH

One header in. Correct isolation out. Every box is a slide we just walked.

Five numbered steps tie the chapter together. If you can draw this from memory, you can wire multi-tenant HAPI.

Request to partition - end to end

Cross-references slides 03, 06, 07, 08, 04.

(down)

(down)

3

Registry resolves slug to id

from its 60s cache - slide 07

(down)

4

id flows back to the Pointcut

the partition id HAPI works in - slide 08

(down)

5

HAPI JPA does the work

stamps PARTITION_ID on write - scopes the search on read - slide 04

— TRY IT YOURSELF - PROVE YOUR OWN ISOLATION

Register two clinics. Post a Patient to each. Then drop the header.

Enable partitioning in your pinned HAPI, register northgate and southbay in your tenant table, then run the test that matters.

— THE EXERCISE

POST a Patient with each X-Clinic-ID, then a scoped GET

Confirm a scoped GET for one clinic never returns the other's Patient. Then the negative test - drop the header entirely and confirm the request is rejected, not parked in DEFAULT. If it lands in DEFAULT, you've reproduced footgun one.

Recap

- Header-tenant mode + the partition interceptor
- Tenant registry - slug to id, 60s cache
- Reject, never DEFAULT - scope every READ

— CHAPTER 12 · SUBSCRIPTIONS, BULK DATA, MDM · BEAT 1 — THREE FEATURES, ONE CHAPTER

So far the server answered when we asked. Now it acts on its own — at population scale.

Three features sit together this chapter. Subscriptions, so the server pushes you a notification when something changes. Bulk Data export, so you pull a whole population out asynchronously. And MDM — master data management — the golden record that decides two patient files are really one person.

What you will be able to do

- Subscriptions — criteria push + rest-hook vs message channel
- Bulk \$export — async NDJSON at system, group, patient scope
- MDM — golden record, EID anchor, match scoring

— CONCEPT · THE R4 SUBSCRIPTION — TWO ESSENTIAL PARTS

A criteria search string, and a channel. That is the whole resource.

The criteria is just a search — the exact same search we built two chapters back. The channel is a type plus an endpoint. Match on create or update, and the server fires.

— SUBSCRIPTION — R4

```
{ "status": "requested", "criteria": "Observation?code=http://loinc.org|2823-3",  
  "channel": { "type": "rest-hook", "endpoint": "https://my-app.example/hooks/labs",  
  "payload": "application/fhir+json" } }
```

If you can write the search, you can write the subscription. That is the elegant part.

Three things to point at

- **criteria** — a search string · the same engine as Chapter 09
- **channel.type** — rest-hook here · message is the other option
- fires on create or update of a matching resource

— CONCEPT · THE R4 PAIN — POWERFUL PER-SUBSCRIBER, BUT NO SHARED TOPICS

Can I subscribe to a shared topic everyone reuses? In R4 — no.



Here is the scar — no. In R4 there is no SubscriptionTopic. No shared, curated topic abstraction at all.



Every one. And criteria is single-resource-type — you cannot say Observation or DiagnosticReport in one. And it fires on create or update only — delete a resource and no notification goes out.

— COMPARISON · R5 FIXED IT — AND HAPI BACKPORTS THE FIX ONTO R4

R5 introduced SubscriptionTopic. HAPI brings it back to the R4 wire.

A SubscriptionTopic is a canonical, curated topic that defines the filters and the notification shape once, centrally. Subscribers just reference the topic — and you do not have to leave R4 to get it.

— R4 NATIVE · THE LIMIT

Each subscriber writes its own criteria string

No shared topic. Three teams watching the same event maintain three fragile strings — and they drift. Single-resource-type, no curation.

— R5-STYLE BACKPORT · THE FIX

Subscriptions reference one canonical SubscriptionTopic

Filters and notification shape defined centrally, via HAPI's Backport Subscription implementation guide. Same R4 messages on the wire — now with shared topics.

● **Version-sensitive — enabled in HAPI's topic-based subscription configuration**

● **The model is stable · check your HAPI version's docs for the exact setting**

— COMPARISON · CHANNEL.TYPE — THE TWO THAT MATTER

rest-hook for simple. message for serious.

The channel decides how the notification leaves the server. Two choices carry real weight; websocket and email also exist but are rarely what you want.

— REST-HOOK

Server POSTs to your endpoint

A synchronous HTTP POST to channel.endpoint, optionally with the full resource as payload. Simple, one consumer. Great for a webhook into a single service.

— MESSAGE

Delivered as a FHIR message to a broker

In HAPI this maps to a message-queue — a broker like Kafka. Durable, replayable, fans out to many consumers at once.

● **rest-hook — simple single-consumer webhook · down means the event is lost**

● **message — durable, multi-consumer, replayable fan-out**

● **websocket and email exist — rarely the right reach**

— DIAGRAM · BULK DATA V2 — THE \$EXPORT OPERATION

That was events one at a time. Bulk moves a whole population.

Bulk Data version two is the dollar-export operation, at three scopes. Search answers a question; bulk moves an entire population to analytics, a payer, or a machine-learning job. Different operation class, different mechanics.

● SCOPE · SYSTEM

/\$export

Everything the server holds. The most expensive — bound it carefully.

● SCOPE · GROUP

/Group/[id]/\$export

A defined cohort — the usual choice for a population pipeline.

● SCOPE · PATIENT

/Patient/\$export

One patient compartment. The safest scope to experiment with.

● Why bulk is not search — it moves whole populations, not query answers

● Bound it with `_type` · `_since` · `_outputFormat` · `_typeFilter`

— WALKTHROUGH · BULK EXPORT IS ASYNC BY CONTRACT

A whole population cannot come back in one response. So it does not.

Four steps. There is no synchronous bulk GET — async or nothing.

● STEP · 1 KICKOFF

Prefer: respond-async

You request the export with this header. You are asking the server to work in the background.

● STEP · 2 ACCEPTED

202 + Content-Location

No data yet — a 202 Accepted and a polling URL in the Content-Location header.

● STEP · 3 POLL

202 working · 200 done

Poll the URL. A 202 means still working — honor Retry-After. A 200 means done.

● STEP · 4 DOWNLOAD

NDJSON manifest

The 200 body lists file links — one NDJSON file per resource type. Download them.

● NDJSON — newline-delimited JSON · one resource per line, built for streaming

● One file per resource type · no synchronous bulk GET exists

— CONCEPT · MDM — MASTER DATA MANAGEMENT

The same person, twice. MDM links them into one golden record.

Clinic A and clinic B each file the same human. MDM links those source records into one golden resource — the master. It is the one feature today that writes a new derived resource, not just reads or pushes.

• SCORING

**MATCH ·
POSSIBLE_MATCH ·
NO_MATCH**

Incoming resources score into three buckets. A clear MATCH links automatically.

• ANCHOR

EID

The Enterprise Identifier — a cross-system id saying this is the same person everywhere.

• SURVIVORSHIP

Golden fields

Survivorship rules populate the golden resource's fields from the linked sources.

• REVIEW

POSSIBLE_MATCH

An uncertain match surfaces for a human to confirm — never linked blindly.

● **N source records link to ONE golden resource**

● **MDM writes a derived record — the only write-back feature here**

— DIAGRAM · PUSH, PULL, AND IDENTITY — ONE COHERENT DATA FLOW

Picture one event. All three features wire together around it.

A new Observation is created. No new facts here — just the wiring that makes the chapter one story.

• PUSH

Subscription fires

The matching Subscription fires through its channel — rest-hook or the broker. The event goes out the moment it happens.

• PULL

\$export cohort

A scheduled bulk dollar-export sweeps the same population into an NDJSON cohort. The whole set, on a schedule.

• IDENTITY

MDM golden record

Threading through both — the golden record is the identity every consumer resolves to, so push and pull agree on who the patient is.

● Push — the instant notification

● Pull — the population sweep

● Identity — the deduplicated patient both rely on

— COMPARISON · TWO CAUTIONS THAT BITE IN PRODUCTION — THE SCARS

One silently drops events. The other quietly drains your budget.

These are the failures almost no tutorial names. Internalize both before you scale.

— CAUTION · SUBSCRIPTION DELIVERY RELIABILITY

The in-memory cache is not shared across instances

HAPI's subscription matching cache lives in memory, per node. Run more than one HAPI instance without a shared broker like Kafka, and notifications get silently dropped — no error, just missing events. If you scale out, you need the broker. This is the footgun we teased at the start.

● More than one node without a broker = dropped events (Ch 11 · Ch 21)

● Bound large exports — scope and `_since` are not optional at scale

— CAUTION · BULK EXPORT COST

A full-system \$export is expensive

Full-system jobs run long, consume storage for the NDJSON files, and can time out your polling. Scope to a group or a patient, and bound it with `_since` and `_type`. Do not dollar-export the universe on a whim.

— TRY IT YOURSELF · THE ASYNC HANDSHAKE IN TWO COMMANDS

Kick off a \$export, read the Content-Location, then poll it.

Start at the patient level. Read the polling URL from the response header, then GET it until a 200 returns the NDJSON manifest. If your sandbox does not have bulk enabled, run it against the local docker-compose HAPI from Chapter 07.

— RUN THIS IN YOUR TERMINAL

```
curl -i -H 'Prefer: respond-async' 'https://hapi.fhir.org/baseR4/Patient/$export' # then:  
curl -i '<Content-Location URL>' until it returns 200
```

The 202 hands you a polling URL; keep polling until 200 returns the list of NDJSON files. That is the whole Bulk Data contract, end to end.

Recap

- Subscriptions push
- Bulk \$export pulls
- MDM reconciles identity
- 1 footgun — the non-shared subscription cache

— ACT III | INTEGRATION -- WE TURN THE CORNER

The server is built. Now we build the app that consumes it.

For seven chapters we lived inside HAPI -- architecture, profiles, interceptors, search, partitions. From here we build the application layer, starting with a doorway into FHIR -- the Backend-for-Frontend.

● DTO map

● Retry

● Idempotency

● Error translate

— CONCEPT | BACKEND-FOR-FRONTEND -- ONE BACKEND PER USER EXPERIENCE

The browser never calls HAPI. ddx-api does -- and only ddx-api.

The single legal path to FHIR

A server-side facade built for one frontend's needs (Sam Newman)



3 HAPI FHIR | :18080

The only party ddx-api talks to downstream

— FORBIDDEN EDGE

Browser --[X]--> HAPI :18080

A direct browser-to-HAPI arrow is drawn struck through. That edge never exists. We meet the UI side of this same rule next chapter.

— DISCIPLINE ONE | FHIR → DTO -- DON'T LEAK FHIR TO THE UI

A full FHIR Patient goes in. Four screen fields come out.

— THE PROBLEM | RAW FHIR PATIENT

```
{ "resourceType": "Patient", "meta": { ... },  
  "identifier": [ { "system": "...", "value": "MRN-  
8842" } ], "name": [ { "use": "official", "family":  
"...", "given": [ ... ] } ], "extension": [ ... ] }
```

Nested CodeableConcept and Reference, a meta block, extensions, polymorphic value-x.
Verbose, tenant-revealing, version-coupled.

— THE DISCIPLINE | PATIENTSUMMARYDTO

```
class PatientSummaryDto { id: string;  
  displayName: string; birthDate: string; mrn:  
string; }
```

class-transformer and class-validator behind a ValidationPipe. One-way, explicit. When a FHIR version or profile changes, the blast radius is one file: the mapper.



Never. That is discipline one -- map it. The UI never depends on the FHIR wire shape.



Containment. Profiles and versions can shift downstream -- nothing upstream in the UI moves.

— DISCIPLINE TWO | RETRY WITH BACKOFF -- BOUNDED, NEVER BLIND

5xx is transient -- retry it. 4xx is your fault -- never retry it.

Bounded retry on HAPI 5xx

RxJS `retry({ count, delay })` on the @nestjs/axios `HttpService`

● GET / PUT / DELETE -- idempotent, retry freely

● POST -- retry ONLY when guarded by an idempotency key

— DISCIPLINE THREE | IDEMPOTENT CREATE -- THE DESIGN SPACE, TWO WAYS

A naive POST retry creates two Patients. Two ways to stop that.

— OPTION A | CLIENT IDEMPOTENCY-KEY

POST /api/v1/patients Idempotency-Key: 7f3c-...
→ BFF stores (key → resource id) → replay
returns the ORIGINAL result
App-level. An IETF draft standardizes the
header. The BFF dedups before HAPI ever sees a
duplicate.



Ava

That you confirm in the code -- it is verified against the live server
at build time, not asserted on a slide.

— OPTION B | FHIR-NATIVE IF-NONE-EXIST

POST [base]/Patient If-None-Exist:
identifier=mrn|123 → HAPI conditional create →
match exists? no-op, returns 200 not 201
Server-level. HAPI itself enforces the dedup as
part of R4 conditional create semantics.



Marcus | expert

The BFF exposes one idempotent create to the UI -- and
translates it to one of these downstream.

— DISCIPLINE FOUR | OPERATIONOUTCOME → TYPED ERROR -- TRANSLATE, DON'T FORWARD

HAPI says no in FHIR. The UI hears it in clean HTTP.

— PARSE VIA EXCEPTION FILTER → CLEAN BODY

```
{ "statusCode": 404, "message": "Patient not found", "code": "not-found" }
```

An OperationOutcome is a list of issues -- each with a severity, a code, and diagnostics. The BFF never forwards that verbose, internal-leaking payload. It throws a typed NestJS exception through a filter.

• NOT-FOUND

404 NotFound

Resource id absent

• INVALID

400 / 422

Bad or Unprocessable -- exact status verified live

• SECURITY

403 Forbidden

Auth / scope denial

• PROCESSING

502 BadGateway

Upstream failure

The typical mapping above is the discipline. The exact issue.code → HTTP status HAPI emits is implementation-specific -- whether a failed validate returns 422 or 400 you confirm against the live :18080 at build time. The discipline is fixed; the table you verify.

— PROPAGATION | WHO THE BFF IS WHEN IT CALLS HAPI

Two headers, injected once -- never plumbed call by call.

Central injection on the shared `HttpService`

An axios request interceptor reads from request-scoped context

**3**

Authorization: Bearer <service token>

The BFF's own downstream credential

4

X-Clinic-ID: <tenant slug>

Drives HAPI's partition interceptor -- slug → partitionId, from the multi-tenant chapter

— THE TRUST RULE

Never trust a client-supplied X-Clinic-ID

Derive and validate the tenant from the verified token. The end-user login itself -- SMART and OIDC

DUDOXx is a later chapter; here we carry only the downstream service identity.

— CODE | THE WRAPPER WHERE ALL FOUR DISCIPLINES LIVE

One seam every HAPI call flows through. The FHIR mess stops here.

— ILLUSTRATIVE SKELETON -- NOT PRODUCTION SOURCE

```
@Injectable() class FhirGateway {  constructor(private http: HttpService) {}  async getPatient(id: string): Promise<PatientSummaryDto> {    return firstValueFrom(      this.http.get(`/Patient/${id}`).pipe(        retry({ count: 3, delay: backoffWithJitter }),        map(r => toPatientSummaryDto(r.data)),      ),    );  } }
```

● **Interceptor stamps the two headers -- on the way out**

● **retry guards transient 5xx -- on the way out**

● **map → DTO -- on the way back**

● **filter translates `OperationOutcome` -- on the way back**

Every controller in ddx-api just calls a clean, typed method. You write the seam once.

— SYNTHESIS | TRACE ONE REAL CALL -- THE WHOLE BFF IN ONE PICTURE

web → ddx-api → HAPI → back -- four disciplines, one rail.

GET /api/v1/patients/:id

Where each discipline fires, annotated on the lifecycle

4 Resource branch → DTO map
PatientSummaryDto returned as clean JSON

5 OperationOutcome branch → error filter
Typed exception -- clean { statusCode, message, code }

● Headers

● Retry

● DTO map

● Error translate

— RECAP & BRIDGE | THE BOUNDARY DISCIPLINE, IN YOUR EDITOR

Sketch the wrapper. Trace one call. Then cross the boundary.

— TRY IT YOURSELF

Build the seam, then trace a GET

Sketch the `HttpService` wrapper -- inject the two headers from request context, retry 5xx three times with backoff and jitter, map one `OperationOutcome` code to a typed exception. Then trace a single GET Patient call: web → ddx-api → HAPI.

● BFF

● DTO

● Retry

● Idempotency

● Errors

— NEXT CHAPTER

Ch 14 -- Next.js 16 rendering FHIR

We cross to the other side of this same boundary -- the BFF rule seen from the UI. Same rule, new vantage point.

— CHAPTER 14 · NEXT.JS 16 + FHIR · COLD OPEN

Last chapter we built the BFF. Now we stand in front of it and render FHIR.

Next.js sixteen, rendering real Patients, Encounters, and Observations from the NestJS BFF we built last chapter. Three goals: the one rule that governs every healthcare frontend, fetching with Server Components, and rendering FHIR resources without crashing on missing data.

What you will be able to do

- The BFF rule — the browser never touches HAPI
- Server Components — async fetch via the BFF
- Rendering resources — defensively, every time

— THE RULE · THE LOAD-BEARING POSTURE OF THIS WHOLE CHAPTER

THE BFF RULE — the browser never talks to HAPI. Ever.

• HOP 1 · CLIENT

Browser

Holds no FHIR token, no HAPI URL, no clinic header. Calls only same-origin /api/v1/.*

• HOP 2 · SAME-ORIGIN

Next.js route

/api/v1/* in ddx-web. Forwards the request server-side to the BFF.

• HOP 3 · SERVER ONLY

ddx-api BFF

The BFF we built last chapter. Holds the bearer token and X-Clinic-ID; only it reaches HAPI :18080.

• DOWNSTREAM

HAPI :18080

FHIR server. Reached by the BFF alone, with bearer + tenant header injected server-side.

— FORBIDDEN · THE SHORTCUT THAT LEAKS CREDENTIALS

~~Browser~~ → ~~HAPI :18080~~

Skip the BFF and the access token lives in the browser where anyone can read it, and the tenant header becomes client-controlled, breaking isolation. Three hops, token off the browser, tenant injected server-side — that is the whole posture.

• ALLOWED — Browser to /api/v1/* to BFF to HAPI

• FORBIDDEN — Browser direct to HAPI

— WALKTHROUGH · APP ROUTER · THE PAGE IS AN ASYNC FUNCTION

A Server Component awaits the BFF — credentials never enter the bundle.

In the App Router a page becomes an async function and awaits its data right there. Because Server Components render only on the server, the bearer and tenant header are safe to send — the query logic and credentials never reach the client bundle.

— APP/PATIENT/[ID]/PAGE.TSX · SERVER COMPONENT

```
export default async function Page({ params }) {  
  const [patient, encounters, observations] = await Promise.all([ api.getPatient(params.id),  
    api.getEncounters(params.id), api.getObservations(params.id) ]); return <Chart patient={patient}  
    encounters={encounters} observations={observations} />; }  
}
```

— V16 GOTCHA · FETCH

Not cached, blocks render

In Next.js sixteen fetch is no longer cached by default and blocks render. Wrap the fetching component in Suspense to stream, or reach for the cache directive.

— V16 GOTCHA · PARALLEL

Promise.all, not waterfall

Patient, Encounters, and Observations fire together and await with Promise.all — never one after another. Or give each its own Suspense boundary to stream independently.

— FRAMEWORK ANCHOR · THE CLIENT-COMPONENT PATH (CONCEPTUAL)

On the client, a typed client calls methods — not URLs.

— CONCEPTUAL SHAPE · NOT A LITERAL SIGNATURE

const patient: Patient = await api.getPatient(id); // targets /api/v1/*

Interactive widgets and forms call a method and get back a typed FHIR resource — instead of hand-writing fetch to /api/v1/*. Server Components call the BFF directly; this typed client is the client-component path.



Ava

That is where a typed client earns its keep — one method per resource, a typed Patient back.



Marcus · one seam, not many

Exactly. Cardinal twenty — extend the existing seam. One client, one place to evolve.

— WALKTHROUGH · PATIENT · ALMOST EVERY FIELD IS OPTIONAL

Compute the display name. Show the MRN — never the resource id.

Name is an array of HumanName: pick the official use if present, else the first; render its text, or join given and family. Birth date can be year-only and still valid. Telecom and address are arrays you filter and tolerate empty.

• NAME · HUMANNAME[]

Display name

use=official else first; name.text, else given + family. Never assume name[0] exists.

• IDENTIFIER · IDENTIFIER[]

MRN

The identifier whose type code is MRN — the human-facing medical record number.

• BIRTHDATE · DATE

Partial dates valid

A year-only birthDate is legal. Render what is there; do not pad or guess.

• TELECOM · ADDRESS

Filter, tolerate empty

ContactPoint[] and Address[] — filter by system phone/email; an empty array is normal.

— THE TRAP · RESOURCE ID IS NOT THE PATIENT NUMBER

patient.id // server-assigned logical id — NOT the MRN

The resource id is the server-assigned logical id. Never surface it to a clinician as the patient number. The human-facing medical record number lives in identifier, the one with the MRN type code — show that.

— DIAGRAM · ENCOUNTER · A TIMELINE SORTED BY PERIOD.START

Sort by period.start. Badge status and class. Absent end means ongoing.

Encounters become a timeline ordered by period.start — and you handle a null start, because a planned encounter may not have one yet. References carry a display string, so render subject.display and participant.display and skip the extra fetch.

● PERIOD.START · EARLIEST

2026-01-12 · AMB

Status finished. Class ambulatory.
subject.display rendered straight from
the reference.

● finished

● PERIOD.START · RECENT

2026-04-03 · IMP

Status in-progress. Class inpatient.
period.end absent — so we render
ongoing.

● in-progress · ongoing

● PERIOD.START · NULL

planned · EMER

No start yet. A planned encounter still
has a place — do not drop it from the
list.

● planned

● class AMB / IMP / EMER — ambulatory, inpatient, emergency

● absent period.end renders "ongoing", never a blank cell

— WALKTHROUGH · OBSERVATION · WHERE CH 03 AND CH 04 PAY OFF

Switch on value[x]. Show the unit; compare on the UCUM code.

value[x] is a choice type. Most vitals and labs use valueQuantity — the Quantity we met in Chapter three — but it might be a CodeableConcept or a string, so switch on which one is present.

— VALUEQUANTITY · UNIT VS CODE

```
{ "value": 70, "unit": "kg", "system":  
  "http://unitsofmeasure.org", "code": "kg"  
}
```

Show value and unit to the human — 70 kg. But the canonical unit is the UCUM code from Chapter four. Display unit; compare and convert on code.

— COMPONENT[] · BLOOD PRESSURE

component: [systolic, diastolic]

Blood pressure is two component entries — systolic and diastolic — each its own Quantity. Render both, each with its own value and unit.

• DISPLAY LADDER · 1

.text

The human-readable original. Prefer it — callback to the CodeableConcept of Chapter three.

• DISPLAY LADDER · 2

coding[].display

First coding's display, optionally the system you prefer — SNOMED or LOINC from Chapter four.

• DISPLAY LADDER · 3

bare code = defect

A raw code shown to a clinician, e.g. 38341003, is a defect. Last resort only, flagged.

— CONCEPT · THE DEFINING TRAIT OF REAL FHIR DATA IS SPARSENESS

Assume absence, not presence — one bad resource degrades one card.

Nearly every element is optional, arrays come back empty, and a choice type means exactly one variant exists. The renderer must assume absence, not presence.

— BAD · ASSUMES PRESENCE, CRASHES

patient.name[0].text · coding[0].code · period.start

Index zero into an empty array and the whole chart throws. The clinician sees a blank page, not a missing field. One sparse resource takes down everything.

— GOOD · NULLISH-SAFE, DEGRADES GRACEFULLY

patient.name?.[0]?.text ?? "Not recorded"

Reach every value through nullish-safe accessors. Show a neutral placeholder — a dash, or "Not recorded" — never the word undefined.

• RULE 1

Never index zero

Never assume coding[0], name[0], or period.start exist. Reach them safely.

• RULE 2

Neutral placeholder

Missing renders "—" or "Not recorded" — readable, not a stack trace.

• RULE 3

dataAbsentReason

Distinguish plain absence from dataAbsentReason — FHIR telling you it is missing on purpose.

• RULE 4

Per-card boundary

Wrap each card in an error boundary — one malformed resource degrades one card, not the page.

— DIAGRAM · THE SYNTHESIS · ONE PAGE, THE WHOLE FLOW

Page fires Promise.all to the BFF; three components render defensively.

The page is a Server Component. It fires Promise.all to the BFF — Patient, Encounters, and Observations together. Each result flows into its own component, each rendering defensively.

— SERVER SIDE · THE LINE YOU CANNOT CROSS

Page (Server Component) ----> Promise.all ----> ddx-api BFF ----> HAPI :18080

The bearer token and the tenant header live only on the server side of this diagram. The browser never sees them.

● RENDER · SERVER

PatientCard

Display name, MRN, telecom — nullish-safe, neutral placeholders.

● RENDER · SERVER

EncounterTimeline

Sorted by period.start; status and class badges; "ongoing" for absent end.

● RENDER · SERVER

ObservationList

Switch on value[x]; unit for humans, UCUM code for compare; display ladder.

● CLIENT · INTERACTIVE

One widget

A single interactive widget takes the ClientApiClient path back to /api/v1/*.

— RECAP · TRY IT YOURSELF · AND THE BRIDGE TO CH 15

Three things to keep: the rule, the parallel fetch, the defensive render.

● The BFF rule — browser to your BFF, never to HAPI

● Server-render — fetch in parallel, credentials off the client

● Defensive — almost nothing is required

— TRY IT · ONE PROMISE.ALL, THREE RESOURCES

```
const [patient, encounters, observations] = await Promise.all([ api.getPatient(id),  
  api.getEncounters(id), api.getObservations(id) ]);
```

One Promise.all to /api/v1/*, three resources at once — never hapi.fhir.org directly. That single call is the whole chapter in practice.

● Next · Ch 15 · SSE — live updates pushed to the UI when a clinician saves a new Observation

— 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.

HAPI write → re-render · the five-layer loop

Each layer is numbered, tinted, and previewed below.

3

Redis Streams

The event is appended to a durable, channel-scoped log.

4

S S E gateway

Reads the stream and writes it out as S S E frames over HTTP.

5

Browser EventSource

Consumes frames, updates state — the open view re-renders, live.

— 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.

Connection lifecycle

Authorize before any subscription.



Replay / fan only entitled

The gateway streams only what the connection is allowed to see.

— 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.



3 BFF XADDs
sse:{clinic}:patient — tenant-scoped.

4 Gateway streams
The frame goes out with its id:.

5 EventSource → re-render
Zod-validates · the Ch 14 PatientCard updates, live.

— 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

— ACT III - INTEGRATION - TEST DATA

Deterministic data is the bedrock under every test, demo, and dev server.

Last chapter we wired the SSE event contract. Now we step back to something quieter but just as load-bearing. Deterministic means the same seeder run always produces the same resource graph - same IDs, same references, same values. That is what lets a test assert on a known ID, gives a demo a stable patient roster, and makes every developer's local server identical.

The four disciplines that protect it

● Idempotent

● Phased

● Atomic

● Tenant-aware

— DISCIPLINE 1 - IDEMPOTENT SEEDING

Run the seeder twice - get one identical state, never a duplicate.



Ava

Nothing. Same state, no duplicates - that is idempotency. The footgun is the naive loop where every run adds another Patient.

Two ways to converge

An idempotent run leaves exactly one record.

— DISCIPLINE 2 - PHASE ORCHESTRATION

Seed the referenced before the referencing - every reference resolves.

FHIR resources point at each other - an Encounter references a Patient, an Observation references both Patient and Encounter, a Patient references a managing Organization. Seed in dependency order so every reference target already has a known ID.

Five phases, one dependency rail

5**Observation**

References Patient and Encounter - both already seeded.

• **Alternative - one transaction Bundle with urn:uuid forward-refs lets HAPI resolve intra-Bundle order (Ch 05)**

• **But phase staging keeps each batch small and debuggable**

— DISCIPLINE 3 - ATOMIC BATCHES

Wrap each phase in a transaction Bundle - all entries commit or all roll back.

HAPI commits every entry in a transaction Bundle, or rolls the whole batch back on any failure - the same atomicity we built in Chapter 05. A loop of single REST calls can fail halfway and leave you torn; the transaction Bundle is the seed-batch primitive you want.

— BUNDLE.TYPE = TRANSACTION - IDEMPOTENT ENTRY

```
{ "fullUrl": "urn:uuid:pat-1", "request": {  
  "method": "POST", "url": "Patient",  
  "ifNoneExist": "identifier=mrn|001" } }
```

urn:uuid fullUrls carry the forward-references; If-None-Exist on each request makes a matching re-run a clean no-op - atomicity and idempotency combined.

— WHY NOT A LOOP OF CALLS

A loop half-applies on error -> torn state

Individual REST calls have no shared rollback. One failure mid-loop leaves some resources created and others missing - exactly the partial dup a transaction Bundle prevents.

— DISCIPLINE 4 - TENANT-AWARE SEEDING

Loop the clinics - one partition per batch, fixed IDs coexist.



Marcus - production scar

For each clinic slug, set X-Clinic-ID, then run the whole phase pipeline. The partition interceptor from Chapter 11 resolves the header to a partition ID.

Clinic loop -> partition isolation

The same seed-pat-001 exists once per tenant.

— RESET - A TRUE BLANK SLATE

A soft delete keeps history. A hard wipe starts genuinely empty.

Tests need a clean slate between runs - but a normal FHIR delete is a soft delete, and re-seeding the same fixed IDs then collides with retained version state.

— THE TRAP - SOFT DELETE

Tombstone, history retained

A normal DELETE marks a tombstone and keeps the history rows. Re-seed the same fixed IDs and you collide with retained version state.

● **Re-seed collision**

— THE CLEAN PATH - HARD WIPE

Expunge resources and their history

The delete-expunge operation hard-removes resources and their history rows by a search match, so re-seeding starts truly empty. Scope it to one partition - run under that clinic's X-Clinic-ID - to reset one clinic without touching siblings.

● **Destructive - gated - dev/test only**

It is destructive and gated behind an expunge config. The exact parameters and the enable key you confirm against your HAPI version.

— TWO SOURCES - ONE PIPELINE

Hand-curated fixtures for assertions; generated volume for everything else.

A typical pipeline uses both - a small fixture core the tests assert on, plus an optional generated bulk layer the tests merely tolerate.

— FIXTURES - HAND-AUTHORED

Fixed IDs, known clinical content

Version-controlled resource files with exact, recognizable values. Reach for these when a test asserts on a known ID, or a demo needs a stable roster.

● **Assertion-grade**

— GENERATED - SYNTHESIZED AT VOLUME

Synthea-style or a parameterized factory

Programmatic volume for load tests, search-result variety, and populated-UI screenshots. Determinism still applies - seed the random generator from a fixed value so volume is reproducible too.

● **Fixed seed -> reproducible**

— SYNTHESIS - THE CONCEPTUAL SEEDER SHAPE

All four disciplines on one rail - the pattern, not a literal tool.

Think of it as the Dudoxx seeder shape - conceptual, the pattern a seeder like ddx-fhir-seeder would follow. Outermost is the clinic loop; teardown lives beside it.

Clinic loop -> phase pipeline -> reset path

● **Phased**

● **Atomic**

● **Idempotent**

● **Tenant-scoped**

— TRY IT - RECAP - NEXT

Write a two-phase idempotent seeder - then run it twice.

— TRY IT YOURSELF

PUT /Organization/seed-org-001 -> then a transaction Bundle creating a Patient with If-None-Exist on the MRN

Run it twice and confirm the second run changes nothing - same IDs, no duplicates. Then wrap it in a clinic loop that sets X-Clinic-ID per tenant.

What you can now do

● Deterministic

● Idempotent

● Phased

● Atomic

● Tenant

● Reset

● Next - Chapter 17 - SMART on FHIR + RBAC

— CHAPTER 17 · SMART ON FHIR 2.2 · ZERO-TRUST RBAC

Authentication is not authorization. A FHIR server has to prove both.

We built and seeded the server. Now the hardest question - who may read this patient's blood pressure, and who may not. Authentication asks who you are. Authorization asks what you may touch. Confuse the two and you leak protected health information. SMART on FHIR answers both - and it is live regulatory pressure, with the TEFCA FAST security deadline landing in January 2026.

What we will cover

- App launch & the OAuth2 / OIDC flow
- The version-two scope grammar
- The zero-trust enforcement chain

— CONCEPT · SMART APP LAUNCH 2.2.0 - OAUTH2 + OIDC, PROFILED FOR HEALTHCARE

Four players. The FHIR server never sees a password - only a token.



Ava & Marcus

● SMART 2.2.0 · STU on R4

OAuth2 plus OpenID Connect, profiled for healthcare. Version 2.2.0 is the current trial-use release - it carries the granular v2 scopes we are about to meet.

The four parties

SMART is OAuth2, with healthcare roles

4

User

Clinician or patient - grants consent

— DIAGRAM · AUTHORIZATION-CODE FLOW - PKCE MANDATORY FOR EVERY CLIENT

How the app earns its token - code first, then exchange.



Ava & Marcus

• PKCE · S256

Because in 2026 PKCE is mandatory for all clients - even confidential ones. It defends the code if it is ever intercepted.

The code-then-exchange dance

Two round-trips to the authorization server

4

Access token (+ ID token)

openid fhirUser → identity layer

— CONCEPT · LEAST PRIVILEGE, WRITTEN AS A STRING

A v2 scope reads like a sentence - context, resource, permissions.

— THE GRAMMAR

`{context}/{Resource}.{cruds}?params`

cruds spells out five permission letters - c r u d s - create, read, update, delete, search. The context says whose data, the params narrow it further.

● **patient/Observation.rs** - read + search this patient's observations

● **user/*.cruds** - full access to everything the user can reach

● **patient/Observation.rs?category=vital-signs** - vitals only

— THE C-R-U-D-S LETTERS

c create · **r** read · **u** update · **d** delete · **s** search.
Combine only the letters an app actually needs
- that is least privilege by construction.

— THE THREE CONTEXTS

patient/ just this patient · **user/** everything the logged-in user may see · **system/** backend services, no user in the loop.

— COMPARISON · V1 COARSE VS V2 GRANULAR - AND THE RULE THAT BITES

Read-everything became read exactly what you need.

Version-one clinical scopes were coarse - read-all or write-all. Version two replaces them with per-letter cruds, so you grant exactly what an app needs and nothing more. A server may negotiate or downgrade - but never mix the two grammars in a single request. They are distinct.

— V1 · DEPRECATED

`patient/Observation.read`

`patient/Observation.write`

Coarse - read everything or write everything. HTI-1 and US Core 7 deprecate dot-read and dot-write for ONC-certified systems. Non-compliance is a certification failure.

● **Never mix v1 and v2 grammars in one authorization request**

— V2 · LEAST PRIVILEGE

`patient/Observation.rs`

Read + search, narrowed with

`?category=vital-signs`

Per-cruds letters grant exactly the access an app needs. In 2026 version two is not a nicety - it is the requirement.

● **v2 is least-privilege by construction**

— CONCEPT · HOW THE APP LEARNS WHICH PATIENT IS ON SCREEN

Same scopes, two doorways - EHR launch and standalone.

When the app opens, how does it know which patient is in front of the clinician? That is launch context, and there are two ways in. The difference is whether the host EHR hands the app a session, or the app starts cold and asks for what it needs.

— EHR LAUNCH

Launched from inside the record

The app receives a **launch** parameter tied to the EHR session and requests **launch/patient**. The authorization server resolves context and returns the patient and encounter ids alongside the token - plus an array of typed context references, an intent, and a styling url so the app matches its host.

— STANDALONE LAUNCH

The app starts cold

No **launch** parameter. The app requests the context scopes it needs directly, and the user picks the patient during login.

● **fhirContext** = typed context references, not a single named field

● Same scope grammar, different entry point

— WALKTHROUGH · FRAMEWORK ANCHOR - WHERE AUTHORIZATION LANDS

The scope string literally becomes a guard rule.



Ava & Marcus

• PRE_HANDLED seam · Ch 10

The resource server. In HAPI an authorization interceptor - configured with a rule-builder pattern. Conceptually you write rules like allow read of Observation inside this patient's compartment, and the scope string drives which rules apply.

The enforcement chain

Decision happens before the FHIR op runs

4

Access decision at the seam

Allow or deny - before the op

— CONCEPT · ZERO TRUST - NO SESSION YOU CAN RIDE

Every request re-validates the token. A valid signature is not permission.

Authorization lands on every single request - that is zero trust. No implicit trust, no session to coast on. Each call re-validates the bearer JWT through four checks, and even a flawless token can still earn a 403 if its scope does not cover the resource.

• CHECK 1

Signature

Verified against the authorization server's JWKS public keys.

• CHECK 2

exp

The token has not expired.

• CHECK 3

aud

Minted for this resource server.

• CHECK 4

iss

From the authorization server we trust.

— THE PART PEOPLE MISS

A valid signature proves who, not what. Scopes still gate the operation - a perfectly signed token still gets a **403** when its scope does not cover that resource. An expired or replayed token dies right here, at the seam.

— DIAGRAM · SYNTHESIS - ONE APP SESSION, EVERY GATE LABELLED

The whole chapter on one canvas.

Get the token

Launch through token exchange

4

Access token

+ scopes + patient context

Use the token

Validate, then enforce the scope

8

Match → data · Miss → 403

Returned or refused

● launch · PKCE · scopes · context · validate · enforce - one picture

— RECAP · TRY IT YOURSELF - THEN FORWARD TO CHAPTER 18

Decode a token, write three scopes, map each to one check.

— TRY IT · DECODE A TOKEN

Grab a sample SMART access token and decode it at `jwt.io`. Read the claims - `iss`, `aud`, `exp`, the `scope` list, and `patient`.

— TRY IT · WRITE THREE SCOPES

For a vitals-dashboard app:

`patient/Observation.rs?category=vital-signs`,
`launch/patient`, and `openid fhirUser`. Map each to the single interceptor check it becomes.

● App launch & OAuth

● PKCE

● v2 scope grammar

● Launch contexts

● Enforcement chain

● Zero-trust JWT

● Next: Chapter 18 - FHIR over MCP for LLM agents · same gate, system/ scopes, no human in the loop

— CHAPTER 18 · AGENTS & MCP

Giving an LLM agent safe hands on FHIR.

A model can reason about a patient beautifully -- but only if it can reach the data. Clinical data lives in FHIR. This chapter is how an agent talks to it safely: typed tools, a consent gate, and an orchestration layer that knows when to pause for a human.

● MCP tools over FHIR

● Read vs write + human in the loop

● Mastra + BIRDY orchestration

— THE PROTOCOL · MODEL CONTEXT PROTOCOL

MCP: a host spawns one client per server.

● HOST

The agent runtime

Claude, or an agent runtime, spawns one client for each server it connects to.

● Data layer: JSON-RPC 2.0

● CLIENT

One per server

Each client holds one stateful connection. Messages are JSON-RPC two point zero.

● Tools / Resources / Prompts

● SERVER

Capabilities

Offers Tools, Resources and Prompts. Tools is the one this chapter uses.

● Current spec: 2025-11-25

● TRANSPORT

stdio or HTTP

stdio for a local process; Streamable HTTP for remote servers with auth.

— THE SHIFT · URLS VS TOOLS

The model picks a tool -- it never builds a URL.

— RAW HTTP · BRITTLE

The agent builds the request

The model constructs a URL -- GET Patient, name equals -- then paginates, parses a searchset Bundle, and untangles an OperationOutcome by hand. Every step is a chance to get it wrong, and it touches the token directly.

— TYPED MCP TOOL · CALM

The agent fills a schema

A named tool whose input schema says exactly which arguments are legal. The server holds the tenant and auth context, so the model never sees a token. Structured content comes back -- the model just reads it.

— ANATOMY · A TOOL WRAPPING A FHIR OP

A tool is four fields -- and a schema is a contract.

• NAME

search_patient

The handle the model invokes via tools/call.

• TITLE + DESCRIPTION

Human label

What the model reads to decide when to reach for it.

• INPUTSCHEMA

JSON Schema

Constrains the args -- search params, or a required resource id.

• DISCOVERY

tools/list

The host lists tools, then runs one with tools/call.

— ILLUSTRATIVE TOOLS WRAPPING HAPI

Tools like **search_patient** (read), **read_observation** (read), and **create_encounter** (write). Names are illustrative -- one named tool per clinical operation you want the agent to have.

— THE CONTRACT

The model cannot pass a parameter the schema rejects. The server takes the validated arguments, calls HAPI, and feeds the structured result back.

— THE SPLIT · FOUNDATION OF THE SAFETY MODEL

A wrong read wastes tokens. A wrong write is real.

— READ-ONLY · AUTO-RUNNABLE

search · read

No side effects. An agent can run them all day -- worst case it learns something it did not need. A hallucinated search costs a few tokens, nothing more.

— MUTATING · ALWAYS GATED

create · update · delete

Creating an Encounter, writing a MedicationRequest -- these change clinical state. A hallucinated medication order is a patient-safety incident. So the rule writes itself.

● Read-only runs unattended

● Mutating never runs without a gate

● Which is where the human comes in

— HUMAN IN THE LOOP · THE HOST OWNS THE GATE

The host must get explicit consent before any tool call.



Clinical agent

● Awaiting consent

Book a follow-up encounter for this patient next Tuesday.

The consent gate

MCP cannot enforce at the wire -- the host builds it



3

Elicitationspec-native confirm channel



4

Clinician approvesonly then does it write



5

Annotations untrustedunless the server is trusted

— CONCEPTUAL · A DUDOXX BRIDGE SERVER

The MCP server is just another well-behaved FHIR client.

ddx-fhir-r4-mcp

A local stdio process bridging the agent to HAPI



3

Bearer + X-Clinic-ID the same seam from Ch 13



4

HAPI :18080 the FHIR server we have built

— NOT A NEW AUTH SYSTEM

It reuses the bearer token and X-Clinic-ID seam from Ch 13, and the SMART scopes from Ch 17 still apply -- a read tool needs `patient/Observation.rs`, like any client.

— TENANT ISOLATION HOLDS

The partition interceptor from Ch 10 still isolates the tenant, even when an agent is the caller. The server gets no backstage pass -- the agent inherits the seams we already built.

— CONCEPTUAL · SHAPE, NOT A VERBATIM API

A workflow is the recipe; tools are the ingredients.

A clinical workflow

A durable sequence of steps over FHIR tools



3

Executor runs them, step by step

4

Reporter assembles the answer

— MASTRA · TYPESCRIPT AGENT FRAMEWORK

The workflow primitive composes steps in order, and each step can call one of our tool-backed FHIR operations.

— VERSION-SENSITIVE

The exact Mastra API moves quickly. Hold onto the shape, not the signatures. What matters: a workflow is a durable sequence, and any step can reach FHIR through a tool. Durable is about to matter a lot.

— CONCEPTUAL · THE MECHANISM BEHIND THE GATE

The read just runs -- the write waits, and nothing is lost.



Workflow runner

● Suspended

Summarize recent vitals, then create the follow-up encounter.

Suspend / resume

The human-in-the-loop gate, made real in the workflow



3 Surface decision clinician sees the proposed write



4 Resume on approval picks up exactly where it left



5 Write performed principle = consent, mechanism = this

— A DUDOXX PATTERN · NOT AN INDUSTRY STANDARD

BIRDY is the conductor; the MCP tools are the instruments.

• COORDINATOR

Receives intent

"Summarize recent vitals and book a follow-up" arrives as one clinical intent.

• DISPATCH

Routes to sub-agents

Each piece goes to a specialized tool backed by an MCP-exposed FHIR op.

• AGGREGATE

Collects results

It assembles what comes back into one coherent answer.

• GATE

HITL on mutation

On anything that writes, it applies the same human-in-the-loop gate.

— ORCHESTRATION ABOVE THE TOOLS

The tools, the read-write split, the consent gate, suspend and resume -- those are the parts. BIRDY decides which parts, in what order, for a given clinical intent.

— CONCEPTUAL · OURS, NOT A STANDARD

BIRDY is a Dudoxx-internal orchestrated-dispatch pattern. Frame it as the orchestration layer sitting above the ddx-fhir-r4-mcp tool surface.

— DEFENSE IN DEPTH · NO SHORTCUT FOR THE AGENT

The agent inherits the trust model, plus two human gates.

Five layers

An agent is the most powerful client -- so the most scrutiny



3 MCP consent consent + Elicitation gate the call



4 Suspend / resume holds every write for a human



5 AuditEvent + Provenance every mutation -- the whole of Ch 20

— NO NEW POWERS

The agent does not gain privileges. It inherits the same multi-layer trust model as any client -- plus two new human gates: consent, and the suspended write.

— FORWARD TO CH 20

Every mutation emits an AuditEvent and a Provenance record. That accountability layer is the whole of Ch 20 -- here it is the final safety net.

— SYNTHESIS · ONE REQUEST, START TO FINISH

Powerful where it is safe; paused where it counts.

The read path

Clinician prompt to Bundle



3 **stdio to ddx-fhir-r4-mcp**scopes + partition apply



4 **HAPI :18080**returns a Bundle

The write gate

Where the human sees the one moment that matters



3 **Clinician approves**the one moment they see



4 **Resume + write**the write lands



5 **AuditEvent emitted**scopes, partition, consent, audit

— TRY IT · THE SAFE HALF

Feel the tool-shaped response -- then wire it up.

— READ-ONLY SEARCH · PUBLIC SANDBOX

Point a search at the public HAPI sandbox:

```
hapi.fhir.org/baseR4/Patient?  
name=smith&_count=2
```

That structured result is exactly the shape an MCP read tool hands your agent.

— WIRE IT UP · CONCEPTUAL

The `claude.json` config takes a server stanza -- conceptually, an entry for a FHIR sandbox server:

```
~/.claude.json -> mcpServers: ddx-fhir-sbx
```

The shape you would configure, not a verified config.

● MCP tools over FHIR

● Read vs write + consent

● Suspend / resume + BIRDY

● Next: Ch 19 -- Telemedicine + calendar + RAG

— ACT III · APPLIED FHIR

A telemedicine visit is four FHIR resources — plus two layers FHIR does not model.

Underneath the booking is a chain of four scheduling resources. Bolted on by convention: a video room, and a searchable index of the patient's clinical documents.

● Scheduling quartet

● Calendar flow

● Video tie-in

● Clinical RAG

— DIAGRAM · THE SCHEDULING QUARTET

Four resources, one reference chain.

The reference chain

Read it top to bottom — each resource points at the one above.

↑ `Schedule.actor`

↑ `Slot.schedule`

SL

Slot

one window — free, busy, with start and end

↑ `Appointment.slot`

AP

Appointment

consumes Slots, names attendees via `participant.actor`

— WHY PRACTITIONERROLE HEADS THE CHAIN

PractitionerRole binds the clinician to an organization, a location, and a specialty. The calendar is for "Dr Okonkwo as a cardiologist at the Harbor Clinic" — not the bare Practitioner.

— PARTICIPANT.ACTOR RE-ASSERTS THE BOOKING

On the Appointment itself, `participant.actor` names both the patient and the practitioner. The booking is self-describing — you never have to walk back up the chain to know who is in the room.

— WALKTHROUGH · APPOINTMENT.STATUS

The happy path, and the three branches off it.

Happy path

Ten codes in the value set — these five are the spine.



3 arrived / checked-in
present in clinic or joined the room



4 fulfilled
the encounter is complete



— OFF THE HAPPY PATH

● **cancelled**

● **noshow**

● **waitlist**

— TWO CONSTRAINTS TO MEMORIZE

Only proposed or cancelled may omit start and end — anything booked or later must be timed. A cancellation reason applies only when the status is cancelled or noshow.

— PARTICIPANT.STATUS ROLLS UP

Each participant carries accepted, declined, tentative, or needs-action — often mediated by an AppointmentResponse — and those roll up into the overall status.

— CONCEPT · SLOT AVAILABILITY + SCHEDULE.ACTOR

The Slot is the unit of availability.

Each Slot carries a status and a start and end. Availability lives on the Schedule's actor — that is how a free Slot ties back to a specific PractitionerRole.

— SLOT.STATUS VALUE SET

● free

● busy

● busy-unavailable

● busy-tentative

● entered-in-error

— FIND AN OPEN SLOT

GET Slot?

**schedule.actor=PractitionerRole/123
&status=free&start=ge2026-05-25**

Some servers expose a dedicated find operation for availability — but where that is not available, plain Slot search by status free is the dependable baseline. Confirm which your HAPI deployment supports rather than assume.

— WALKTHROUGH · CALENDAR FLOW — CONCEPTUAL

Discover, book, busy-out, launch.

The shape of the Dudoxx calendar flow

Conceptual — your bindings confirm the details.



3

Busy-out

flip the chosen Slot to busy



4

Launch

the booked Appointment anchors the consult

— THE BFF OWNS THE UNIT OF WORK

The backend performs slot-search-then-book as one server-side unit — it never trusts the browser to pick a slot that might already be taken.

— ATOMIC — CALLBACK TO CHAPTER 05

Steps two and three happen together in a transaction Bundle: book the Appointment and busy-out the Slot atomically, so a race can never leave a booking against a slot someone else grabbed.

— CONCEPT · VIDEO TIE-IN — THE DESIGN SPACE

FHIR has no video-room field. Two ways teams wire it.

Neither LiveKit nor Jitsi is modelled in FHIR — the integration is convention, not a core element. Present both; assert neither.

— OPTION A · DETERMINISTIC ROOM NAME

Derive the room identity from the Appointment logical id — stable and idempotent — and store the join link in Appointment.supportingInformation or a linked endpoint.

— OPTION B · BFF-SIDE MAPPING

Keep a backend-side mapping keyed by the appointment id and store nothing extra on the resource itself.

— THE SECURITY LINE — CALLBACK TO CHAPTER 13

LiveKit mints a signed access token, and Jitsi exposes a room URL with an optional token — always server-side, never from the browser. Access is gated by status: join only a booked appointment, in its window, once marked arrived or checked-in. On disconnect, flip to fulfilled. The exact token schema and mapping field are yours to confirm.

— DIAGRAM · RAG OVER CLINICAL DOCUMENTS

DocumentReference is the index entry — the bytes live in Binary.

Ingest rail

Enumerate, resolve, fetch, then decode by content type.

↓ `content.attachment.url`

↓ extract by `contentType`



Extract → chunk

PDF extraction, plain passthrough, or OCR for scans

↓ embed



Embed → retrieve

vectorize chunks into the index

— THE ATTACHMENT CARRIES THE PAYLOAD

Each `content.attachment` is an Attachment datatype: an `attachment.url` pointing at a Binary or external store, or inline base-64 data — and a `contentType`, say `application/pdf` or `text/plain`, that tells the pipeline how to decode it.

— EXTRACTION STAYS CONCEPTUAL

The steps are fixed — extract, chunk, embed, retrieve. The concrete PDF and OCR tooling is an implementation choice you bind to your own stack.

— DIALOGUE · GROUNDED CLINICAL Q&A

Ask the documents. Answer with a citation.



Clinical Q&A

• patient-scoped

What did the last cardiology note say about this patient's medication?

What the system does

Callback Chapter 18 — this is the tool a FHIR-over-MCP agent calls.

4

Cite the DocumentReference

so a clinician can verify

— WALKTHROUGH · ONE LOOP, END TO END

The calendar produces the encounter. The encounter feeds the next.

The full patient journey

Scheduling and RAG are two halves of one loop.



3

Fulfilled → note

the clinician's note persists as a DocumentReference



4

Index → grounded Q&A

at the next visit, the answer cites the note we just watched get booked

— THE PAYOFF

The calendar produces the encounter, the encounter produces the document, and the document feeds the retrieval that informs the next encounter.

— TWO FHIR RESOURCES, TWO BOLTED-ON LAYERS

● Appointment

● DocumentReference

● Video room

● RAG index

— CONCEPT · PRODUCTION NOTES

Four habits to carry into production.

— ATOMICITY

Book the Appointment and busy-out the Slot in a single transaction Bundle so a race can never double-book.

— SCOPE DISCIPLINE

RAG retrieval is patient and tenant scoped — partition the index the way you partition the data.

— TOKEN DISCIPLINE

The video-room token is minted server-side only, gated by appointment status, never handed out from the browser.

— CITATION DISCIPLINE

Every generated answer carries its source DocumentReference so a clinician can open the original and verify.

● **Confirm against your deployment:**

● **\$find vs Slot?status=free**

● **room-mapping field**

● **extraction stack**

— RECAP · TRY IT, THEN THE BRIDGE

Book a visit, index its note — then pay the audit bill.

— TRY IT · CALENDAR

Search free Slots for a PractitionerRole, POST a booked Appointment in a transaction Bundle that also flips the chosen Slot to busy, and derive a deterministic video room name from the appointment id.

— TRY IT · RAG INGEST

Enumerate a patient's DocumentReferences, resolve content.attachment.url, fetch the Binary, then outline extract, chunk, embed, retrieve — with a DocumentReference citation on every answer.

— RECAP

● Quartet

● Flow

● Video

● RAG

— NEXT · CHAPTER 20

Everything we built creates obligations we have not paid: who joined the room, which documents entered the context, on what consent. Next: HIPAA, GDPR, the EHDS regulation, and the AuditEvent that records all of it.

— CHAPTER 20 · COMPLIANCE & AUDIT · THE HOOK

The moment the data is real, the law is watching.

Last chapter we built a working telemedicine platform — appointments, clinical documents, a searchable corpus of patient records. Compliance is not a feature you bolt on at the end. It is a property the platform must hold, continuously, on every single request. FHIR hands you two records and one gate.

— THE PROMISE

AuditEvent — who touched it · Provenance — where it came from · Consent — whether it may be touched at all

Three FHIR primitives, pulled in different directions by three regulatory regimes. The engineer's job is to hold all three at once.

What you will be able to do

— CONCEPT · AUDITEVENT — THE SECURITY AUDIT TRAIL

AuditEvent is the camera in the room.

It answers who did what, when, and where — and it is written by the responding server, the one that handled the request. It builds on the older IHE ATNA audit standard, so it is not new ground.



Ava & Marcus

• Discussing AuditEvent

How do we prove who actually looked at a record?

Core fields

What every AuditEvent records

4

outcome · entity

success or refusal · resources touched

5

purposeOfEvent

why the access happened

— CONCEPT · PROVENANCE — DATA LINEAGE

Provenance answers where the data came from.

AuditEvent's sibling, and people confuse the two. Provenance is lineage — not who accessed the system, but who produced the data. It is prepared by the initiating application, the one that actually performed the create or update.



Marcus & Ava

• Discussing Provenance

What sits on a Provenance record — and who writes it?

Lineage fields

What every Provenance records

4 occurred
optional time the activity happened

5 signature
non-repudiation, when needed

— REGIME 1 · HIPAA — THE AUDIT-CONTROLS MANDATE

The audit trail is not paperwork. It is a legal control.

The HIPAA Security Rule mandates audit controls — hardware, software, and procedural mechanisms that record and examine activity in any system holding electronic protected health information, ePHI. The FHIR-native answer writes those records for you.

● THE MANDATE

Audit controls

Mechanisms that record and examine activity in systems holding ePHI. A requirement of the HIPAA Security Rule.

● THE RESOURCE

AuditEvent

The FHIR-native record of who did what, when, and where — the exact shape the Rule asks for.

● THE MECHANISM

Emit-on-access

An interceptor that emits one AuditEvent on every read and every write. The records write themselves.

● ON TOP

Retention · tamper-evidence

Operational layers you add over the resource — the spec already gives you the record itself.

● No audit, no compliance

● AuditEvent + emit-on-access interceptor satisfies the Security Rule

— REGIME 2 · GDPR — ACCESS VS ERASURE

Two articles. One easy. One collides with the audit trail.

Make the data unreadable, without destroying the proof that it existed. That is the heart of the tension between a patient's right to be forgotten and an append-only audit and Provenance model.

— ARTICLE 15 · RIGHT OF ACCESS

Maps cleanly to FHIR

The right of access lands directly on a FHIR read and the \$everything operation. The patient asks, you return their record. Easy.

- read + \$everything

— ARTICLE 17 · RIGHT TO ERASURE

Collides head-on

Erasure conflicts with append-only audit and Provenance, and with clinical-record retention duties. The pattern is logical-delete or de-identification — not a physical purge.

- de-identify · preserve the trail

- Audit logs are a lawful-basis exception · clinical safety often overrides erasure

— REGIME 3 · EU AI ACT — HIGH-RISK CLINICAL AI

The audit story and the AI story are the same story.

The EU AI Act classifies AI used in medical devices and clinical decision-making as high-risk — and high-risk triggers real duties. The LLM agent from Chapter 18, reading and writing FHIR over MCP, is not exempt: if it influences a clinical decision, it can fall squarely in scope.

● DUTY

Logging

AuditEvent and Provenance become the technical evidence trail the Act's logging mandate demands.

● DUTY

Human oversight

A person must remain able to intervene in decisions the system influences.

● DUTY

Transparency

Users must know an AI system is in the loop and how it reaches its outputs.

● DUTY

Risk management

A continuous process across the system's lifecycle, not a one-time check.

Phased — dates still settling

● Obligations phased broadly through twenty twenty-five to twenty twenty-seven

● High-risk applies broadly from around twenty twenty-six

● Ch 18 FHIR-over-MCP agents land in scope

— FRAMEWORK ANCHOR · CONSENTINTERCEPTOR 403-ABORT

Deny-by-default. The patient's own voice can override a valid role.

Audit tells us who looked — consent decides who may look at all. A Consent resource records a patient's policy: permit or deny, with a scope, a provision, the actor, the action, and the purpose.



Marcus & Ava

• Discussing the consent gate

How is this different from the RBAC from Chapter 17?

The decision

Evaluated at the request seam, before storage

4 Permit → storage runs
the request proceeds

5 Deny → abort 403
short-circuit before storage

— SYNTHESIS · ONE REQUEST, END TO END

Permit or deny, the camera never stops recording.

Follow a single incoming read across every gate. Both paths leave a trace — a refused access is itself audit-worthy.



Ava & Marcus

● Tracing one request

Walk it through — what happens if both gates pass, and what happens if consent denies?

The whole topic, one canvas

From incoming read to recorded outcome

4

PERMIT → AuditEvent + Provenance

storage runs · who/what/when · lineage on writes

5

DENY → 403 · AuditEvent still logged

the denied attempt is recorded too

— RECAP & BRIDGE · TRY IT YOURSELF

Two records, two questions — read them side by side.

Put a sample AuditEvent and a sample Provenance next to each other, label the fields, and say in one sentence which question each answers.

— AUDITEVENT · WHO TOUCHED THE SYSTEM

type · agent · source · entity · outcome

The security audit trail — who did what, when, and where. Written by the responding server.

— PROVENANCE · WHO PRODUCED THE DATA

target · agent · entity · activity

The data-lineage record — where this resource came from. Prepared by the initiating application.

Recap

● AuditEvent

● Provenance

● HIPAA audit mandate

● GDPR access vs erasure

● EU AI Act high-risk

● ConsentInterceptor 403

— CAPSTONE · THE SCARS

Every chapter carried a warning. Here they are in one place.

This chapter teaches nothing new. It gathers the production warnings from twenty chapters into one honest post-mortem, so you recognise them on day one instead of at two in the morning.

● Every chapter had a warning

● Here they are in one place

● How to not get burned

— SCAR · R4 SUBSCRIPTIONS — CALLBACK TO CHAPTER 12



Ava & Marcus

• War story

It worked perfectly in the demo. What actually breaks when an R4 Subscription is real?

Why it stays silent

No error. No log. Just a missing notification.

3

In-memory match cache

not shared across nodes — multi-node drop without a broker

4

The cure exists

R5-style SubscriptionTopic backport — Chapter 12 covered it

— SCAR · PARTITION FOOTGUNS — CALLBACK TO CHAPTER 11

Two leaks that never throw and never log.

The most dangerous property a bug can have is silence. Neither of these errors. They just leak rows — and a single-tenant test passes while the data quietly bleeds.

— FOOTGUN 1 · DEFAULT-PARTITION LEAK

Forgot to assign a partition

A missed assignment — or a tenant-registry miss that falls through instead of being rejected — lands the resource in DEFAULT, visible to anyone whose query is not partition-scoped.

— FOOTGUN 2 · CROSS-TENANT READ HOOK

A Pointcut that returns too much

A READ Pointcut that returns more than the caller's own partition — or returns null or all — silently exposes every tenant's data at once.

- **Catch it only with a two-tenant isolation test — A cannot see B**

- **HAPI 8.x default / null-partition semantics are version-sensitive — verify against your pinned tag**

— SCAR · SEARCH BLOWUPS — CALLBACK TO CHAPTER 9

Search power is a footgun unless you bound it.

Two convenience operations scale badly the moment they run unbounded. Neither is exotic — both are the default behaviour of features you reach for on instinct.

— BLOWUP · `_INCLUDE=*`

The bundle that ate the server

A star include or `:iterate` recursion pulls referenced resources in until the bundle is enormous and the server is on its knees. Name your includes explicitly; cap with `_count`.

\$ever

COST · WHOLE- COMPARTMENT SWEEP

Patient/\$everything walks every resource that references the patient — expensive by nature. Bound it with `_count` and paging.

- Custom SearchParameter does NOT retroactively index existing rows — reindex or get zero results

— SCAR · PROFILE DRIFT — CALLBACK TO CHAPTER 8

A profile is a contract. Contracts have version numbers.

The quiet scar — no boot failure, no leaked data, just a contract that moved under you. Profiles are versioned, and the version moves while your tooling stays still.

— WHEN THE TOOLING WAS WRITTEN

US Core at v7

A lot of tooling and curricula were written against US Core v7. Must-support lists and bindings reflect that version's understanding of conformant.

● Pin the US Core version

— CURRENT IN 2026

US Core 8.0.1 · STU 8

If the validator is pinned to one version and the IG package loaded is at another, they silently disagree on what conformant means — or the validator goes quiet and waves things through.

● Validator and IG package must agree

— SCAR · HAPI 8.X CONFIG TRAPS — CALLBACK TO CHAPTER 7

Three boot-time scars — pattern and symptom, not config keys.

Every one of these is an it-worked-on-my-machine failure, and every one is prevented by pinning the exact tag and verifying against it.

1

EMAILSENDER MANDATORY

Enable the email subscription channel and autoconfig demands an email-sender bean. Symptom: the server will not boot, the context fails to start.

2

PARTITIONING BOOLEAN FLIP

A partitioning boolean's meaning shifted across 8.x minors. Symptom: same yaml, a minor bump, different behaviour — a silent semantic flip.

3

SPRING.PROFILES PRECEDENCE

Env-vs-yaml activation precedence means the dev config you were sure was active quietly is not. Symptom: wrong datasource or flags applied without complaint.

- **Verify exact bean name, property paths, and shipped profile names against your pinned 8.x tag**

— SCAR · SSE DELIVERY + BACKPRESSURE — CALLBACK TO CHAPTER 15

The live update can drop. Gap-detection is the contract.

Honest about the distributed-systems reality under the live-update loop — these are patterns, not a specific package's verified API.

— FAILURE · THE HONEST REALITY

Delivery is at-least-once

SSE delivery is at-least-once, not exactly-once. A slow client plus a capped ring buffer — the MAXLEN you set — drops events off the back. Proxies and CDNs love to buffer and can stall the stream completely.

● At-least-once · capped MAXLEN · proxy buffering

— MITIGATION · THE CONTRACT

Idempotent + gap-refetch

Handlers must be idempotent — re-applying patient.updated is safe by design. On a detected gap (Last-Event-ID older than the buffer head) the client falls back to a full refetch. Disable buffering on the stream route.

● Idempotent handlers · gap-detection refetch · no-buffer route

— EARNED WISDOM · THE META-LESSON

The bugs were never exotic — they were the boring three you skipped.

Every scar in this chapter collapses into three habits. Exotic bugs are rare; these three are the ones that actually ship.

● PILLAR 1

Test against real data

Demo data hides the partition leak, the include blowup, the multi-node subscription drop. One tenant, ten rows, one node — none fire. Two tenants, a million rows, two nodes — all do.

● PILLAR 2

Pin every version

The 8.x config flip and the US Core drift are both version-skew. Pin the HAPI tag, pin the IG package, pin them in the lockfile — not in your head.

● PILLAR 3

Validate at the boundaries

Reject non-conformant writes at the interceptor. Reject an unknown tenant instead of defaulting it. Validate every SSE payload against the shared schema.

● Three habits map 1:1 onto the seven scars above

— EARNED WISDOM · PRE-PRODUCTION CHECKLIST

The questions to answer before you tell anyone it is done.

— VERSIONS · SLIDE 06 & 05

Pinned the exact HAPI 8.x image tag.
Verified config keys and the spring profile against THAT tag.
Pinned US Core / IG package version — validator and IG agree.

— TENANCY · SLIDE 03

Ran a two-tenant isolation test — tenant A cannot see tenant B.
Unknown or absent tenant header is REJECTED, never defaulted.

— SEARCH · SLIDE 04

Every `_include` is explicit — no `_include=*`; \$everything bounded with `_count`.
Reindexed after adding any custom `SearchParameter`.

— EVENTS · SLIDE 02 & 07

Shared broker if more than one HAPI node.
SSE handlers idempotent with gap-detection refetch.

● Check every box and the scars above are scars you will never earn

— EARNED WISDOM · WRAP

The spec is generous. The implementations are subtle.

The scars are how you learn the difference between the two. Every pitfall here came from a real codebase — yours will produce its own, and now you will recognise the shape of them before they cost you.

● [Chapter ZZ — course wrap-up & where to go next](#)

● [Demo · reading list · contact](#)

— COURSE COMPLETE • THE JOURNEY, AND WHAT TO BUILD NEXT

You can ship FHIR R4 now.

You started not knowing what a Resource was. Now you can stand one up, scale it across tenants, secure it, expose it to agents, and trust it in production. That is the whole arc of this course.

● Resource modelling to multi-tenant HAPI

● SMART on FHIR auth, plus agents over MCP

● You know where the pitfalls hide

Build this next

— THANKS FOR WATCHING

Go build something real with FHIR.

— WHERE TO GO NEXT

Built by Dudoxx

The studio behind production FHIR systems in Hamburg and Paris. See this stack running live, and find the next chapters.

dudoxx.com

wahid@acceleate.com

— BUILT & PRESENTED BY

Walid Boudabbous

CTO Dudoxx UG · CEO Acceleate Consulting OU
28 years across AI, healthcare, and fintech.
Thank you for spending this time with us — we will see you in the next course.