— 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 */ } }
```

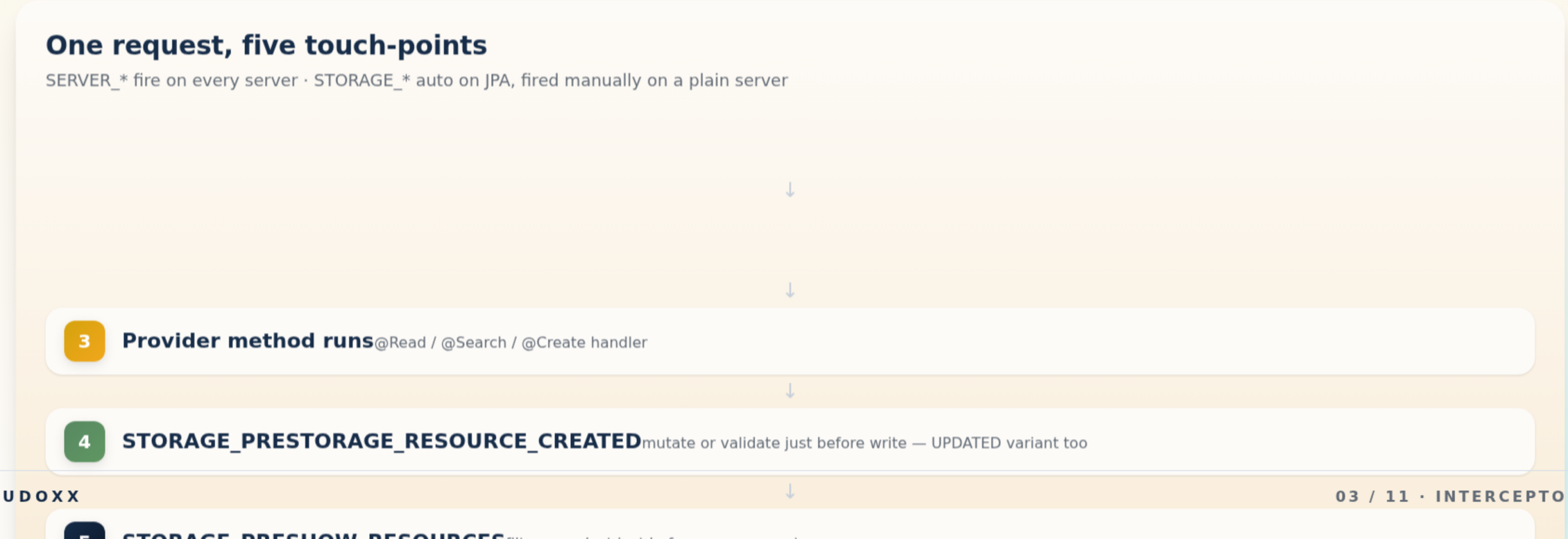
The framework injects context by parameter type

- 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.



— 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.

— 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