Scala 3 Domain Design & Typelevel Stack Cookbook

A cookbook for intermediate Scala 3 developers: domain modeling and copy-pasteable backend recipes with cats, cats-effect, http4s and fs2.

David Flores

Table of Contents

Introduction

What this book is not

This is not a book that will make you a Scala 3 master, a category theory scholar, or someone who can explain the Yoneda lemma at a dinner party. There are excellent books for that, and if that's what you're after, close this one and go read *Functional Programming in Scala* instead — no hard feelings.

What this book is

This is a cookbook. You have a problem — "I need to model a domain where illegal states shouldn't compile," "I need to stand up an http4s server with JWT auth by Thursday" — and you want a working answer you can adapt, not a semester of theory before you get there. Every chapter in this book is built around that shape: here's the problem, here's the tool, here's a real example you can lift into your own codebase, here's where it bites people.

You'll come out the other side with working knowledge you can use on Monday morning, not a mental model you need six more books to complete. If that working knowledge later makes you curious about *why* a type class desugars the way it does, or *why* **`IO`** is a free monad in spirit — great, that curiosity is the good kind, and there are better books than this one waiting for it. This one just isn't trying to be that book, on purpose.

Who this is for

Someone who already writes Scala 3 day to day, knows what a `for` comprehension is, has heard of cats and cats-effect even if they haven't used them in anger yet, and is tired of every blog post either being a "hello world" that doesn't survive contact with a real codebase, or a research paper that assumes you already know the answer. Intermediate, in other words — the most underserved level in programming books, somehow, despite being where most working developers actually live.

The "stack level" problem this book keeps coming back to

If you've touched cats, cats-effect, and http4s together, you've probably hit the same wall: each library's docs are fine *in isolation*, but nobody tells you how the pieces actually stack. Is **`EitherT`** a cats-effect thing or a cats thing? Why does **`HttpRoutes[IO]`** return an **`OptionT`**? Why does adding one middleware change the type signature of everything downstream of it? Why does the compiler error you get from a missing given sometimes point at a completely unrelated line?

Part 2 of this book is named **"TypeStack Cookbook"** specifically because of this — every recipe includes not just the working code, but a note on *which layer of the stack* each piece belongs to (cats' data types, cats-effect's runtime, http4s's request/response plumbing) so that when something breaks, you know which library's mental model to reach for instead of guessing.

How to read this book

Cover to cover if you like, but it's built to be dipped into. Part 1 chapters are mostly self-contained, though they build on each other loosely (Chapter 3's smart constructors lean on Chapter 1's opaque types; Chapter 5's state machines lean on Chapter 2's ADTs). Part 2 recipes assume you've at least skimmed Recipe 1 (the server bootstrap) and can otherwise be read in whatever order matches the problem on your desk today.

Every recipe ends with a **Cheat Sheet** — the two-minute version, for when you're back here in three months trying to remember the shape of a recipe you already read once and don't have time to re-read the whole chapter.

Let's get into it.

Part 1 — Scala 3 Domain Design

What this part covers

Six chapters, each centered on one Scala 3 feature and one question: *what production bug does this feature prevent from ever being written?* Not "what does the syntax do" — you can get that from the Scala docs in thirty seconds — but "what does this buy you that a plain case class and a `String` don't."

The through-line across all six chapters is the same idea approached from different angles: **push correctness to compile time, and push validation to the boundary of your system**, so that everything inside your domain logic can trust the types it's holding instead of re-checking them.

- **Opaque types** stop you from mixing up primitives that happen to share a runtime representation (an **`AccountId`** is not a **`MerchantId`**, even though both are **`Strings`** underneath).
- **ADTs & enums** stop you from constructing states that shouldn't exist (a shipped order with no **`paidAt`**).
- **Smart constructors** stop invalid *values* from ever becoming an instance of a type in the first place (no Email without a validated address behind it).
- **Type classes** let you add behavior to types — including types you don't own — without touching an inheritance hierarchy.
- **State machines** stop you from calling the wrong transition on the wrong state (shipping a **`Pending`** order).
- **Domain errors as data** stop failure modes from hiding outside a function's signature.

By the end of Part 1, you should be able to look at a domain model with a lot of **`Option[X]`** fields, boolean flags, and stringly-typed status columns, and see exactly which of these six tools fixes it.

What this part deliberately skips

Category theory terminology beyond what you need to read cats' type signatures. Laws and proofs. Performance micro-benchmarking of opaque types vs case classes (short version: opaque types win, by design — see Chapter 1 if you want the "why"). This part is about applying features to real domain problems, not about the language design decisions behind them.

Chapter 1 — Opaque Types: Type Safety Without the Runtime Tax

The problem

At some point in every codebase, someone writes a function like this:

```
def transferMoney(fromAccount: String, toAccount: String, amount:
BigDecimal): F[Unit]
```

And at some later point, someone calls it like this:

```
transferMoney(toAccount, fromAccount, amount) // oops
```

The compiler says nothing, because **`String`** is **`String`** is **`String`**. You find out about the bug when Karen from accounting emails you, and by then it's a Slack incident, not a compiler error.

The classic fix is a wrapper case class:

```
final case class AccountId(value: String)
```

This works, but it's not free — every **`AccountId`** is a heap-allocated object wrapping a **`String`**. In a hot path (serialization, high-throughput APIs, anything iterating over millions of rows) that allocation adds up, and you'll see it in a flight recorder before you see it in a code review.

The Scala 3 feature

Opaque types give you a *new type at compile time* that is *literally the same type at runtime* — zero wrapper, zero allocation, zero excuses.

```
opaque type AccountId = String

object AccountId:
  def apply(raw: String): AccountId = raw
  extension (id: AccountId) def value: String = id
```

Outside the companion object, **`AccountId`** is a distinct type — you cannot pass a raw **`String`** where an **`AccountId`** is expected, and you cannot accidentally swap an **`AccountId`** for a **`MerchantId`** even though both are **`String`** under the hood. Inside the companion object, the compiler still knows it's a **`String`**, so you get to write normal string operations without fighting yourself.

A real domain example

Let's model a payments domain with a few primitives that are all, annoyingly, strings or longs underneath:

```
opaque type AccountId = String
opaque type MerchantId = String
opaque type Cents = Long

object AccountId:
```

```
  def apply(raw: String): AccountId = raw

object MerchantId:
  def apply(raw: String): MerchantId = raw

object Cents:
  def apply(value: Long): Cents = value
  extension (c: Cents)
    def +(other: Cents): Cents = (c: Long) + (other: Long)
    def toDollars: BigDecimal = BigDecimal(c: Long) / 100
```

Now this signature actually protects you:

```
def transferFunds(from: AccountId, to: AccountId, amount: Cents):
IO[Unit] =
  IO.println(s"Moving ${amount.toDollars} from $from to $to")
```

Try calling **`transferFunds(merchantId, accountId, cents)`** — it won't compile. That's the whole point. You moved a footgun from "runtime bug found by a customer" to "red squiggly line in your editor."

Bonus: opaque types over non-primitives

Opaque types aren't limited to wrapping String/Long. You can wrap refined versions of existing types too, which pairs nicely with smart constructors (Chapter 3):

```
opaque type Email = String

object Email:
  def parse(raw: String): Either[String, Email] =
    if raw.matches("^[^@\\s]+@[^@\\s]+\\.[^@\\s]+$") then Right(raw)
    else Left(s"'$raw' is not a valid email")

  extension (e: Email) def value: String = e
```

Once you have an Email, you *know* it's valid — there's no code path that produces one without going through parse. That guarantee is worth more than the few characters you saved by not writing a case class.

Gotchas

- **Opaque types have no runtime representation of their own**, which means no runtime type checks, no **`isInstanceOf[AccountId]`**, and — this one bites people — no free **`Show`/`Encoder`/`Decoder`** instances. You need to derive or write those explicitly for the opaque type, not for the underlying type.
- **They're only opaque within the file/module where they're not imported through the companion.** Scoping opaque types across multiple files needs a bit of care; keep them in one file per domain concept and export the companion object, not the raw alias.
- **Don't use them for structural validation with many fields.** If a value needs several related fields, you want a case class (possibly built from several opaque types), not one giant opaque type with a packed encoding.

- **Circe/http4s codecs**: you'll write **`given Encoder[AccountId] = Encoder.encodeString.contramap(_.value)`** a lot. It's a one-liner, but you'll write it for every opaque type you introduce — budget for it.

Takeaway

Reach for opaque types whenever you catch yourself passing "just a String" or "just a Long" around that actually means something specific in your domain (an ID, a currency amount, an email, a version string). You get compile-time safety with the exact runtime shape of the primitive — there's very little reason *not* to do this once it's in your muscle memory.

Cheat Sheet

- **Use when:** a primitive (**`String`, `Long`, `Int`**) actually means something specific in your domain (an ID, a currency amount, an email) and you want zero runtime overhead over the primitive.
- **Skeleton:**

```
opaque type X = Underlying
object X:
  def apply(raw: Underlying): X = raw          // or parse(...) returning Either
  extension (x: X) def value: Underlying = x
```

- **Don't use when:** the value needs several related fields — reach for a case class (possibly built from several opaque types) instead.
- **Remember:** write `Encoder`/`Decoder`/`Show` instances explicitly — they don't come along for free from the underlying type.

Chapter 2 — ADTs & Enums: Making Illegal States a Compile Error

The problem

You've seen this shape a thousand times, probably in your own codebase from two years ago:

```
final case class Order(
      status: String,    // "pending" | "paid" | "shipped" | "cancelled"
      paidAt: Option[Instant],
      shippedAt: Option[Instant],
      cancelReason: Option[String]
)
```

Nothing stops you from constructing **`Order("pending", Some(now), None, Some("customer changed their mind"))`** — a paid, cancelled order that was somehow also never shipped and never actually cancelled properly. This compiles fine and blows up in a report six months later when finance asks why revenue doesn't reconcile.

The Scala 3 feature

Algebraic Data Types (sealed hierarchies) let you say "an **`Order`** is *exactly one* of these shapes, and each shape only carries the fields that make sense for it." Scala 3's **`enum`** syntax makes this almost pleasant to write, compared to the **`sealed trait` + `case class`** boilerplate of Scala 2.

```
enum Order:
  case Pending(id: OrderId, items: List[LineItem])
  case Paid(id: OrderId, items: List[LineItem], paidAt: Instant)
  case Shipped(id: OrderId, items: List[LineItem], paidAt: Instant,
shippedAt: Instant)
  case Cancelled(id: OrderId, items: List[LineItem], reason: String)
```

Now **`Order.Cancelled`** simply cannot carry a **`shippedAt`** — there's no field for it. There is no invalid state to construct because the invalid states don't exist as types.

Pattern matching, exhaustively

The other half of the deal is exhaustiveness checking:

```
def summarize(order: Order): String = order match
  case Order.Pending(id, items)                => s"Order $id: waiting
for payment"
  case Order.Paid(id, _, paidAt)               => s"Order $id: paid at
$paidAt"
  case Order.Shipped(id, _, _, shippedAt)       => s"Order $id: shipped
at $shippedAt"
  case Order.Cancelled(id, _, reason)           => s"Order $id:
cancelled ($reason)"
```

Add a new case to the enum next month (say, Refunded) and this match becomes a compiler warning (or error, if you set **`-Xfatal-warnings`** which — do it, seriously). You find out about the missing branch during sbt compile, not during an on-call page.

A real domain example: parsing a webhook payload

Payment providers love sending you a JSON blob with a type field and a different shape per type. This is basically an ADT with extra steps:

```
enum StripeEvent:
  case PaymentSucceeded(paymentIntentId: String, amountCents: Long)
  case PaymentFailed(paymentIntentId: String, reason: String)
  case ChargeRefunded(chargeId: String, amountCents: Long)
  case Unhandled(rawType: String)

object StripeEvent:
  def fromJson(json: Json): StripeEvent =
    json.hcursor.get[String]("type").getOrElse("unknown") match
      case "payment_intent.succeeded" =>
        PaymentSucceeded(
          json.hcursor.downField("data").get[String]("id").getOrElse(""),

json.hcursor.downField("data").get[Long]("amount").getOrElse(0L)
        )
      case "payment_intent.payment_failed" =>
        PaymentFailed(
          json.hcursor.downField("data").get[String]("id").getOrElse(""),

json.hcursor.downField("data").get[String]("failure_message").getOrElse("
unknown")
        )
      case "charge.refunded" =>
        ChargeRefunded(
          json.hcursor.downField("data").get[String]("id").getOrElse(""),

json.hcursor.downField("data").get[Long]("amount_refunded").getOrElse(0L)
        )
      case other => Unhandled(other)
```

The **`Unhandled`** case is doing real work here: Stripe *will* add new event types over time, and your webhook handler needs a graceful "I don't know what this is, log it and move on" path instead of throwing a **`MatchError`** at 3am.

Plain enum for simple closed sets

Not everything needs case classes attached. For genuinely simple closed sets, plain enum cases are the right tool and read closer to a Java enum:

```
enum Currency:
  case USD, EUR, MXN, GBP

enum SubscriptionTier derives CanEqual:
  case Free, Pro, Enterprise
```

Gotchas

- **`sealed trait + case classes is still the right call`** when your cases need to share a nontrivial method implementation, or when the hierarchy is genuinely open for extension in ways enum doesn't model well (like a plugin architecture). enum is sugar over exactly that pattern, so reach for the plain version when the sugar gets in the way.
- **Exhaustiveness checking needs sealed** (which enum gives you for free) — if you're still writing trait without sealed anywhere in your domain layer, stop, because you're opting out of the entire point of this chapter.
- **`derives CanEqual`** matters more than people think — without it, Scala 3's multiversal equality won't stop you from writing **`Currency.USD == SubscriptionTier.Free`** and getting false silently instead of a compile error.
- **Don't let cases balloon.** If your ADT has 15 cases each with 8 fields, that's usually a sign you're modeling several different concepts as one enum. Split it.

Takeaway

Every time you write an **`Option[X]`** field that's only sometimes populated depending on the value of *another* field, that's an ADT trying to escape your case class. Let it out. The compiler will do more of your testing for you than any unit test suite you'll write for this logic.

Cheat Sheet

- **Use when:** a value is genuinely "exactly one of N shapes," and different shapes need different fields (order status, event types, parsed webhook payloads).
- **Skeleton:**

```
enum X:
  case A(fieldsForA: Type)
  case B(fieldsForB: Type)
```

- **Get exhaustiveness checking for free** by matching on the sealed hierarchy — add **`-Xfatal-warnings`** so a missing case is a build failure, not a warning nobody reads.
- **Don't use when:** cases need to share nontrivial method logic or the hierarchy needs to stay open for external extension — plain sealed trait gives you more room there.
- **Remember: `derives CanEqual`** if you want the compiler to catch comparisons between unrelated enum types.