ΑΠΟ MID-LEVEL APP ΣΕ PRODUCTION SYSTEM

Sinatra

μέχρι Τέλους

Rack, HTTP, architecture, concurrency και operations χωρίς framework magic.

30

Από route σε ολόκληρο σύστημα

ΚΕΦΑΛΑΙΑ · SINATRA 4.2.1 · RUBY 3.4.7

ΠΡΙΝ ΑΝΟΙΞΟΥΜΕ ΤΟ REQUEST

Το single-file app τελείωσε. Τώρα χτίζουμε το σύστημα.

Το πρώτο Sinatra route είναι σχεδόν υπερβολικά εύκολο. Αυτή είναι η δύναμη του framework και μαζί η παγίδα του. Όταν η εφαρμογή μεγαλώσει, το δύσκολο κομμάτι δεν είναι να προσθέσεις άλλο ένα `get`. Είναι να ξέρεις ποιος κρατά state, πότε ένα response έχει πραγματικά τελειώσει, ποια middleware βλέπουν μια exception, τι εμπιστεύεσαι από proxy headers και πόση δουλειά χωρά σε κάθε Puma process.

Το βιβλίο ξεκινά από mid-level. Δεν εξηγεί τι είναι Ruby block ή HTTP verb. Δείχνει πώς το Sinatra μετατρέπει class-level DSL σε compiled routes, πώς ένα Rack env γίνεται request instance και πώς το body συνεχίζει να ζει αφού το route έχει επιστρέψει. Μετά περνά από security, persistence, distributed effects, streaming, concurrency και production releases.

Πώς διαβάζεται

Τα κεφάλαια 1 έως 5 χτίζουν τον χάρτη της εφαρμογής. Τα 6 έως 11 ακολουθούν το request μέσα από routing, control flow και errors. Τα 12 έως 18 καλύπτουν rendering, HTTP και security. Τα 19 έως 23 σχεδιάζουν application και data boundaries. Τα 24 έως 30 κατεβαίνουν στο runtime και τελειώνουν σε ένα production correctness packet.

Κάθε κεφάλαιο έχει `SOURCE` `TRACE` και `PROBE`. Το πρώτο δείχνει την πραγματική διαδρομή στην έκδοση αναφοράς. Το δεύτερο μετατρέπει το mental model σε μικρό πείραμα. Τα internals δεν παρουσιάζονται ως API. Χρησιμοποιούνται για να εξηγήσουν observable behavior και για να βρεις ξανά την αλήθεια μετά από upgrade.

Έκδοση αναφοράς

Η έκδοση 1.0 ελέγχθηκε τον Σεπτέμβριο του 2026 με Sinatra 4.2.1, Ruby 3.4.7, Rack 3.2.7, Rack::Protection 4.2.1, Tilt 2.7.0 και Puma 8.0.2. Όπου μια συμπεριφορά εξαρτάται από server, proxy, storage adapter ή browser, επισημαίνεται αντί να αποδίδεται στο Sinatra.

Το βιβλίο δεν γεμίζει το Sinatra με abstractions μέχρι να μοιάσει σε άλλο framework. Κρατά το system explicit και προσθέτει structure μόνο όταν υπάρχει συγκεκριμένο ownership ή failure mode.

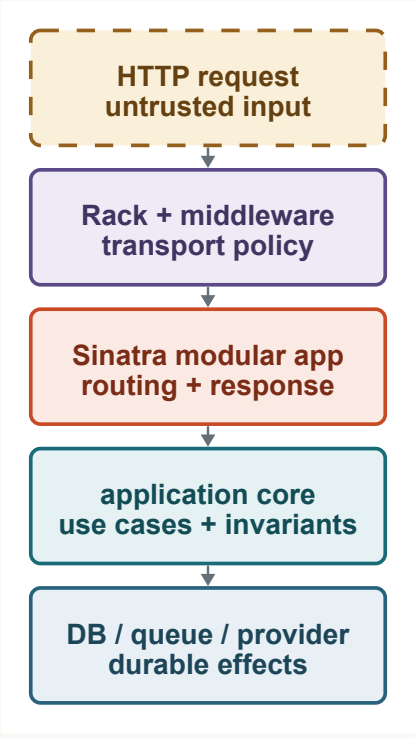
Ο ΧΑΡΤΗΣ ΤΟΥ ΒΙΒΛΙΟΥ

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1

Πέρα από το single-file demo

Το Sinatra μένει μικρό μόνο όταν το application model είναι καθαρό. Το μέγεθος του αρχείου δεν λέει τίποτα για το μέγεθος του συστήματος.



Σχήμα 1.1 · Από το HTTP boundary στα durable effects.

Ένας mid-level Sinatra developer ξέρει ήδη να γράφει routes, filters, helpers και templates. Το επόμενο βήμα δεν είναι να μάθει άλλα δέκα DSL methods. Είναι να σταματήσει να θεωρεί το route ολόκληρη την εφαρμογή. Το route είναι ένας transport adapter. Παραλαμβάνει HTTP input, το μετατρέπει σε command, καλεί application code και μεταφράζει το outcome σε HTTP response.

Αυτή η διάκριση κρατά το Sinatra εκεί που είναι δυνατό: λεπτό, άμεσο και προβλέψιμο. Δεν χρειάζεται να χτίσεις ένα μικρό Rails μέσα του. Χρειάζεται να δώσεις owner σε κάθε απόφαση. Το middleware κατέχει transport policy. Το route κατέχει HTTP mapping. Ένα use case κατέχει το workflow. Η database κατέχει durable constraints. Ένας adapter κατέχει την επικοινωνία με τρίτο σύστημα.

Η ελάχιστη modular βάση

Για εφαρμογή που θα ζήσει περισσότερο από ένα script, ξεκίνα από `Sinatra::Base`. Η class είναι Rack endpoint και μπορεί να δοκιμαστεί ή να τοποθετηθεί σε stack χωρίς να μολύνει το top-level namespace.

```ruby
require "sinatra/base"

module Seatly
  class App < Sinatra::Base
    configure do
      set :show_exceptions, false
      set :raise_errors, false
      set :default_content_type, "application/json"
    end

    get "/health" do
      content_type :json
      JSON.generate(status: "ok")
    end
  end
end
```

Το `configure` εδώ είναι boot-time configuration. Δεν είναι request callback. Η `Seatly::App` κρατά routes, settings, filters και middleware ως class-level registries. Όταν έρθει request, το Sinatra χρησιμοποιεί request instance για τα `env`, `request`, `response`, `params` και τα instance variables του route. Αυτό το split θα μας απασχολήσει σε όλο το βιβλίο.

Το `config.ru` είναι το composition root του HTTP process:

```ruby
require_relative "lib/seatly/app"

run Seatly::App
```

Δεν βάζουμε business initialization κρυμμένο σε route. Φτιάχνουμε τα long-lived objects στο boot και τα περνάμε ρητά. Ένας απλός container μπορεί να είναι plain Ruby object, όχι dependency injection framework.

```ruby
module Seatly
  Container = Data.define(:reservations, :clock, :events)

  class App < Sinatra::Base
    def self.with(container:)
      Class.new(self).tap { |app| app.set :container, container }
    end

    post "/reservations" do
      command = CreateReservationInput.from_http(request)
      result = CreateReservation.call(command, deps: settings.container)
      present(result)
    end
  end
end
```

Το example δείχνει contract, όχι υποχρεωτικό folder layout. Το `from_http` επιτρέπεται να ξέρει JSON, headers και status mapping. Το `CreateReservation.call` δεν πρέπει να ξέρει Rack env ή Sinatra helpers. Έτσι μπορεί να εκτελεστεί από job, CLI ή test χωρίς fake request.

Τέσσερα όρια που πρέπει να φαίνονται

Το πρώτο είναι το **process boundary**. Class settings, database pools, HTTP clients και caches ζουν περισσότερο από ένα request. Πρέπει να ξέρεις αν είναι thread-safe και τι συμβαίνει μετά από fork.

Το δεύτερο είναι το **request boundary**. Params, current user, response headers και request id ανήκουν σε μία εκτέλεση. Δεν πρέπει να διαρρέουν σε class variables ή global registries.

Το τρίτο είναι το **transaction boundary**. Το ότι ένα route επέστρεψε 201 δεν αποδεικνύει ότι database write και εξωτερικό effect έγιναν atomically. Συνήθως δεν έγιναν. Το workflow χρειάζεται explicit commit, outbox ή reconciliation.

Το τέταρτο είναι το **protocol boundary**. Route paths, JSON shapes, cookies, webhook payloads και cache validators είναι deployed contracts. Άλλη έκδοση του client ή άλλος process μπορεί να τα χρησιμοποιεί όταν το νέο code έχει ήδη κυκλοφορήσει.

Structure μόνο για πραγματικό ownership

Ένα directory `services/` δεν δημιουργεί architecture. Αν όλες οι classes τραβούν `settings`, `request` και global clients, έχεις distributed route handler με περισσότερα filenames. Καλύτερο test είναι να γράψεις για κάθε object μία πρόταση: ποια απόφαση κατέχει και ποια effects επιτρέπεται να κάνει.

Το Sinatra δεν επιβάλλει model, ORM, job system ή component lifecycle. Αυτό είναι ελευθερία, αλλά σημαίνει ότι η εφαρμογή πρέπει να δηλώσει τα δικά της contracts. Κράτα το composition μικρό. Προτίμησε immutable configuration, interfaces που εκφράζονται με μικρά Ruby methods και adapters που μπορούν να αντικατασταθούν σε test χωρίς αλλαγή του domain code.

SOURCE TRACE
Η modular είσοδος περιγράφεται στο `README.md`, ενότητα `Sinatra::Base - Middleware, Libraries, and Modular Apps`. Στο tag `v4.2.1`, η Rack είσοδος είναι `Sinatra::Base.call` και `#call!` στο `lib/sinatra/base.rb`. Ακολούθησε `Base.build -> Rack::Builder -> dispatch!` για να δεις πού τελειώνει το composition και πού αρχίζει το request.

ΠΑΓΙΔΑ
Μη μετατρέπεις το `settings` σε service locator που διαβάζεται από παντού. Είναι χρήσιμο composition surface, αλλά αν κάθε object βρίσκει μόνο του τις dependencies, ownership και tests γίνονται αόρατα.

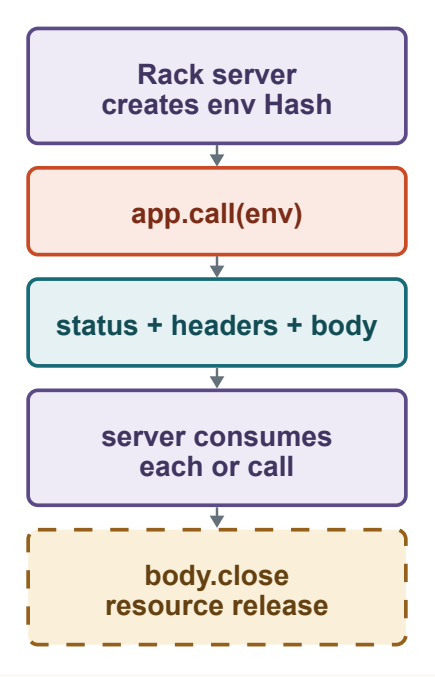
ΜΗΧΑΝΙΣΜΟΣ
Το route είναι HTTP adapter, όχι application layer. Η πιο χρήσιμη Sinatra architecture έχει μικρό transport shell και plain Ruby core με explicit dependencies και invariants.

PROBE
Πάρε ένα υπάρχον route με database write και provider call. Σχεδίασε τη διαδρομή request, parse, decision, transaction, provider, response. Σημείωσε τον owner και το failure result κάθε βέλους. Αν δεν μπορείς να πεις τι γίνεται όταν ο process πέσει σε κάθε σημείο, βρήκες το πρώτο πραγματικό refactor.

ΚΕΦΑΛΑΙΟ 2

Το Rack contract ως αρχιτεκτονικό όριο

Κάθε Sinatra request είναι μία κλήση `call(env)`. Ό,τι συμβαίνει πριν και μετά από αυτή την κλήση ανήκει στο Rack protocol, όχι στη DSL.



Σχήμα 2.1 · Το Rack response τελειώνει όταν καταναλωθεί και κλείσει το body.

Το Rack application contract είναι μικρό: ένα object που απαντά σε `call`, παίρνει ένα unfrozen Hash και επιστρέφει `[status, headers, body]`. Η μικρή επιφάνεια κρύβει ένα σημαντικό temporal detail. Η επιστροφή του tuple δεν σημαίνει ότι το response έχει σταλεί. Ο server πρέπει ακόμη να καταναλώσει το body και να το κλείσει.

Στο Rack 3.2, το status είναι Integer τουλάχιστον 100. Τα response header names είναι lowercase strings. Οι values είναι strings ή arrays από strings χωρίς NUL, CR ή LF. Το body απαντά σε `each` ή `call`. Αν απαντά και στα δύο, θεωρείται enumerable και ο server καλεί `each`. Ό,τι yield κάνει πρέπει να είναι String.

```ruby
class TinyRackApp
  def call(env)
    request_id = env.fetch("app.request_id")
    body = ["request=#{request_id}\n"]
    [200, { "content-type" => "text/plain" }, body]
  end
end
```

Το env είναι κοινός χώρος συνεργασίας middleware και endpoint. Τα CGI keys όπως `REQUEST_METHOD` δεν έχουν τελεία. Application-specific keys πρέπει να έχουν namespace, όπως `seatly.request_id`. Μην αποθηκεύεις εκεί object που θέλεις να ζήσει μετά το request χωρίς σαφές cleanup. Το env ταξιδεύει προς τα μέσα στη στοίβα και μπορεί να κρατήσει reference σε buffers, sessions, transactions ή trace state.

Το body είναι resource boundary

Ένα array από strings είναι απλό body. Ένα file iterator, streaming producer ή database cursor έχει lifecycle. Το Rack απαιτεί να κληθεί `close` όταν το body το υποστηρίζει, ακόμη και όταν η κατανάλωση αποτύχει ή όταν middleware αντικαταστήσει το body.

```ruby
class ClosingBody
  def initialize(rows, release:)
    @rows = rows
    @release = release
  end

  def each
    @rows.each { |row| yield JSON.generate(row) << "\n" }
  end

  def close
    @release.call
  end
end
```

Η `close` πρέπει να είναι ασφαλής αν κληθεί ξανά. Μην υποθέτεις ότι το happy path είναι η μόνη διαδρομή. Client disconnect, middleware error ή internal subrequest μπορούν να κλείσουν το body νωρίτερα. Αν το body κατέχει connection, file descriptor ή subscription, η idempotent cleanup είναι μέρος του contract.

Middleware που θέλει να μετρήσει όλη τη διάρκεια δεν πρέπει να καταναλώσει το body μέσα στο `call`. Πρέπει να το τυλίξει και να ολοκληρώσει τη μέτρηση στο `close`.

```ruby
require "rack/body_proxy"

class TotalDuration
  def initialize(app, emit:)
    @app = app
    @emit = emit
  end

  def call(env)
    started = Process.clock_gettime(Process::CLOCK_MONOTONIC)
    status, headers, body = @app.call(env)
    wrapped = Rack::BodyProxy.new(body) do
      elapsed = Process.clock_gettime(Process::CLOCK_MONOTONIC) - started
      @emit.call(status: status, seconds: elapsed)
    end
    [status, headers, wrapped]
  end
end
```

Αυτό μετρά μέχρι το close, όχι απλώς μέχρι να παραχθούν headers. Αν το `@app.call` σηκώσει exception πριν επιστρέψει body, χρειάζεται ξεχωριστό error path για να καταγραφεί η αποτυχία. Αν η εφαρμογή χρησιμοποιεί το optional `rack.response_finished`, ο server μπορεί να καλέσει registered callbacks με env, status, headers και error αφού επεξεργαστεί το response. Επειδή είναι optional, πρώτα επιβεβαίωσε ότι ο handler το υλοποιεί.

Internal requests δεν είναι δωρεάν

Το Sinatra επιτρέπει `call env.merge("PATH_INFO" => "/other")`. Αυτό είναι νέα Rack κλήση, όχι Ruby goto. Το returned body έχει lifecycle. Αν το κάνεις `body.map`, το καταναλώνεις, αλλά πρέπει ακόμη να φροντίσεις το close. Συνήθως είναι καθαρότερο να εξαγάγεις κοινό use case αντί να καλέσεις route από route.

Το `SCRIPT_NAME` και το `PATH_INFO` μαζί δείχνουν πού είναι mounted η εφαρμογή. URL generation που αγνοεί το mount point δουλεύει στη ρίζα και σπάει πίσω από `map "/admin"`. Το ίδιο ισχύει για scheme και authority πίσω από proxy. Αυτά είναι Rack request semantics και χρειάζονται trusted proxy configuration, όχι string concatenation.

Lint ως executable specification

Το `Rack::Lint` δεν αποδεικνύει business correctness. Πιάνει όμως malformed env, headers, status και body behavior. Βάλ' το σε tests γύρω από custom middleware και streaming bodies. Ένα middleware μπορεί να φαίνεται σωστό με array body και να χάνει `close` μόλις μπει file ή stream.

SOURCE TRACE

Το normative contract είναι το `SPEC.rdoc` του Rack 3.2.7. Για τη Sinatra μετάφραση δες `Sinatra::Base#call!`, `Sinatra::Response#body=` και `Sinatra::Response#finish` στο `lib/sinatra/base.rb`. Για HEAD behavior δες `Rack::Head#call` στο Rack 3.2.7, το οποίο κλείνει το original body και το αντικαθιστά με κενό array.

ΠΑΓΙΔΑ

Μην μετράς streaming endpoint μόνο γύρω από `app.call`. Θα καταγράψεις γρήγορο response ενώ ο worker, το socket και το body μένουν ενεργά για λεπτά.

ΜΗΧΑΝΙΣΜΟΣ

Το Rack tuple είναι αρχή του response phase, όχι το τέλος του. Headers commit, body consumption και body close είναι διαφορετικά χρονικά σημεία.

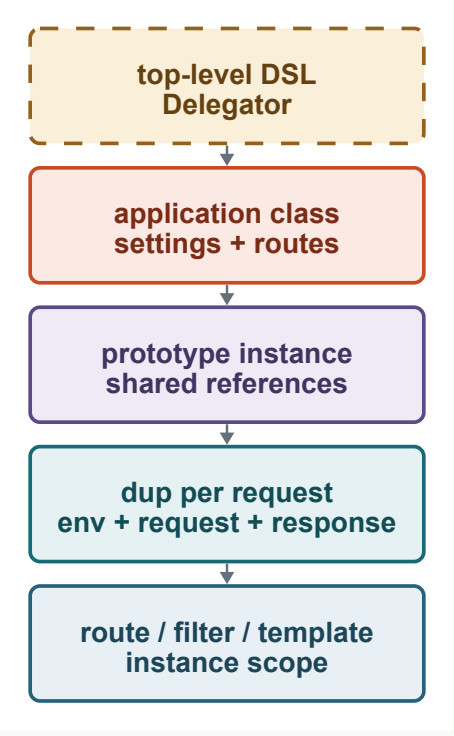
PROBE

Φτιάξε body που γράφει timestamps στα `each` και `close`. Πέρασέ το από `Rack::Lint`, `Rack::Head` και ένα middleware που αντικαθιστά το response. Δοκίμασε GET, HEAD και exception στο δεύτερο chunk. Επιβεβαίωσε ότι το close τρέχει μία τουλάχιστον φορά σε κάθε διαδρομή.

ΚΕΦΑΛΑΙΟ 3

Classic, modular και τρία scopes

Το Sinatra έχει class scope, request scope και delegation scope. Τα bugs αρχίζουν όταν ένα object τοποθετείται στο λάθος από τα τρία.



Σχήμα 3.1 · Η top-level DSL καταλήγει σε class registries και request duplicates.

Με `require "sinatra"`, τα top-level `get`, `before` και `set` δεν ορίζονται πάνω στο `main` object. Το `Sinatra::Delegator` τα προωθεί στη μοναδική `Sinatra::Application`. Αυτό είναι το classic style. Είναι εξαιρετικό για ένα μικρό executable app, αλλά σημαίνει μία classic application ανά Ruby process και implicit global composition.

Με `require "sinatra/base"`, ορίζεις subclass του `Sinatra::Base`. Αυτό είναι το modular style. Μπορείς να έχεις πολλές εφαρμογές, να τις κάνεις mount, να τις περάσεις ως middleware και να δημιουργήσεις isolated subclasses για tests. Δεν γίνεται αυτομάτως καλή architecture. Απλώς σου δίνει καθαρό object boundary.

```ruby
require "sinatra/base"

class PublicApi < Sinatra::Base
  get("/health") { [200, { "content-type" => "text/plain" }, ["ok"]] }
end

class AdminApi < Sinatra::Base
  get("/health") { [200, { "content-type" => "text/plain" }, ["admin ok"]] }
end
```

Application scope

Η class body τρέχει κατά το load. Εκεί καλούνται `get`, `before`, `helpers`, `register`, `use` και `set`. Routes και filters αποθηκεύονται σε class-level registries. Settings γίνονται singleton methods πάνω στην application class. Άρα αυτό το scope είναι process-level configuration, όχι χώρος για current request data.

```ruby
class Catalog < Sinatra::Base
  set :page_size, 50

  configure :production do
    set :show_exceptions, false
  end

  helpers do
    def canonical_limit
      Integer(params.fetch("limit", settings.page_size))
    end
  end
end
```

Το block του `helpers` εκτελείται σε class scope για να ορίσει instance methods. Τα methods αργότερα τρέχουν σε request scope. Το `settings.page_size` διαβάζει shared configuration. Το `params` διαβάζει per-request state.

Request scope και το shallow dup

Η class κρατά ένα prototype instance. Για κάθε request, το instance method `Base#call` κάνει `dup.call!(env)`. Το duplicate παίρνει νέα instance variables για env, params, request και response μέσα στο `call!`. Τα instance variables που γράφεις σε before filter ή route ανήκουν λοιπόν στο request duplicate.

Το `dup` όμως είναι shallow. Objects που υπήρχαν ήδη ως instance variables του prototype αντιγράφονται ως references. Το built-in `@template_cache` είναι χαρακτηριστικό παράδειγμα: τα request duplicates βλέπουν το ίδιο cache object. Το ίδιο θα συμβεί αν κάνεις initialize custom mutable array στο application instance.

```ruby
class UnsafeCounter < Sinatra::Base
  def initialize(app = nil)
    super
    @counts = Hash.new(0)
  end

  get "/hit/:key" do
    @counts[params[:key]] += 1
    @counts[params[:key]].to_s
  end
end
```

Το hash δεν είναι request-local. Δημιουργήθηκε στο prototype πριν από τα dups και μοιράζεται. Σε threaded server το read-modify-write δεν έχει application atomicity. Σε multi-process server κάθε worker έχει διαφορετικό hash. Άρα το ίδιο code είναι ταυτόχρονα racy και ασυνεπές ανά process.

Request-local state να δημιουργείται μέσα σε `call!`, filter ή route. Shared state να είναι explicit dependency με τεκμηριωμένο concurrency contract. Durable counters να ανήκουν σε store που υποστηρίζει την απαιτούμενη atomic operation.

`call` και `call!` δεν είναι συνώνυμα

Το instance `call(env)` δημιουργεί duplicate. Το `call!(env)` χρησιμοποιεί το ίδιο instance. Η επίσημη τεκμηρίωση αναφέρει το `call!` για internal request στο ίδιο application instance, αλλά αυτό έχει μεγάλο coupling: αντικαθιστά env, params, request και response πάνω στο active object. Nested `call!` μπορεί να διαλύσει το outer request context. Προτίμησε κοινό helper ή use case.

Η class method `App.call(env)` περνά από `synchronize` όταν το setting `lock` είναι ενεργό και μετά καλεί το prototype. Με `lock` false, διαφορετικά server threads μπορούν να εκτελούν διαφορετικά duplicates παράλληλα. Με `lock` true, όλο το process γίνεται single-request critical section. Αυτό είναι safety switch, όχι capacity strategy.

Closures κρατούν ό,τι συνέλαβαν

Το route block μετατρέπεται σε `UnboundMethod`, αλλά objects που έκλεισε το lexical scope παραμένουν references. Αν ένα array ή client δημιουργήθηκε έξω από την class και χρησιμοποιείται μέσα στο route, είναι shared ακόμη κι αν το route τρέχει σε νέο duplicate.

```ruby
audit_sink = Queue.new

AuditApp = Class.new(Sinatra::Base) do
  post "/events" do
    audit_sink << request.body.read
    status 202
  end
end
```

Το `Queue` έχει thread-safe contract, άρα η επιλογή μπορεί να είναι σωστή για in-process handoff. Δεν είναι durable και κάθε process έχει διαφορετικό queue. Το σημαντικό δεν είναι αν το object είναι global ή closure. Είναι το lifetime, το concurrency model και η εγγύηση που του ζητάς.

SOURCE TRACE

Δες `lib/sinatra/main.rb` για την ενεργοποίηση του Delegator και `Sinatra::Application`, `Sinatra::Delegator`, `Sinatra::Base.prototype`, `Base#call` και `Base#call!` στο `lib/sinatra/base.rb` του tag `v4.2.1`. Το request instance προκύπτει με `dup`, όχι με πλήρες `new` ανά request.

ΠΑΓΙΔΑ

Μη βαφτίζεις request-local οτιδήποτε είναι instance variable. Αν δημιουργήθηκε στο prototype πριν από το `dup`, το referenced mutable object μπορεί να είναι shared από όλα τα requests του process.

ΜΗΧΑΝΙΣΜΟΣ

Class scope κρατά configuration και registries. Request scope κρατά execution state. Delegation scope είναι convenience proxy προς την classic application. Το lifetime του object είναι πιο σημαντικό από το syntax που το προσπέλασε.

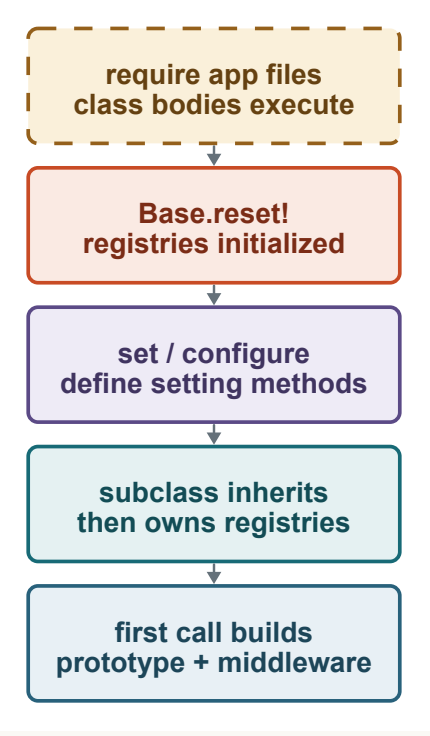
PROBE

Βάλε `object_id` για `self`, `settings`, `request`, `response` και ένα mutable object που αρχικοποιείται στο `initialize`. Στείλε δύο παράλληλα requests. Κατάγραψε ποια ids αλλάζουν και ποια μένουν ίδια. Επανάλαβε με δύο Puma workers για να δεις τη διαφορά thread sharing και process isolation.

ΚΕΦΑΛΑΙΟ 4

Boot, settings και inheritance

Στο Sinatra μεγάλο μέρος του προγράμματος εκτελείται όταν φορτώνεται η class. Το boot είναι φάση με side effects, ordering και fork semantics.



Σχήμα 4.1 · Από το class load στην πρώτη κατασκευή του Rack stack.

Όταν η Ruby διαβάζει μια Sinatra class, οι DSL calls εκτελούνται αμέσως. Το `get` κάνει compile route. Το `use` καταγράφει middleware. Το `set` ορίζει singleton getter, setter και predicate. Το `configure` αποφασίζει εκείνη τη στιγμή αν θα τρέξει το block για το ενεργό environment. Δεν υπάρχει κρυφή δεύτερη configuration pass.

Αυτό σημαίνει ότι η σειρά των requires είναι observable. Αν extension περιμένει setting, route hook ή helper να υπάρχει πριν εγγραφεί, το load order είναι μέρος του boot contract. Κυκλικά requires και autoload surprises είναι ιδιαίτερα ακριβά επειδή μπορούν να αφήσουν μισοχτισμένη class.

Τι κάνει πραγματικά το `set`

Για scalar value, το Sinatra παράγει singleton method που επιστρέφει το value. Για Proc, ο Proc γίνεται getter και εκτελείται κάθε φορά που διαβάζεται το setting. Για Hash, ο generated setter κάνει merge με την προηγούμενη τιμή. Αυτές οι τρεις μορφές έχουν διαφορετικό χρόνο αξιολόγησης.

```ruby
class ConfiguredApp < Sinatra::Base
  set :api_version, 2
  set(:asset_root) { File.join(root, "public") }
  set :limits, request_bytes: 1_048_576, page_size: 100

  configure :production do
    set :limits, page_size: 50
  end
end
```

Το `api_version` είναι stable scalar. Το `asset_root` υπολογίζεται σε κάθε read και εξαρτάται από το τρέχον `root`. Το δεύτερο `set :limits` κάνει merge, οπότε κρατά το `request_bytes` και αλλάζει το `page_size`. Αυτό είναι χρήσιμο, αλλά μπορεί να κρύψει typo σε key. Κάνε validation του τελικού config στο boot.

```ruby
module BootChecks
  def self.validate!(settings)
    limits = settings.limits
    raise "request_bytes missing" unless limits[:request_bytes].is_a?(Integer)
    raise "page_size invalid" unless (1..500).cover?(limits[:page_size])
  end
end

BootChecks.validate!(ConfiguredApp.settings)
```

Configuration error πρέπει να ρίχνει το process πριν αρχίσει να δέχεται traffic. Silent fallback σε production credential, hostname ή limit συνήθως μετατρέπει boot bug σε runtime incident.

Inheritance αντιγράφει policy, όχι live state

Όταν δημιουργείται subclass, το `Base.inherited` καλεί `subclass.reset!` και ορίζει `app_file`. Το reset φτιάχνει νέες registries για routes, filters, errors, middleware και extensions. Κατά την ανάγνωση, πολλά από αυτά συνδυάζονται με τα registries του superclass. Settings είναι inherited Ruby singleton methods μέχρι το child να τα ξαναορίσει.

```ruby
class ApiBase < Sinatra::Base
  set :default_content_type, :json

  before do
    headers "cache-control" => "no-store"
  end
end

class InternalApi < ApiBase
  set :audience, "internal"

  get("/health") { JSON.generate(status: "ok", audience: settings.audience) }
end
```

Το child route list είναι δικό του, αλλά το dispatch περνά και από routes του superclass. Τα filters των ancestors τρέχουν πριν από τα child filters επειδή το `filter!` κατεβαίνει πρώτα στο superclass. Αν η

base class γίνει μεγάλο implicit framework, κάθε αλλαγή έχει πλατύ blast radius. Κράτα την base class μικρή και προτίμησε modules ή middleware για policy με σαφές contract.

Prototype και middleware χτίζονται lazy

Η class method `prototype` κάνει memoize ένα instance. Η `new` ενός Sinatra app δεν επιστρέφει απλώς application instance: δημιουργεί instance, χτίζει το Rack middleware pipeline και επιστρέφει `Sinatra::Wrapper`. Η class-level `call` χρησιμοποιεί το memoized prototype, άρα η πρώτη πραγματική κλήση μπορεί να πληρώσει initialization αν δεν έχει προηγηθεί warmup.

Το `use` θέτει `@prototype = nil`, ώστε νέα middleware definition να αναγκάσει rebuild. Dynamic route mutation ή middleware mutation αφού ξεκινήσει traffic είναι δύσκολο concurrency contract. Treat the application class as frozen μετά το boot, ακόμη κι αν η Ruby δεν την έχει κάνει `freeze`.

Prefork αλλάζει το πού ανοίγουν resources

Με Puma workers και preload, η εφαρμογή φορτώνεται πριν από fork. Read-only Ruby pages μπορούν να μοιραστούν μέσω copy-on-write. Database connections, threads, mutex state, sockets και background loops δεν πρέπει να κληρονομηθούν τυφλά. Άνοιξε process-owned resources σε worker boot hook ή κάνε reconnect μετά το fork. Το Sinatra `on_start` λειτουργεί μόνο όταν το Sinatra ξεκινά μόνο του τον server, όχι γενικά για κάθε rackup deployment, άρα δεν είναι portable worker lifecycle hook.

Environment-specific config να επιλέγεται στο boot, αλλά secrets να μην τυπώνονται σε errors ή diagnostics. Κράτα fingerprint, presence και source name, όχι value. Σε rolling release δύο generations μπορεί να τρέχουν μαζί. Το config schema πρέπει να είναι backward compatible στο overlap window.

SOURCE TRACE
Στο `lib/sinatra/base.rb` του Sinatra 4.2.1 ακολούθησε `reset!`, `set`, `configure`, `inherited`, `prototype`, `.new`, `build` και `use`. Τα `on_start` και `on_stop` καλούνται από το self-hosted `run!` path μέσω `start_server` και `quit!`, όχι από οποιονδήποτε Rack handler lifecycle.

ΠΑΓΙΔΑ
Μην ανοίγεις thread ή network connection σε class body χωρίς να έχεις ορίσει τι συμβαίνει σε preload, fork, restart και test reload. Το ότι το boot code τρέχει μία φορά στο laptop δεν είναι production lifecycle guarantee.

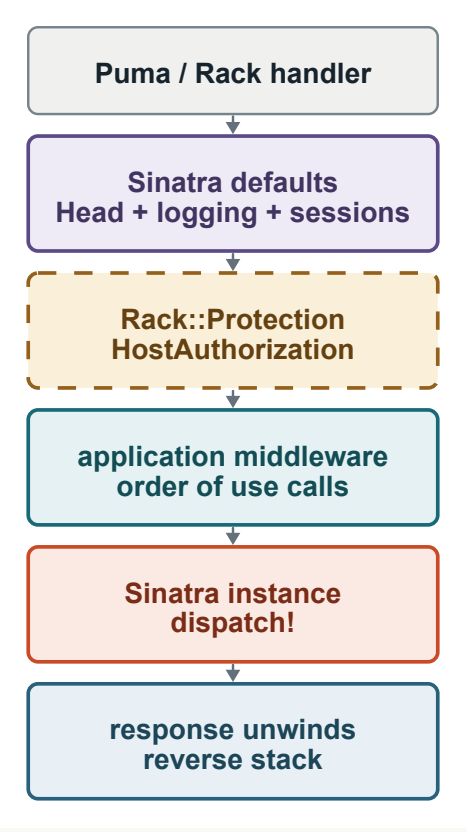
ΜΗΧΑΝΙΣΜΟΣ
Το Sinatra app είναι executable class definition και lazy-built Rack stack. Μετά το boot, settings και registries πρέπει να αντιμετωπίζονται ως immutable process configuration.

PROBE
Κατάγραψε PID και monotonic timestamp στο class load, στο `initialize`, στο πρώτο `call` και σε Puma worker hook. Τρέξε single mode, cluster χωρίς preload και cluster με preload. Επιβεβαίωσε πού δημιουργείται κάθε pool, socket και thread της εφαρμογής σου.

ΚΕΦΑΛΑΙΟ 5

Η πραγματική middleware στοίβα

Η σειρά των middleware είναι control flow. Καθορίζει ποιος βλέπει το request, ποιος χειρίζεται την exception και ποιος παρατηρεί το τελικό body close.



Σχήμα 5.1 · Τα defaults του Sinatra τυλίγουν τα application middleware και το endpoint.

Το `use` μοιάζει με λίστα, αλλά το runtime είναι nested calls. Με `use A`, `use B`, `run App`, το request περνά `A -> B -> App` και το response επιστρέφει `App -> B -> A`. Αν το `B` δεν καλέσει downstream, το App δεν εκτελείται. Αν το App σηκώσει exception, το `B` τη βλέπει πρώτο καθώς ξετυλίγεται η Ruby stack.

Το Sinatra 4.2.1 χτίζει πρώτα τα default middleware και μετά όσα δήλωσε η application class. Η σειρά των defaults είναι `ExtendedRack`, προαιρετικό `ShowExceptions`, προαιρετικό `Rack::MethodOverride`, `Rack::Head`, logging, sessions, `Rack::Protection` και `Rack::Protection::HostAuthorization`. Μετά έρχονται τα application `use` entries και στο κέντρο το Sinatra endpoint.

Επειδή το `Rack::Builder#to_app` κάνει reverse injection, το πρώτο `use` είναι το εξωτερικότερο wrapper. Άρα ένα middleware που δηλώνεται μέσα στην Sinatra class βρίσκεται πιο κοντά στο endpoint από τα default protections. Δεν θα δει request που κόπηκε από εξωτερικό HostAuthorization. Αν θέλεις global access logging για κάθε response, βάλε το έξω από την Sinatra app στο `config.ru`.

```ruby
require_relative "lib/seatly/app"
require_relative "lib/seatly/access_log"

use Seatly::AccessLog
run Seatly::App
```

Αυτό το `use` ανήκει σε εξωτερικό Rack::Builder. Τυλίγει ολόκληρο το `Seatly::App`, μαζί με το middleware pipeline που χτίζει εσωτερικά το Sinatra. Η διαφορά δεν είναι αισθητική. Αλλάζει ποια failures και blocked requests γίνονται observable.

Middleware contract με body-aware completion

Ένα σωστό middleware χωρίζει header latency από total response duration. Δεν μετατρέπει streaming body σε array και δεν ξεχνά το close.

```ruby
require "rack/body_proxy"

module Seatly
  class AccessLog
    def initialize(app, sink: EventSink.new)
      @app = app
      @sink = sink
    end

    def call(env)
      started = Process.clock_gettime(Process::CLOCK_MONOTONIC)
      status, headers, body = @app.call(env)
      wrapped = Rack::BodyProxy.new(body) do
        @sink.emit(
          name: "http.response.finished",
          status: status,
          route: env["sinatra.route"],
          seconds: elapsed_since(started)
        )
      end
      [status, headers, wrapped]
    rescue Exception => error
      @sink.emit(name: "http.request.failed", error_class: error.class.name)
      raise
    end

    private

    def elapsed_since(started)
      Process.clock_gettime(Process::CLOCK_MONOTONIC) - started
    end
  end
end
```

Το rescue χρησιμοποιεί `Exception` επειδή κάνει observation και ξανασηκώνει αμέσως. Δεν μετατρέπει signal ή fatal exception σε application response. Το body proxy αναλαμβάνει να προωθήσει `close` στο

original body. Ο sink πρέπει να έχει bounded, non-blocking ή σαφώς budgeted behavior. Synchronous network export μέσα στο response close μπορεί να κρατήσει worker μετά την αποστολή.

Env ownership και namespacing

Middleware μπορούν να προσθέσουν request id, authenticated principal, deadline ή parsed metadata στο env. Το key πρέπει να είναι namespaced. Ο owner πρέπει να ορίσει αν downstream επιτρέπεται να το αλλάξει και ποιος κάνει cleanup.

```ruby
class RequestIdentity
  def initialize(app, generator: SecureRandom.method(:uuid))
    @app = app
    @generator = generator
  end

  def call(env)
    incoming = env["HTTP_X_REQUEST_ID"].to_s
    env["seatly.request_id"] = trusted?(incoming) ? incoming : @generator.call
    @app.call(env)
  end

  private

  def trusted?(value)
    value.match?(/\A[a-zA-Z0-9_-]{16,80}\z/)
  end
end
```

Το να δεχτείς οποιοδήποτε request id επιτρέπει log injection και unbounded cardinality. Ακόμη και harmless metadata είναι input contract.

Authentication, transactions και exceptions

Authentication middleware είναι χρήσιμο όταν πολλαπλά Rack apps μοιράζονται τον ίδιο credential protocol. Authorization συνήθως χρειάζεται domain resource και μένει πιο καθαρό στο application layer. Transaction middleware γύρω από ολόκληρο request είναι επικίνδυνο για streaming και external I/O: κρατά transaction ανοιχτή μέχρι body close ή, αν κλείνει στο tuple return, μπορεί να κάνει commit πριν το lazy body εκτελέσει queries.

Το `ShowExceptions` είναι έξω από application middleware. Σε development μπορεί να μετατρέψει unhandled exception σε debug response. Ένα outer observability middleware στο `config.ru` βλέπει το αποτέλεσμα. Ένα inner middleware μπορεί να δει το raw exception καθώς ανεβαίνει, ανάλογα με το Sinatra error handler και τα settings `raise_errors` και `show_exceptions`.

Map και cascade

Με `Rack::Builder#map`, διαφορετικά apps παίρνουν προσαρμοσμένα `SCRIPT_NAME` και `PATH_INFO`. Με Sinatra ως middleware, ένα route miss καλεί `forward` προς το downstream app. Χωρίς downstream, σηκώνει `Sinatra::NotFound`. Το `X-Cascade: pass` σε 404 είναι convention που μπορεί να χρησιμοποιήσει outer router, αλλά δεν σημαίνει ότι κάθε server θα κάνει αυτόματα fallback.

SOURCE TRACE

Δες `Sinatra::Base.build`, `setup_default_middleware`, `setup_middleware` και τα επιμέρους `setup_*` methods στο `lib/sinatra/base.rb` του tag `v4.2.1`. Η nesting σειρά επιβεβαιώνεται στο `Rack::Builder#to_app` του Rack 3.2.7, όπου το `@use.reverse.inject(app)` κατασκευάζει τη στοίβα.

ΠΑΓΙΔΑ

Μη βάζεις database transaction middleware γύρω από streaming response. Ή θα κρατήσεις locks όσο περιμένει ο client ή θα κλείσεις την transaction πριν εκτελεστεί το lazy body.

ΜΗΧΑΝΙΣΜΟΣ

Application `use` και outer `config.ru use` δεν έχουν το ίδιο observation surface. Σχεδίασε middleware order με request path, response path, exception path και body close path.

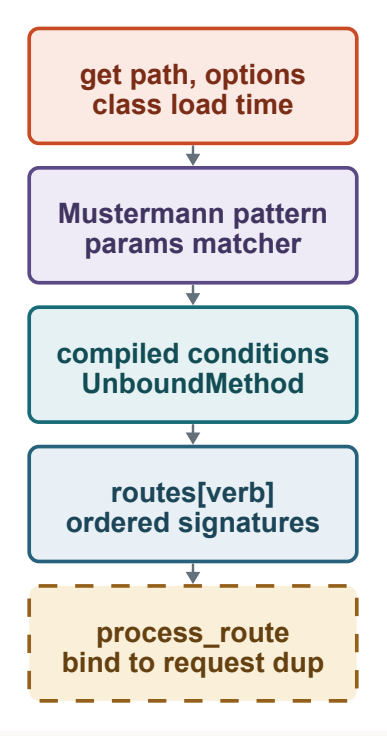
PROBE

Φτιάξε τρία middleware που γράφουν `before`, `after`, `rescue` και `close` με μοναδικό όνομα. Βάλε ένα έξω από το app και δύο μέσα. Δοκίμασε normal route, 404, blocked Host και body που σηκώνει exception στο δεύτερο chunk. Η σειρά του trace είναι η πραγματική σου middleware αρχιτεκτονική.

ΚΕΦΑΛΑΙΟ 6

Routes ως compiled πρόγραμμα

Ένα route δεν είναι hash lookup. Στο boot μετατρέπεται σε pattern, conditions και bindable Ruby method που μπαίνουν σε ordered registry.



Σχήμα 6.1 · Η DSL κάνει compile το route πριν έρθει το πρώτο request.

Όταν γράφεις `get "/events/:id"`, το Sinatra εκτελεί class method. Η `route` καλεί `compile!`, η οποία μετατρέπει το path σε Mustermann pattern, συλλέγει conditions και δημιουργεί προσωρινά instance method με το route block. Παίρνει το `UnboundMethod`, αφαιρεί το named method από την class και αποθηκεύει wrapper που θα το κάνει bind στο request instance.

Το registry είναι `routes[verb]`, με array signatures στη σειρά declaration. Δεν υπάρχει automatic specificity ranking. `/events/:id` πριν από `/events/new` μπορεί να καταπιεί το `new` ως id. Η σειρά είναι μέρος του deployed protocol και πρέπει να δοκιμάζεται.

```ruby
class EventsApi < Sinatra::Base
  get "/events/new" do
    content_type :json
    JSON.generate(form: "new_event")
  end

  get "/events/:id" do |id|
    content_type :json
    JSON.generate(id: id)
  end
end
```

Η στατική route μπαίνει πρώτη επειδή είναι πιο στενή. Αυτό είναι ανθρώπινη policy, όχι optimization του router. Σε μεγάλη εφαρμογή κράτα related routes κοντά και γράψε precedence tests για overlaps.

Mustermann είναι το pattern boundary

String paths δεν γίνονται απλή Ruby Regexp από το Sinatra. Περνούν από `Mustermann.new` με τα global `mustermann_opts` και τα per-route overrides. Το pattern object απαντά σε `params(route)`. Για regular patterns το Sinatra κάνει επιπλέον match, γεμίζει `params[:captures]` και περνά captures ως block arguments.

```ruby
class AssetsApi < Sinatra::Base
  set :mustermann_opts, type: :sinatra

  get "/assets/:name.:ext" do |name, ext|
    halt 415 unless %w[css js].include?(ext)
    [200, { "content-type" => "text/plain" }, ["#{name}.#{ext}"]]
  end
end
```

Το pattern κάνει structural match. Δεν επικυρώνει ότι το `name` αντιστοιχεί σε επιτρεπτό file ούτε ότι το id υπάρχει. Route matching, input validation και authorization είναι τρία διαφορετικά βήματα.

Regex routes είναι χρήσιμα, αλλά δημιουργούν δύο κινδύνους. Ο πρώτος είναι catastrophic backtracking αν το pattern εφαρμοστεί σε attacker-controlled path. Ο δεύτερος είναι ασαφές capture contract μετά από refactor. Προτίμησε Mustermann string syntax όταν εκφράζει το path. Αν χρειάζεται regex, κράτα το anchored σύμφωνα με τις Mustermann options, μικρό και με test σε μακριά adversarial strings.

Το route block διατηρεί Ruby semantics

Το generated `UnboundMethod` σημαίνει ότι `self` κατά το request είναι το request duplicate. Helpers είναι instance methods και βρίσκονται κανονικά. Lexical constants και captured local objects παραμένουν κανονικά Ruby references. Η compilation δεν μετατρέπει business code σε immutable plan.

Η arity αλλάζει την κλήση. Zero-arity block καλείται χωρίς extracted values. Άλλο block παίρνει flattened pattern values. Params παραμένουν διαθέσιμα και ως `params`, αλλά block arguments κάνουν το route contract πιο ορατό.

```ruby
get "/reports/:year/:month" do |year, month|
  period = Date.new(Integer(year, 10), Integer(month, 10), 1)
  ReportRenderer.call(period: period)
rescue ArgumentError
  halt 400, "invalid period"
end
```

Το local rescue εδώ καλύπτει μόνο το parse που γνωρίζουμε. Δεν κάνει catch κάθε `StandardError`, γιατί database outage ή programming bug δεν είναι invalid client input.

HEAD registration και route hooks

Το `get` αποθηκεύει GET route και ξαναχρησιμοποιεί τις ίδιες conditions για να καταγράψει HEAD route. Αυτό επιτρέπει στο route να υπολογίσει headers όπως σε GET, ενώ το outer `Rack::Head` αφαιρεί και κλείνει το body. Αν έχεις explicit `head` route, η declaration order αποφασίζει ποιο HEAD route θα τρέξει.

Extensions μπορούν να υλοποιήσουν `route_added` hook. Καλείται με verb, path και original block μετά την εγγραφή. Είναι καλό σημείο για route inventory ή boot-time validation. Δεν είναι λόγος να μεταλλάξεις route registry ενώ τρέχουν requests.

Route inventory ως artifact

Σε production system, paths και verbs είναι compatibility surface. Μπορείς στο boot να εξαγάγεις ένα bounded inventory από `settings.routes`, χωρίς να κάνεις request-time introspection. Κατέγραψε verb, pattern string, source location και policy tags. Μην καταγράφεις captured closure objects ή source code.

Ένα route count metric δεν λέει πολλά. Ένα diff που δείχνει ότι αφαιρέθηκε `POST /reservations` ή ότι generic wildcard μετακινήθηκε πριν από health route είναι release evidence.

SOURCE TRACE
Στο Sinatra 4.2.1 δες `Sinatra::Base.route`, `compile!`, `compile`, `generate_method`, `get` και `invoke_hook` στο `lib/sinatra/base.rb`. Το runtime matching συνεχίζει στα `route!` και `process_route`. Το pattern contract προέρχεται από το compatible Mustermann 3.x.

ΠΑΓΙΔΑ
Μη δηλώνεις dynamic routes από request. Μεταλλάσσεις shared class registry, αλλάζεις precedence για concurrent requests και δημιουργείς unbounded methods ή closures χωρίς lifecycle.

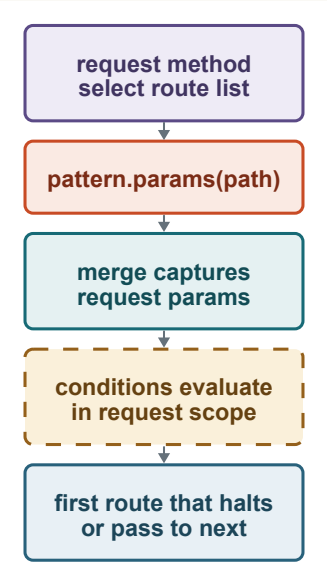
ΜΗΧΑΝΙΣΜΟΣ
Routes είναι ordered compiled program. Declaration order, pattern semantics, conditions, arity και captured references ανήκουν στο execution model.

PROBE
Τύπωσε για κάθε verb τα stored pattern classes και το source location του original block μέσω ενός `route_added` extension. Πρόσθεσε static, named, wildcard και regex routes που overlap. Στείλε test requests και επιβεβαίωσε τη σειρά που υπολόγισες πριν τρέξεις το app.

ΚΕΦΑΛΑΙΟ 7

Precedence, conditions και negotiation

To matching είναι pipeline: verb, ordered pattern, extracted params, conditions και τελικά route body. Κάθε στάδιο μπορεί να αλλάξει το αποτέλεσμα.



Σχήμα 7.1 · Το route επιλέγεται μόνο αφού περάσουν pattern και conditions.

Στην αρχή του `route!`, το Sinatra παίρνει μόνο τη λίστα του current request method. Για κάθε signature καλεί `process_route`. Το path κανονικοποιείται ανάλογα με `empty_path_info` και `strict_paths`, μετά το pattern επιστρέφει params ή nil. Αν υπάρχει match, τα route params μπαίνουν προσωρινά στο current params hash και οι conditions εκτελούνται με `self` bound στο request instance.

Η πρώτη route που ολοκληρώνεται κάνει `throw :halt` μέσω του `route_eval`. Condition που επιστρέφει false συμπεριφέρεται σαν `pass` και η αναζήτηση συνεχίζει. Αν τελειώσουν οι child routes, το Sinatra ψάχνει routes των superclasses. Άρα inheritance και declaration order σχηματίζουν μία συνολική ordered chain.

Conditions είναι executable policy

Built-in conditions υπάρχουν για host name, user agent και provided media types. Μπορείς να ορίσεις custom condition μέσω setting που καλεί `condition`. Το block τρέχει σε request scope και μπορεί να διαβάσει principal, headers ή feature state.

```ruby
module RoutePolicy
  def self.registered(app)
    app.set(:requires_role) do |role|
      condition do
        principal = env["seatly.principal"]
        principal && principal.roles.include?(role.to_s)
      end
    end
  end
end

class AdminApi < Sinatra::Base
  register RoutePolicy

  get "/metrics", requires_role: "operator" do
    content_type :json
    JSON.generate(Metrics.snapshot)
  end
end
```

Αυτό λειτουργεί, αλλά authorization που εξαρτάται από loaded resource συνήθως είναι καθαρότερο μέσα στο use case. Route condition ταιριάζει σε coarse access policy πριν από ακριβή δουλειά. Αν η condition κάνει database query για κάθε candidate, overlapping routes πολλαπλασιάζουν το κόστος και κάνουν timing δύσκολο να προβλεφθεί.

Condition name είναι setting method. Collision με άλλο setting ή extension μπορεί να αλλάξει semantics. Namespacing και boot-time contract test είναι χρήσιμα όταν γράφεις reusable extensions.

Content negotiation χωρίς κρυφές παραδοχές

Το `provides` μετατρέπει extensions σε MIME types και χρησιμοποιεί `Request#preferred_type`. Το Sinatra αναλύει το `Accept`, ταξινομεί entries με quality, wildcard specificity και αριθμό parameters και βρίσκει πρώτο compatible offered type. Αν υπάρχει match και δεν έχει ήδη response content type, η condition το ορίζει.

```ruby
get "/events/:id", provides: :json do |id|
  event = Events.fetch(id)
  JSON.generate(event)
end

get "/events/:id", provides: :html do |id|
  @event = Events.fetch(id)
  erb :event
end
```

Η order εδώ είναι product decision όταν ο client στείλει `*/*`. Ο πρώτος compatible route κερδίζει. Αν browser και API client χρειάζονται σταθερά defaults, γράψε τα ρητά και πρόσθεσε `Vary: Accept` όταν shared cache μπορεί να κρατήσει διαφορετικές representations του ίδιου URL.

Before filter που θέτει content type το κάνει pin. Στο `dispatch!`, μετά από κάθε before filter το Sinatra σημειώνει αν υπάρχει content type. Στο `route!` καθαρίζει το content type πριν από κάθε candidate μόνο αν δεν ήταν pinned. Αυτό εμποδίζει failed `provides` candidates να αφήσουν stale negotiation state, αλλά σημαίνει ότι global before content type περιορίζει τις routes.

Path normalization είναι policy

Με `strict_paths` true, `/events` και `/events/` είναι διαφορετικά. Με false, το Sinatra αφαιρεί το trailing slash εκτός από root πριν κάνει match. Empty `PATH_INFO` γίνεται `/` εκτός αν έχει ενεργοποιηθεί `empty_path_info`. Αυτές οι ρυθμίσεις επηρεάζουν mounted apps και proxies, άρα δεν αλλάζουν αθόρυβα σε rolling release.

Path traversal protection μπορεί να τροποποιήσει το request path πριν φτάσει στις routes. Αυτό είναι ένα ακόμη παράδειγμα όπου middleware order προηγείται του router mental model. Test encoded slashes, dot segments, duplicate slashes και UTF-8 paths μέσω πραγματικού Rack stack, όχι μόνο με direct method call.

`pass` ως controlled fallback

Δύο routes μπορούν να μοιράζονται pattern και να αποφασίζουν αργότερα. Αυτό είναι χρήσιμο για negotiation ή gradual migration, αλλά η απόφαση πρέπει να είναι cheap και deterministic. Αν route A κάνει effect και μετά `pass`, route B θα εκτελεστεί επίσης. Το `pass` δεν κάνει rollback.

RUBY

```ruby
get "/documents/:id" do |id|
  document = Documents.find(id)
  pass unless document.public?
  present_public(document)
end

get "/documents/:id" do |id|
  document = Documents.find(id)
  authorize!(document, action: :read)
  present_private(document)
end
```

Το example κάνει δύο reads. Μπορεί να είναι αποδεκτό ή να χρειάζεται shared loader helper. Μην αποθηκεύσεις loaded document σε process-global cache απλώς για να γλιτώσεις query. Αν το βάλεις σε request env, όρισε namespaced key και authorization semantics.

SOURCE TRACE

Ακολούθησε `route!` και `process_route` στο `lib/sinatra/base.rb` του Sinatra 4.2.1. Για negotiation δες `Request#accept`, `AcceptEntry#priority`, `MimeTypeEntry#accepts?` και το class method `provides`. Το content type pin ορίζεται στο before-filter μέρος του `dispatch!`.

ΠΑΓΙΔΑ

Μη βάζεις non-idempotent effect πριν από πιθανό `pass`. Το επόμενο route θα τρέξει κανονικά και το πρώτο effect δεν αναιρείται.

ΜΗΧΑΝΙΣΜΟΣ

Route matching είναι ordered execution με προσωρινό params state. Conditions είναι code, όχι metadata. Negotiation, inheritance και path settings μπορούν να αλλάξουν ποιο route κερδίζει.

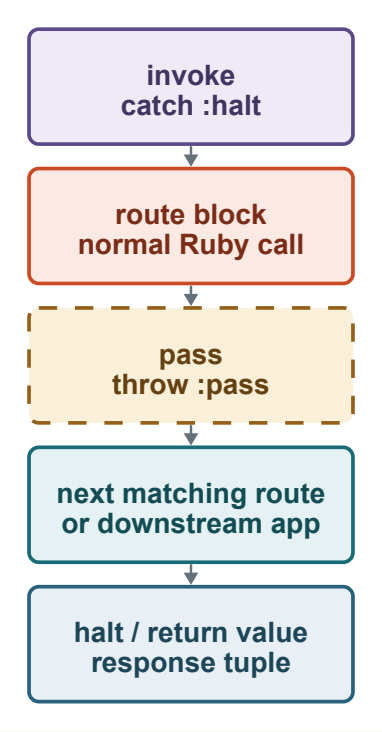
PROBE

Φτιάξε matrix από Accept headers: absent, `*/*`, JSON με μεγαλύτερο q, HTML με μεγαλύτερο q και unsupported type. Κατάγραψε route, status, content type και `Vary`. Πρόσθεσε condition που κάνει `pass` και επιβεβαίωσε ότι params από το πρώτο candidate δεν διαρρέουν στο δεύτερο.

ΚΕΦΑΛΑΙΟ 8

halt, pass, forward και control flow

Τα `halt` και `pass` είναι non-local control flow με Ruby `throw` και `catch`. Το `forward` είναι κανονική downstream Rack κλήση.



Σχήμα 8.1 · Τα τρία control-flow εργαλεία σταματούν ή συνεχίζουν διαφορετικό επίπεδο.

Ένα route return δεν περνά απλώς πίσω στον caller. Το `route_eval` τυλίγει το route και κάνει `throw :halt, yield`. Το outer `invoke` έχει `catch :halt` και μεταφράζει το caught value σε Sinatra response. Αυτό εξηγεί γιατί ένα απλό String, status Integer, Rack tuple ή enumerable body μπορούν να είναι route result.

Το `halt` κάνει το ίδιο throw νωρίτερα. Μπορεί να πάρει status, headers και body. Δεν είναι exception και ένα `rescue` δεν το πιάνει. Τα Ruby `ensure` blocks που βρίσκονται ανάμεσα στο throw και στο catch τρέχουν κανονικά.

```ruby
post "/admin/reindex" do
  principal = env["seatly.principal"]
  halt 401, { "content-type" => "application/json" },
       [JSON.generate(error: "unauthenticated")] unless principal

  halt 403, { "content-type" => "application/json" },
       [JSON.generate(error: "forbidden")] unless principal.operator?

  Reindex.enqueue
  status 202
end
```

Το explicit tuple αποφεύγει default HTML content type. Τα header names είναι lowercase για Rack 3 correctness. Η authentication και authorization απόφαση είναι ξεχωριστή ώστε 401 και 403 να μη συγχέονται.

`pass` συνεχίζει το route search

Το `pass` κάνει `throw :pass`. Το `process_route` έχει το αντίστοιχο catch. Άρα εγκαταλείπει conditions ή route block και επιστρέφει στο loop του `route!`. Δεν βγαίνει από ολόκληρο το request. Optional block στο `pass` μπορεί να κρατηθεί ως fallback και να αξιολογηθεί αν εξαντληθούν routes.

```ruby
get "/exports/:id" do |id|
  export = Exports.find(id)
  pass { halt 409, "export is not ready" } unless export.ready?
  send_file export.path, disposition: :attachment
end

get "/exports/:id" do
  halt 404
end
```

Το fallback block είναι advanced feature και εύκολα δυσκολεύει την ανάγνωση. Συνήθως ένα explicit branch μέσα σε ένα route είναι πιο καθαρό. Χρησιμοποίησε `pass` όταν όντως υπάρχουν πολλαπλοί handlers για το ίδιο path space.

`forward` χρειάζεται downstream app

Όταν Sinatra app λειτουργεί ως middleware, κρατά `@app`. Αν κανένα route δεν ταιριάξει, το `route_missing` καλεί `forward`. Το `forward` καλεί downstream με το ίδιο env και αντιγράφει status, body και headers στη Sinatra response. Χωρίς downstream σηκώνεται `Sinatra::NotFound`.

```ruby
class ApiGateway < Sinatra::Base
  get("/health") { "gateway ok" }

  before "/internal/*" do
    halt 403 unless env["seatly.principal"]&.operator?
  end
end

use ApiGateway
run LegacyRackApp
```

Εδώ το Sinatra είναι middleware. Route miss περνά στο `LegacyRackApp`. Ένα matching before filter μπορεί να κόψει το request πριν από το downstream. Η σειρά και το path filter είναι gateway policy, άρα χρειάζονται integration tests που καλύπτουν both matched και forwarded paths.

Internal `call` είναι subrequest

Μπορείς να καλέσεις `call` με modified env για να ενεργοποιήσεις άλλο route. Το instance `call` κάνει duplicate, ενώ `call!` χρησιμοποιεί το ίδιο request object και αντικαθιστά current state. Και στις δύο περιπτώσεις παίρνεις Rack response με body lifecycle.

```ruby
get "/legacy-status" do
  inner_env = env.merge("PATH_INFO" => "/health", "QUERY_STRING" => "")
  status_code, inner_headers, inner_body = call(inner_env)
  chunks = inner_body.each.to_a
  inner_body.close if inner_body.respond_to?(:close)
  [status_code, inner_headers, chunks]
end
```

Αυτό είναι syntactically valid, αλλά αρχιτεκτονικά ακριβό. Ξανατρέχει dispatch, filters και route matching. Μπορεί να διπλασιάσει logging ή να μπερδέψει request ids. Κοινή λειτουργία να βγαίνει σε method ή use case.

Transaction semantics δεν ακολουθούν το control flow

`halt` μέσα σε transaction block είναι throw. Ruby ensure τρέχει, αλλά ο database adapter μπορεί να θεωρήσει το block επιτυχημένο αν δεν είδε exception. Αν θέλεις rollback, πρέπει να χρησιμοποιήσεις το contract του adapter ή να αποφασίσεις πριν ανοίξεις transaction. Το ίδιο ισχύει για `pass`: δεν ακυρώνει writes που έγιναν ήδη.

Ένα broad wrapper που κάνει commit αν το route δεν raised μπορεί να κάνει commit μετά από `halt 422`. HTTP status και transaction success είναι διαφορετικές διαστάσεις. Βάλε transaction γύρω από application operation, όχι γύρω από arbitrary HTTP control flow.

`invoke` και return shape

Το `invoke` μετατρέπει Integer ή String σε μονο-element array. Αν το result είναι array με πρώτο Integer, αφαιρεί status, τελευταίο body και ό,τι μένει το περνά ως headers. Αν είναι άλλο enumerable,

γίνεται body. Unknown scalar που δεν απαντά σε `each` δεν θα γίνει αυτόματα χρήσιμο response. Προτίμησε explicit return shapes στα shared helpers.

SOURCE TRACE

Δες `Sinatra::Base#halt`, `#pass`, `#forward`, `#invoke`, `#route_eval`, `#route_missing` και `#route!` στο `lib/sinatra/base.rb` του Sinatra 4.2.1. Το `catch(:halt)` βρίσκεται στο `invoke`, ενώ το `catch(:pass)` είναι μέσα στο `process_route`.

ΠΑΓΙΔΑ

HTTP control flow δεν είναι transaction control flow. `halt`, `pass` και normal return μπορούν να οδηγήσουν σε commit αν ο database wrapper δεν έχει δικό του explicit failure signal.

ΜΗΧΑΝΙΣΜΟΣ

`halt` ολοκληρώνει το current dispatch, `pass` συνεχίζει στα επόμενα routes και `forward` καλεί downstream Rack app. Μην τα χρησιμοποιείς ως υποκατάστατο application workflow.

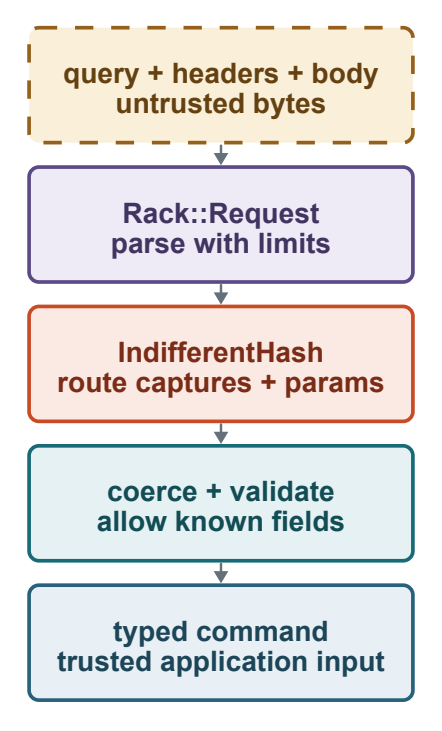
PROBE

Τύλιξε route με Ruby `begin`, `rescue` και `ensure`. Δοκίμασε normal return, `halt`, `pass` και exception. Κατάγραψε ποια blocks τρέχουν. Με test database, επανάλαβε μέσα σε transaction και επιβεβαίωσε αν έγινε commit ή rollback αντί να το συμπεράνεις από το HTTP status.

ΚΕΦΑΛΑΙΟ 9

Request input, params και trust boundaries

Το `params` είναι βολικό merge διαφορετικών πηγών. Δεν είναι validated command και δεν διατηρεί από μόνο του την provenance κάθε value.



Σχήμα 9.1 · Τα untrusted bytes πρέπει να γίνουν typed command πριν φτάσουν στο domain.

Στην αρχή του `dispatch!`, το Sinatra ζητά `request.params` από το `Sinatra::Request`, που κληρονομεί από `Rack::Request`. Query params, form body και multipart fields αναλύονται από το Rack. Το Sinatra τα βάζει σε `IndifferentHash`, εφαρμόζει default encoding και αργότερα προσθέτει route captures όταν εξετάζει candidate route.

Για ίδιο key, το route param συνήθως υπερισχύει του υπάρχοντος request param όταν δεν είναι nil. Αυτό αποτρέπει `?id=other` από το να αντικαταστήσει `/events/:id`, αλλά το merged hash εξακολουθεί να κρύβει την πηγή. Για security decision προτίμησε `request.get_header`, explicit JSON document ή named route argument αντί να ψάχνεις γενικά στο params.

```ruby
post "/events/:event_id/reservations" do |event_id|
  input = ReservationInput.new(
    event_id: Integer(event_id, 10),
    seat: params.fetch("seat").to_s,
    request_key: request.get_header("HTTP_IDEMPOTENCY_KEY").to_s
  )
  result = CreateReservation.call(input)
  present(result)
rescue KeyError, ArgumentError
  halt 422, { "content-type" => "application/json" },
       [JSON.generate(error: "invalid_input")]
end
```

Η μετατροπή σε immutable input object κάνει ορατά τα accepted fields. Δεν περνάμε ολόκληρο το params hash στο repository. Το rescue καλύπτει known coercion failures, όχι κάθε exception.

Parse limits πριν από allocations

Nested query strings, πολλά keys και μεγάλα multipart bodies μπορούν να καταναλώσουν CPU, memory και temp files πριν τρέξει route. Τα limits πρέπει να υπάρχουν στον edge server, στο Rack parser ή σε middleware που τρέχει πριν διαβαστεί το body. Route-level `request.body.read` είναι αργά αν ο server έχει ήδη δεχτεί unbounded upload.

JSON body δεν μπαίνει αυτόματα στο Sinatra params από τον core. Διάβασέ το μία φορά, με media type και size contract. Ένα helper μπορεί να κρατήσει parsed document στο request env.

```ruby
module JsonInput
  MAX_BYTES = 1_048_576

  def json_document
    return env["seatly.json"] if env.key?("seatly.json")

    halt 415 unless request.media_type == "application/json"
    payload = request.body.read(MAX_BYTES + 1)
    halt 413 if payload.bytesize > MAX_BYTES
    env["seatly.json"] = JSON.parse(payload)
  rescue JSON::ParserError
    halt 400, "invalid JSON"
  end
end
```

Το body read είναι consuming operation. Αν άλλο middleware πρέπει να το διαβάσει μετά, χρειάζεται rewind όταν το input το υποστηρίζει ή shared parsed representation με σαφές key. Μην κάνεις parse δύο φορές. Μην λογάρεις raw body σε error, γιατί μπορεί να περιέχει credentials ή προσωπικά δεδομένα.

Invalid encoding και malformed forms

Το `Sinatra::Request#params` μετατρέπει `Rack::Utils::ParameterTypeError` και `InvalidParameterError` σε `Sinatra::BadRequest` με escaped message. EOF σε multipart γίνεται επίσης BadRequest. Αυτό δίνει 400 αντί για generic 500, αλλά δεν αντικαθιστά application validation.

Encoding conversion μπορεί να αποτύχει πριν φτάσει η route. Δοκίμασε invalid UTF-8, duplicate scalar/array shapes και unexpected nested objects. Ένα field που η εφαρμογή περιμένει String μπορεί να φτάσει ως Hash αν ο parser δεχτεί `name[key]=value`. Κάνε type checks πριν από `to_s`, γιατί το `to_s` μπορεί να κρύψει shape attack αντί να το απορρίψει.

Headers έχουν δικό τους contract

Rack env header names έχουν CGI μορφή όπως `HTTP_AUTHORIZATION`, εκτός από `CONTENT_TYPE` και `CONTENT_LENGTH`. Το `request.get_header` κάνει την πηγή σαφή. Normalize μόνο ό,τι επιτρέπει το protocol. Ένα idempotency key μπορεί να έχει bounded ASCII grammar. Bearer token δεν πρέπει να lower-case. Host, Forwarded και client IP χρειάζονται trusted proxy model.

Cookies είναι επίσης attacker-controlled serialized input, ακόμη όταν είναι signed ή encrypted. Η κρυπτογραφία αποδεικνύει ότι εκδόθηκαν από holder του key, όχι ότι το περιεχόμενο είναι τρέχον, authorized ή συμβατό με νέο code.

Tempfiles και lifetime

Multipart upload μπορεί να δώσει tempfile. Μην αποθηκεύεις το path για job που θα τρέξει αργότερα. Το tempfile ανήκει στο request lifecycle και μπορεί να διαγραφεί. Κάνε bounded streaming copy σε owned storage, υπολόγισε digest και κατάγραψε durable reference πριν επιστρέψεις success.

Validation errors χρειάζονται stable machine code και field paths. Μην επιστρέφεις raw parser exception message ως public API. Μπορεί να αλλάξει με Rack upgrade και να αποκαλύψει implementation detail.

SOURCE TRACE
Στο Sinatra 4.2.1 ακολούθησε `Sinatra::Request#params`, `Base#dispatch!` και `Base#process_route` στο `lib/sinatra/base.rb`. Το πρώτο μεταφράζει Rack parser errors, το δεύτερο φορτώνει request params και το τρίτο προσθέτει προσωρινά route params και regex captures.

ΠΑΓΙΔΑ
Μη θεωρείς ότι `params[:id]` έχει μία πηγή ή έναν τύπο. Query, form και route captures ενώνονται. Nested parser input μπορεί να αλλάξει String σε Array ή Hash.

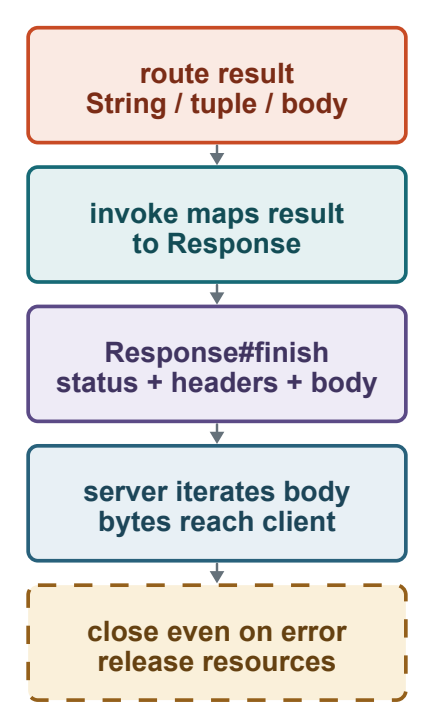
ΜΗΧΑΝΙΣΜΟΣ
Parsing παράγει data structure. Validation παράγει trusted command. Ανάμεσα στα δύο χρειάζονται size, shape, type, encoding και provenance checks.

PROBE
Στείλε query και form με ίδιο key, nested αντί για scalar value, invalid UTF-8, malformed multipart και JSON ένα byte πάνω από το limit. Κατάγραψε status, public error code και allocations. Επιβεβαίωσε ότι raw payload και secrets δεν μπαίνουν στα logs.

ΚΕΦΑΛΑΙΟ 10

Response semantics, HEAD και body lifecycle

To route result γίνεται response object, μετά Rack tuple και τέλος bytes. Σε κάθε μετάβαση αλλάζουν οι κανόνες που ισχύουν.



Σχήμα 10.1 · Από το route return μέχρι την κατανάλωση και το close του body.

Το `invoke` παίρνει το result του route και ενημερώνει `Sinatra::Response`. String γίνεται body. Integer γίνεται status. Array που ξεκινά με Integer ερμηνεύεται ως response shape. Άλλο enumerable γίνεται body. Στο τέλος του `call!`, αν δεν υπάρχει content type, το Sinatra χρησιμοποιεί content type που φέρει το rendered value ή το `default_content_type`.

Το `Response#body=` ξετυλίγει nested Rack::Response objects και τυλίγει String σε array. Το `finish` αφαιρεί content headers για informational, 204 και 304 όπου απαιτείται, αδειάζει body για 204 και 304 και υπολογίζει content-length μόνο όταν το body είναι Array, υπάρχει content type και δεν έχει ήδη οριστεί length.

```ruby
get "/events/:id" do |id|
  event = Events.fetch(id)
  payload = JSON.generate(event)
  [200, { "content-type" => "application/json" }, [payload]]
end
```

Το explicit tuple είναι χρήσιμο σε API boundary. Αν χρησιμοποιήσεις helpers, μπορείς να γράψεις το ίδιο πιο φυσικά, αρκεί να παραμένει σαφής η τελική representation.

HEAD δεν είναι απλώς δεύτερο route

Το `get` καταγράφει και HEAD route. Η route εκτελείται ώστε να παραχθούν status και headers, μετά το default `Rack::Head` κλείνει το returned body αν έχει `close` και το αντικαθιστά με κενό array. Άρα expensive body generation μέσα στο route μπορεί ακόμη να συμβεί σε HEAD, ειδικά αν δημιουργείς ολόκληρο String πριν το return.

Για ακριβό export, ένα explicit HEAD route πριν από GET μπορεί να υπολογίσει metadata χωρίς να παράγει bytes. Πρέπει όμως να δίνει τα ίδια validators και content headers που θα είχε το GET.

```ruby
head "/exports/:id" do |id|
  export = Exports.fetch(id)
  content_type export.media_type
  headers "content-length" => export.bytesize.to_s
  status 200
end

get "/exports/:id" do |id|
  export = Exports.fetch(id)
  send_file export.path, type: export.media_type, disposition: :attachment
end
```

Επειδή `get` προσθέτει επίσης HEAD signature, η explicit HEAD route πρέπει να δηλωθεί πριν από το GET για να κερδίσει στο ordered registry.

Status και body compatibility

204 και 304 δεν επιτρέπεται να μεταφέρουν representation body ή content headers. 1xx responses έχουν επίσης ειδικούς κανόνες. Μην επιστρέφεις generic JSON error helper ανεξάρτητα από status. Το helper πρέπει να γνωρίζει αν το status επιτρέπει body.

Redirect helper επιλέγει 303 για non-GET request πάνω σε HTTP/1.1 και 302 σε άλλες περιπτώσεις. Για public API προτίμησε explicit 303, 307 ή 308 ανάλογα με το αν ο client πρέπει να αλλάξει method. Redirect semantics είναι protocol decision, όχι cosmetic navigation.

Headers δεν πρέπει να περιέχουν raw user input με CR ή LF. Το Sinatra `attachment` καθαρίζει filename με basename και percent replacements για quote, CR και LF. Custom `Content-Disposition` builder πρέπει να κάνει αντίστοιχη δουλειά ή να χρησιμοποιεί vetted helper.

Lazy body, eager headers

Headers και status αποφασίζονται πριν ο server αρχίσει να καταναλώνει body. Αν lazy body σηκώσει exception μετά το πρώτο chunk, συνήθως δεν μπορείς να μετατρέψεις καθαρά το response σε JSON 500. Τα headers έχουν ήδη σταλεί. Η recovery είναι connection close, logging και application-level resume protocol, όχι δεύτερο response.

RUBY

```ruby
class JsonLinesBody
  def initialize(source)
    @source = source
  end

  def each
    @source.each { |item| yield JSON.generate(item) << "\n" }
  end

  def close
    @source.close if @source.respond_to?(:close)
  end
end
```

Το source πρέπει να έχει bounded lifetime. Αν είναι database cursor, long client download κρατά connection. Συχνά είναι καλύτερο να δημιουργήσεις file ή object storage artifact asynchronously και μετά να το σερβίρεις.

Middleware δεν πρέπει να σπάει το body

Compression, checksum, metrics και error wrappers πρέπει να προωθούν `each`, `close` και όπου χρειάζεται `to_path`. Μετατροπή σε `body.each.to_a.join` καταργεί streaming, αλλάζει memory profile και μπορεί να μην κλείσει source σε exception. Το Rack 3 contract απαγορεύει σε middleware να καλεί αυθαίρετα `each`, εκτός από συγκεκριμένο `to_ary` path. Επιστρέφει wrapper body.

Content length είναι bytes, όχι Ruby character count. Αν αλλάξεις encoding ή compression μετά τον υπολογισμό, αφαίρεσέ το ή ξαναϋπολόγισέ το σωστά.

SOURCE TRACE

Στο `lib/sinatra/base.rb` του Sinatra 4.2.1 δες `Base#invoke`, `Response#body=`, `Response#finish`, `Helpers#body`, `Helpers#redirect` και `Helpers#attachment`. Για HEAD δες `Base.get` και `Rack::Head#call` στο Rack 3.2.7.

ΠΑΓΙΔΑ

Μετά το πρώτο body chunk δεν έχεις αξιόπιστο δεύτερο HTTP status. Σχεδίασε streaming failures ως interrupted representation και δώσε client resume ή re-fetch strategy.

ΜΗΧΑΝΙΣΜΟΣ

Route completion, header commit, body consumption και resource close είναι τέσσερα διαφορετικά events. Performance και correctness metrics πρέπει να λένε ποιο από αυτά μετρούν.

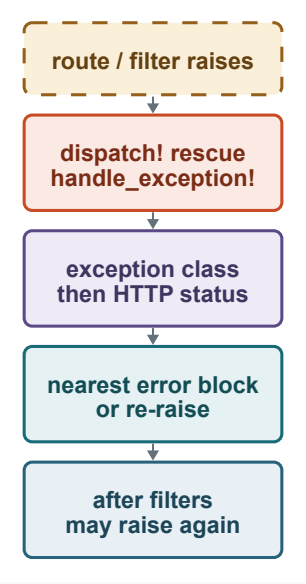
PROBE

Δοκίμασε String, array body, custom enumerable, 204, 304, HEAD και body που σηκώνει exception μετά από ένα chunk. Έλεγξε με Rack::Lint status, lowercase headers, content-length και close count. Μετά επανάλαβε πίσω από τον πραγματικό Puma handler.

ΚΕΦΑΛΑΙΟ 11

Exceptions, error handlers και after filters

Το error pipeline είναι συνδυασμός Sinatra settings, handler lookup, middleware order και του σημείου όπου συνέβη η αποτυχία.



Σχήμα 11.1 · Η exception μεταφράζεται πρώτα σε status και μετά αναζητά handler.

Το `dispatch!` τυλίγει params, static handling, before filters και routing σε rescue `Exception`. Σε αποτυχία καλεί `handle_exception!` μέσα από `invoke`. Το exception αποθηκεύεται στο `env["sinatra.error"]`, ώστε error handlers και middleware να μπορούν να το δουν.

Αν το error είναι `Sinatra::Error` με `http_status`, αυτό γίνεται αρχικό HTTP status. Διαφορετικά το status γίνεται 500. Για server errors, το Sinatra μπορεί να γράψει στο `rack.errors`, να ξανασηκώσει για `show_exceptions` ή να αναζητήσει error block, ανάλογα με settings και environment.

RUBY

```ruby
class InputError < StandardError
  attr_reader :code

  def initialize(code)
    @code = code
    super(code)
  end
end
```

```
class Api < Sinatra::Base
  set :show_exceptions, false
  set :raise_errors, false

  error InputError do
    failure = env.fetch("sinatra.error")
    content_type :json
    status 422
    JSON.generate(error: failure.code)
  end

  error do
    content_type :json
    status 500
    JSON.generate(error: "internal_error")
  end
end
```

Το public error δεν περιέχει exception message ή backtrace. Το internal event μπορεί να κρατήσει class, sanitized message, request id και causal ids με retention policy. Credentials, params και raw body δεν μπαίνουν αυτόματα.

Handler lookup έχει σειρά

Το Sinatra ψάχνει πρώτα handler για exception class και μετά handler για current status. Στο class hierarchy ψάχνει application class και ancestors. Για πολλά handlers στο ίδιο key, εξετάζει τα πιο πρόσφατα πρώτα. Αν δεν βρει class handler, ανεβαίνει στους exception superclasses.

Αυτό σημαίνει ότι generic `error StandardError` μπορεί να σκιάσει inherited policy. Κράτα μικρό αριθμό typed application errors και ένα catch-all. Μην μετατρέπεις adapter timeout, programmer bug και validation failure στο ίδιο 422 επειδή όλα κληρονομούν από `StandardError`.

`raise_errors` και `show_exceptions`

Στο test environment, `raise_errors` είναι true by default. Αυτό βοηθά tests να βλέπουν unhandled failures, αλλά catch-all error handler μπορεί να μη συμπεριφερθεί όπως στην production. Για response contract test ρύθμισε explicitly τα settings της test subclass. Για unit test άφησε το exception να ανέβει.

Σε development, `show_exceptions` είναι true και το outer ShowExceptions middleware μπορεί να εμφανίσει detail page. Με `:after_handler`, το custom handler τρέχει πριν από την debug presentation. Μην αφήσεις development mode σε public environment. Το debug response περιέχει source και request details.

After filters τρέχουν σε ensure

Μετά το main dispatch, το Sinatra τρέχει after filters σε ensure, εκτός αν σερβιρίστηκε static file. Άρα after filter τρέχει σε normal route, halt και handled exception. Αν after filter σηκώσει νέο exception χωρίς να υπάρχει ήδη `sinatra.error`, περνά ξανά από `handle_exception!`. Αν υπάρχει original error, η αρχική failure context παραμένει και το δεύτερο error δεν πρέπει να την αντικαταστήσει.

After filter δεν είναι reliable durable hook. Process kill μπορεί να το παραλείψει. Επίσης το body μπορεί να μην έχει καταναλωθεί ακόμη. Είναι καλό για headers ή μικρό in-process observation, όχι για billing, email ή guaranteed audit write.

```ruby
after do
  headers "x-request-id" => env.fetch("seatly.request_id")
end
```

Αν headers έχουν ήδη commit σε streaming path, αλλαγή αργότερα δεν θα φτάσει στο client. Το normal Sinatra stream συνήθως επιστρέφει πριν την κατανάλωση, οπότε ό,τι πρέπει να είναι header πρέπει να οριστεί πριν επιστραφεί το body.

404 έχει δύο σημασίες

Route miss σηκώνει `Sinatra::NotFound` όταν δεν υπάρχει downstream app. Το handler μπορεί επίσης να ενεργοποιηθεί αν response status είναι 404. Στο default path, το Sinatra προσθέτει `X-Cascade: pass` όταν `x_cascade` είναι ενεργό. Αυτό μπορεί να σημαίνει router fallback σε ορισμένα stacks. Ένα application-level missing record χρειάζεται να αποφασίσει αν θέλει cascade ή terminal 404.

```ruby
get "/events/:id" do |id|
  event = Events.find(id)
  halt 404, { "x-cascade" => "stop" }, ["not found"] unless event
  present(event)
end
```

Μη βασίζεσαι σε μη-standard header χωρίς να έχεις ελέγξει outer router. Σε πολλά deployments είναι απλώς response header.

Errors μετά το tuple

Exception μέσα σε lazy body δεν περνά από το `dispatch!` rescue, γιατί το route έχει ήδη επιστρέψει. Τη βλέπει server ή body-aware middleware. Αυτός είναι ο λόγος που χρειάζονται δύο error surfaces: dispatch failures και body consumption failures. Για streaming, καταγραφή στο body wrapper και cleanup στο close είναι υποχρεωτικά.

SOURCE TRACE
Στο Sinatra 4.2.1 δες `Base#dispatch!`, `#handle_exception!`, `#error_block!`, `#dump_errors!` και τα class methods `error` και `not_found`. Τα defaults για `raise_errors`, `dump_errors`, `show_exceptions` και `x_cascade` δηλώνονται κοντά στο τέλος του `lib/sinatra/base.rb`.

ΠΑΓΙΔΑ
Το after filter δεν είναι after-response και δεν είναι durable. Τρέχει πριν καταναλωθεί lazy body και μπορεί να μην τρέξει καθόλου σε hard process loss.

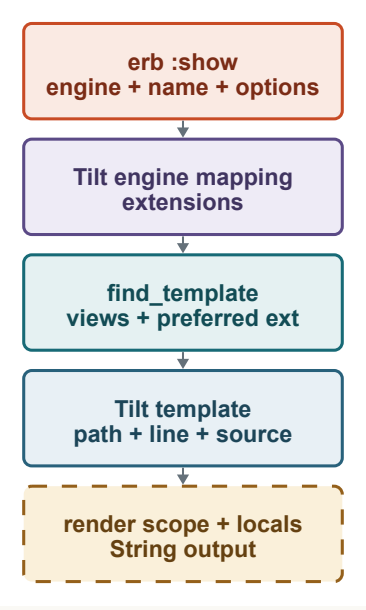
ΜΗΧΑΝΙΣΜΟΣ
Typed error mapping ανήκει στο HTTP boundary. Dispatch exceptions και body exceptions έχουν διαφορετικό capture path. Το public response και το internal diagnostic event χρειάζονται διαφορετικό schema.

PROBE
Φτιάξε failures σε before filter, route, typed error handler, after filter και δεύτερο body chunk. Τρέξε matrix με `raise_errors` true/false και `show_exceptions` false/true/`:after_handler`. Κατάγραψε status, handler, outer middleware trace και το αν είχαν ήδη σταλεί headers.

ΚΕΦΑΛΑΙΟ 12

Views, Tilt και template lookup

Το `erb :show` είναι pipeline από engine registry, lookup, compilation, scope και locals. Το filename είναι μόνο ένα μέρος του render shape.



Σχήμα 12.1 · Το Sinatra βρίσκει engine και source πριν το Tilt δημιουργήσει template.

Τα template helpers του Sinatra καταλήγουν στο private `render`. Εκεί συνδυάζονται application-level engine options με call options, επιλέγονται views directory, layout, layout engine, content type, scope και locals. Μετά το `compile_template` ζητά engine class από το Tilt και αποφασίζει αν το template είναι named Symbol, Proc ή literal String.

Το Symbol είναι lookup intent. Το Sinatra ψάχνει πρώτα preferred extension και μετά τις extensions που το Tilt έχει δηλώσει για το engine. Το String δεν είναι filename. Θεωρείται literal template source. Το Proc είναι block template. Η διάκριση είναι σημαντική για security και caching.

RUBY

```ruby
get "/events/:id" do |id|
  @event = Events.fetch(id)
  erb :event, layout: :application, locals: {
    request_id: env.fetch("seatly.request_id")
  }
end
```

Το template τρέχει by default με scope το request instance. Βλέπει helpers, instance variables και methods του app. Τα locals περνούν ξεχωριστά στο Tilt. Για reusable partials, locals κάνουν dependencies πιο ορατές από implicit instance variables. Για page template, λίγα intentional instance variables μπορούν να μείνουν πρακτικά.

Lookup δεν είναι authorization

Μην περνάς user input ως template Symbol ή view path. Ακόμη κι αν root lookup περιορίζει πού ψάχνει ο engine, ο client δεν πρέπει να επιλέγει server-side code ή presentation source.

```ruby
TEMPLATES = {
  "compact" => :event_compact,
  "full" => :event_full
}.freeze

get "/events/:id" do |id|
  template = TEMPLATES.fetch(params.fetch("view", "compact"))
  @event = Events.fetch(id)
  erb template
rescue KeyError
  halt 400, "unknown view"
end
```

Το allowlist είναι protocol mapping. Δεν κάνουμε `params[:view].to_sym`, γιατί αυτό μετατρέπει arbitrary input σε template identity και παλιότερες Rubies είχαν επιπλέον symbol-lifetime concerns.

Layout είναι δεύτερο render

Το Sinatra πρώτα κάνει render το content template. Έπειτα, αν υπάρχει layout, καλεί ξανά `render` με `layout: false` και δίνει το πρώτο output ως block. Το layout μπορεί να χρησιμοποιήσει `yield`. Το default layout είναι `:layout`, εκτός αν engine options ή call options το αλλάξουν.

Αυτό σημαίνει δύο template lookups και δύο render scopes. `layout_engine` μπορεί να διαφέρει από content engine, αλλά αυξάνει compatibility surface. Κράτα engine choices λίγες. Κάθε engine έχει δικό του escaping, compilation και error behavior.

```erb
<!doctype html>
<html lang="el">
  <head>
    <meta charset="utf-8">
    <title><%= Rack::Utils.escape_html(@title) %></title>
  </head>
  <body><%= yield %></body>
</html>
```

Plain ERB δεν κάνει automatic HTML escaping. Το `<%=` εισάγει το αποτέλεσμα ως έχει. Χρησιμοποίησε explicit escaping για untrusted text ή presentation layer που κάνει safe-by-default serialization. Μην θεωρείς ότι επειδή το Sinatra ενεργοποιεί XSS-related headers, το template output έχει καθαριστεί.

Content type και output object

Engine helpers μπορούν να δηλώσουν default content type. Το `render` μπορεί να επεκτείνει το returned String με μικρό module που κρατά content type. Στο τέλος του request, αν response δεν έχει ήδη type, το `call!` κοιτά το πρώτο body element για αυτό το metadata.

Για API endpoints, προτίμησε explicit `content_type :json`. Για HTML, το ERB helper και το default `text/html` είναι αρκετά, αλλά tests πρέπει να ελέγχουν charset. Content negotiation που διάλεξε representation πρέπει να προσθέτει `Vary` όταν υπάρχει shared cache.

Custom lookup με μικρό contract

Μπορείς να override `find_template` για πολλαπλά view roots, themes ή tenants. Η μέθοδος δεν επιστρέφει path. Κάνει yield κάθε candidate και το caller σταματά στο πρώτο existing file. Κάθε επιπλέον dimension μπαίνει στο cache shape. Αν tenant theme αλλάζει runtime χωρίς versioned path, μπορεί να σερβιριστεί compiled template άλλου tenant.

Κάνε lookup candidates deterministic και bounded. Βάλε tenant theme id σε allowlist, χρησιμοποίησε immutable release directories και μη σκανάρεις ολόκληρο filesystem ανά request. Συνήθως δύο explicit view roots αρκούν.

Inline templates μετά το `__END__` και named templates μέσω `template` είναι χρήσιμα για μικρά tools. Σε μεγάλη εφαρμογή κρύβουν source discovery και editor tooling. Δεν είναι performance feature. Η compiled μορφή έτσι κι αλλιώς περνά από το ίδιο cache.

SOURCE TRACE
Στο Sinatra 4.2.1 ακολούθησε τα `Sinatra::Templates` methods `render`, `compile_template`, `compile_block_template` και `find_template` στο `lib/sinatra/base.rb`. Το engine registry και το actual template object ανήκουν στο Tilt 2.7.0.

ΠΑΓΙΔΑ
Plain ERB δεν κάνει αυτόματο escaping. Rack::Protection headers μειώνουν ορισμένα browser risks, αλλά δεν μετατρέπουν untrusted HTML σε safe text.

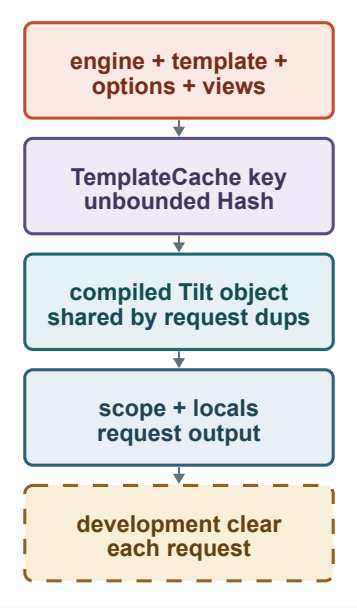
ΜΗΧΑΝΙΣΜΟΣ
Το render shape περιλαμβάνει engine, source identity, options, views, layout, scope και locals. Αν κάποια διάσταση αλλάζει το output, πρέπει να είναι ορατή στο lookup ή στο cache key.

PROBE
Κάνε override `find_template` μόνο σε test subclass και λογάρισε candidates. Δοκίμασε Symbol, literal String, named template, layout και custom views root. Μετά βάλε untrusted `<script>` σε title και επιβεβαίωσε ποιο helper κάνει escaping αντί να βασιστείς στην εμφάνιση του browser.

ΚΕΦΑΛΑΙΟ 13

Template cache, layouts και rendering safety

To compiled template cache είναι process-local, shared και unbounded. Δεν είναι fragment cache και δεν γνωρίζει τίποτα για freshness του domain data.



Σχήμα 13.1 · Το cache key οδηγεί σε compiled Tilt object που renderάρει ανά request.

Κάθε Sinatra application instance δημιουργεί `TemplateCache`, ένα μικρό wrapper γύρω από Hash. Το `fetch` χρησιμοποιεί array key και αποθηκεύει ακόμη και nil result του block. Η ίδια η source comment είναι σαφής: δεν είναι thread-safe, έχει unbounded size και τα key objects δεν αντιγράφονται defensively.

Το prototype instance δημιουργεί το cache. Τα request instances είναι shallow dups, άρα μοιράζονται το ίδιο cache reference. Σε production, όπου `reload_templates` είναι false, compiled templates επαναχρησιμοποιούνται. Σε development, το `call!` κάνει clear το cache στην αρχή κάθε request.

Τι αποτελεί cache key

Για named template, το `compile_template` καλεί `template_cache.fetch` με engine, template identity, options και views. Για literal String κάνει αντίστοιχο fetch με το ίδιο το source String. Mutable options ή views object που αλλάζει hash μετά την εισαγωγή μπορεί να κάνει το entry unreachable ή να δημιουργήσει απρόβλεπτο hit.

```ruby
class HtmlApp < Sinatra::Base
  configure do
    set :erb, escape_html: false
    set :reload_templates, false if production?
  end

  get "/about" do
    erb :about, layout: :application
  end
end
```

Το example κάνει την policy explicit, αλλά `escape_html` δεν είναι portable guarantee για κάθε ERB implementation. Το πραγματικό escaping behavior πρέπει να δοκιμαστεί με το active Tilt engine. Μην αντιγράφεις option από άλλο framework θεωρώντας ότι έχει ίδια σημασία.

Dynamic template source σημαίνει unbounded cache

Αν κάνεις `erb user_supplied_string`, κάθε διαφορετικό String γίνεται πιθανό cache key και executable template source. Αυτό είναι code injection και memory growth μαζί. Ακόμη και trusted dynamically generated source μπορεί να δημιουργήσει unbounded cardinality. Compile fixed templates στο deploy και πέρνα μεταβαλλόμενα values ως locals.

```ruby
CARD_TEMPLATE = <<~ERB.freeze
  <article class="event">
    <h2><%= Rack::Utils.escape_html(title) %></h2>
  </article>
ERB

get "/preview" do
  erb CARD_TEMPLATE, layout: false, locals: {
    title: params.fetch("title", "Preview")
  }
end
```

Ακόμη κι εδώ το literal source είναι constant. Για normal app, file template δίνει καλύτερο source location και deploy discipline.

Thread safety και first-hit races

Σε threaded server δύο πρώτα requests μπορεί να κάνουν ταυτόχρονα miss και να compile το ίδιο template. MRI προστατεύει internal Hash memory από corruption, όχι το higher-level check-then-write contract και όχι άλλες Ruby runtimes. Συνήθως το αποτέλεσμα είναι duplicate compilation, αλλά custom Tilt engine ή mutable options μπορούν να έχουν side effects.

Η απλή mitigation είναι warmup των known templates πριν ανοίξει traffic ή εξωτερικό application lock γύρω από controlled compilation. Μην ενεργοποιήσεις Sinatra `lock` για πάντα μόνο για template warmup, γιατί σειριοποιεί κάθε request. Με deploy που παράγει immutable release, pre-render representative pages σε health warmup και μετά δέξου traffic.

Το cache δεν έχει eviction. Αν template identity ή options έχουν request-level cardinality, το process memory μεγαλώνει μέχρι restart. Παρακολούθησε cache size μόνο ως diagnostic και διόρθωσε το key design. Μην προσθέσεις arbitrary LRU γύρω από private object χωρίς compatibility test.

Fragment caching είναι άλλο σύστημα

Compiled cache αποφεύγει parsing και code generation. Fragment cache αποθηκεύει rendered output. Χρειάζεται domain version, locale, authorization audience, representation και freshness policy. Αν βάλεις private page σε key χωρίς user dimension, έχεις data leak. Αν βάλεις raw user id, έχεις cardinality και privacy risk.

Καλύτερο key είναι bounded namespace με explicit version:

```ruby
def event_cache_key(event, locale:)
  ["event-card", 3, locale, event.id, event.version].join(":")
end
```

Το store adapter καθορίζει atomicity, TTL και race behavior. Sinatra δεν προσφέρει fragment caching contract. Η εφαρμογή πρέπει να ορίσει stampede policy και τι επιτρέπεται να είναι stale.

Rendering failure και partial response

File ERB render είναι eager: συνήθως ολοκληρώνει σε String πριν επιστραφεί το Rack tuple. Exception φτάνει στο Sinatra error pipeline. Αν όμως βάλεις render μέσα σε stream producer, μπορεί να συμβεί μετά τα headers. Τότε δεν υπάρχει καθαρό 500 response. Κράτα template render έξω από long-lived stream ή προετοίμασε chunks πριν το commit όταν χρειάζεσαι atomic HTML response.

Το layout είναι δεύτερο render και μπορεί να αποτύχει αφού το inner template έχει παραχθεί, αλλά πριν επιστραφεί body. Αυτό παραμένει dispatch error στο normal eager path.

SOURCE TRACE
Δες `Sinatra::TemplateCache`, `Base#initialize`, `Base#call`, `Base#call!` και `Templates#compile_template` στο `lib/sinatra/base.rb` του Sinatra 4.2.1. Η source comment του cache δηλώνει ρητά non-thread-safe, unbounded behavior και mutable-key limitation.

ΠΑΓΙΔΑ
Μη χρησιμοποιείς request input ως literal template source ή template name. Δημιουργεί execution risk, path ambiguity και unbounded compiled cache.

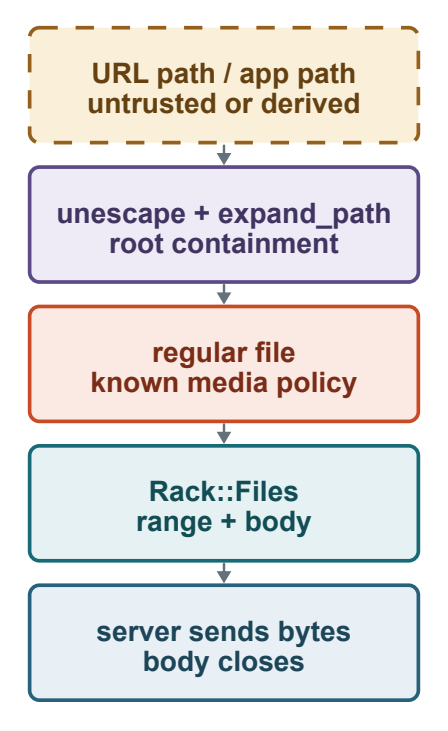
ΜΗΧΑΝΙΣΜΟΣ
Compiled template cache, rendered fragment cache και HTTP cache έχουν διαφορετικό owner, key και freshness model. Το κοινό όνομα cache δεν τα κάνει ίδιο mechanism.

PROBE
Σε staging, άδειασε το compiled cache, στείλε 20 concurrent first requests και μέτρησε compile count, latency και allocations. Μετά κάνε 1.000 renders με διαφορετικό literal source και παρατήρησε cache size. Το δεύτερο probe πρέπει να μείνει isolated και bounded, όχι σε public endpoint.

ΚΕΦΑΛΑΙΟ 14

Static files, send_file και path boundaries

Το filesystem path είναι capability. Όποιος μπορεί να το επηρεάσει μπορεί να διαλέξει ποια bytes θα διαβάσει ο process.



Σχήμα 14.1 · Το file path περνά normalization και containment πριν γίνει Rack body.

Για GET και HEAD, το `dispatch!` δοκιμάζει `static!` πριν από τα before filters. Το method ενώνει `public_folder` και unescaped `path_info`, ελέγχει valid path, κάνει `expand_path` και απαιτεί το αποτέλεσμα να ξεκινά μέσα στο expanded public directory. Μετά βεβαιώνει ότι είναι regular file και καλεί `send_file`.

Άρα τα application before και after filters δεν τρέχουν για built-in static response. Τα Rack middleware εξακολουθούν να περιβάλλουν το endpoint και βλέπουν το response. Security header που υπάρχει μόνο σε after filter θα λείπει από static assets. Βάλ' το σε outer middleware, `static_headers` ή στον reverse proxy που σερβίρει assets.

```ruby
class WebApp < Sinatra::Base
  set :public_folder, File.expand_path("../public", __dir__)
  set :static_cache_control, [:public, { max_age: 86_400 }]
  set :static_headers, {
    "x-content-type-options" => "nosniff"
  }
end
```

Fingerprint assets μπορούν να πάρουν μεγάλο immutable max-age. Αρχεία με stable URL που αλλάζουν περιεχόμενο χρειάζονται μικρότερη freshness policy. Το `static_cache_control` δεν κάνει fingerprinting.

`send_file` εμπιστεύεται περισσότερο τον caller

Το helper `send_file(path)` ορίζει content type, optional disposition και last modified, μετά καλεί `Rack::Files#serving` με το συγκεκριμένο path. Σε αντίθεση με `static!`, δεν εφαρμόζει από μόνο του containment στο public root. Αν το path προκύπτει από params, έχεις arbitrary file read.

```ruby
EXPORT_ROOT = File.expand_path("../exports", __dir__).freeze

def export_path(id)
  candidate = File.expand_path("#{Integer(id, 10)}.csv", EXPORT_ROOT)
  prefix = EXPORT_ROOT + File::SEPARATOR
  halt 404 unless candidate.start_with?(prefix)
  halt 404 unless File.file?(candidate)
  candidate
end

get "/exports/:id" do |id|
  send_file export_path(id), type: "text/csv", disposition: :attachment
end
```

Ακόμη καλύτερα, resolve id μέσω database record που κρατά opaque storage key, όχι filename από client. Το containment check πρέπει να λάβει υπόψη symlinks αν attacker ή άλλο subsystem μπορεί να τα δημιουργήσει μέσα στο root. Για ισχυρό boundary χρησιμοποίησε owned directory χωρίς untrusted filesystem writes και, όπου απαιτείται, `realpath` μετά την ύπαρξη του file.

Range requests και body lifetime

Το Rack 3.2.7 `Rack::Files#serving` υποστηρίζει GET, HEAD, OPTIONS και byte ranges. Για normal file επιστρέφει iterator με `to_path`, ώστε server να μπορεί να χρησιμοποιήσει πιο αποδοτικό transport. Για range δημιουργεί iterator που διαβάζει συγκεκριμένα segments και υπολογίζει `content-range`.

Μην τυλίγεις file body σε middleware που το κάνει join. Χάνεις `to_path`, φορτώνεις ολόκληρο file στη Ruby heap και μπορεί να σπάσεις range semantics. Compression για ήδη συμπιεσμένα media συχνά προσθέτει CPU χωρίς όφελος.

Content type από extension είναι hint. Upload που ονομάστηκε `.png` δεν έγινε ασφαλής εικόνα. Για user uploads, αποθήκευσε έξω από executable public tree, χρησιμοποίησε generated storage key, επέβαλε download content type και `Content-Disposition` σύμφωνα με product policy. Active content όπως HTML ή SVG μπορεί να εκτελεστεί στο origin αν σερβιριστεί inline.

Upload και download είναι διαφορετικά contracts

Upload path χρειάζεται size limit, streaming write, digest, type validation, malware policy και terminal state. Download path χρειάζεται authorization, version, range policy και cache headers. Το ότι και τα δύο χρησιμοποιούν file δεν σημαίνει ότι μοιράζονται controller helper.

Public file response μετά από database authorization μπορεί να έχει time-of-check/time-of-use race αν file αντικατασταθεί. Immutable object keys και append-only artifacts απλοποιούν το model. Αν επιτρέπεται overwrite, δέσε response validator και storage version στο authorization decision.

Offload με ελεγχόμενο boundary

Σε production, reverse proxy ή object storage συχνά σερβίρει μεγάλα files πιο αποδοτικά. Η Sinatra route μπορεί να authorize και να επιστρέψει internal redirect header ή signed short-lived URL. Μην αφήνεις client να διαλέξει το internal path. Το mapping id -> internal location γίνεται server-side και το proxy πρέπει να αγνοεί spoofed version του internal header από upstream client.

Signed URL είναι bearer capability. Βάλε μικρό expiry, fixed resource key, disposition και, όταν χρειάζεται, audience. Μην το logάρεις ολόκληρο.

SOURCE TRACE
Στο Sinatra 4.2.1 δες `Base#dispatch!`, `#static!`, `Helpers#send_file` και `Helpers#attachment`. Το static containment γίνεται μόνο στο `static!`. Για range, iterator και `to_path` δες `Rack::Files` στο Rack 3.2.7.

ΠΑΓΙΔΑ
`send_file params[:path]` είναι arbitrary file read. Το helper δεν περιορίζει το path στο public folder. Resolve opaque id σε owned path και κάνε explicit containment.

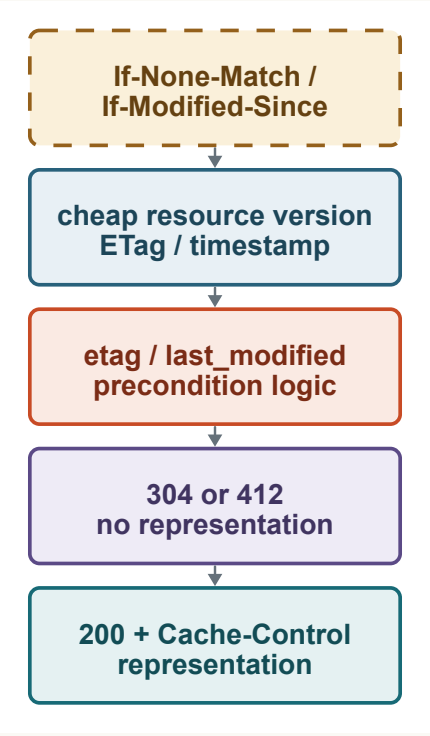
ΜΗΧΑΝΙΣΜΟΣ
Path choice, file ownership, representation type και authorization είναι ξεχωριστές αποφάσεις. Το Rack μπορεί να μεταφέρει bytes σωστά χωρίς να ξέρει αν επιτρεπόταν να τα διαβάσει.

PROBE
Δοκίμασε encoded `..`, duplicate slashes, null-like encodings, symlink μέσα στο export root, missing file, HEAD και single/multiple byte ranges. Επιβεβαίωσε ποια middleware και filters τρέχουν για built-in static response και ποια security headers φτάνουν στον client.

ΚΕΦΑΛΑΙΟ 15

HTTP caching και conditional requests

O validator είναι version contract. Επιτρέπει στον client να αποδείξει ότι η representation που έχει παραμένει current χωρίς να τη μεταφέρεις ξανά.



Σχήμα 15.1 · Οι preconditions αποφασίζονται πριν από το ακριβό render.

Το Sinatra προσφέρει `cache_control`, `expires`, `last_modified` και `etag`. Δεν αποθηκεύει response σε cache. Παράγει headers και εφαρμόζει conditional request logic. Ο browser, CDN ή reverse proxy αποφασίζει αν και πώς θα κρατήσει representation.

Το σωστό order είναι να φορτώσεις το φθηνότερο authoritative version, να καλέσεις validators και μόνο μετά να κάνεις ακριβό render ή downstream fetch.

```ruby
get "/events/:id" do |id|
  event = Events.fetch(id)
  etag "event:#{event.id}:#{event.version}"
  last_modified event.updated_at
  cache_control :public, max_age: 60
  content_type :json
  JSON.generate(event.to_h)
end
```

Αν `If-None-Match` ταιριάξει σε safe request, το `etag` κάνει halt 304. Για unsafe request μπορεί να δώσει 412. Αν υπάρχει `If-Match` και δεν ταιριάζει, δίνει 412. Το `last_modified` δεν εφαρμόζει `If-Modified-Since` όταν υπάρχει `If-None-Match`, ώστε ο ETag να έχει precedence.

Strong και weak identity

Strong ETag σημαίνει ότι δύο representations είναι byte-equivalent για τις σχετικές semantics. Weak ETag δηλώνει semantic equivalence χωρίς απαραίτητα ίδια bytes. Αν JSON serializer αλλάζει whitespace ή key order, database `updated_at` ίσως είναι domain version αλλά όχι byte version.

Για API optimistic concurrency, χρησιμοποίησε validator που αντιστοιχεί στην state version που προστατεύει το write. Το `If-Match` πρέπει να ελέγχεται στην ίδια transaction με το update ή να μεταφράζεται σε conditional SQL. Check έξω και write μέσα έχει race.

```ruby
put "/events/:id" do |id|
  expected = request.get_header("HTTP_IF_MATCH").to_s
  command = UpdateEventInput.from_json(id: id, body: request.body)
  result = Events.update_if_match(command, expected_etag: expected)
  halt 412 unless result.updated?
  etag result.etag
  status 204
end
```

Το repository contract πρέπει να κάνει compare-and-write atomically. Το route δεν αρκεί να καλέσει `etag` πάνω σε stale object και μετά ανεξάρτητο save.

Cache key dimensions

Αν response αλλάζει με `Accept`, `Accept-Language` ή encoding, shared caches χρειάζονται σωστό `Vary`. Αν αλλάζει ανά authenticated user, `private` ή `no-store` είναι συχνά ασφαλέστερο από τεράστιο Vary. Cookies σε request μπορούν να κάνουν CDN policy απρόβλεπτη. Μην βάζεις public caching σε route επειδή το body δεν φαίνεται sensitive σε ένα test.

```ruby
before "/account/*" do
  cache_control :private, :no_store
end
```

`no-cache` δεν σημαίνει πάντα μην αποθηκεύσεις. Σημαίνει revalidate πριν από reuse. `no-store` ζητά να μη γίνει storage. `max-age` είναι freshness lifetime. Γράψε policy με βάση data sensitivity και staleness budget, όχι με βάση τα ονόματα που ακούγονται παρόμοια.

304 δεν περιέχει representation

Όταν helper κάνει halt 304, το `Sinatra::Response#finish` αφαιρεί body, content-type και content-length. Ο client επαναχρησιμοποιεί το cached body. Αν το response είχε διαφορετικό authorization ή representation dimension που δεν ήταν στο cache key, το 304 μπορεί να επικυρώσει λάθος bytes.

Ένα ETag δεν πρέπει να περιέχει sensitive raw value. Είναι visible header και μπορεί να logαριστεί. Hash χωρίς secret μπορεί επίσης να επιτρέπει guessing σε μικρό domain. Προτίμησε opaque version token με bounded length.

Stampede και stale policy

HTTP caching μειώνει load μόνο αν intermediaries επιτρέπεται να μοιράζονται responses. Όταν entry λήξει, πολλοί callers μπορεί να κάνουν origin request μαζί. CDN stale-while-revalidate ή application lease μπορεί να περιορίσει stampede. Η policy πρέπει να λέει πόσο stale επιτρέπεται και ποιο invariant δεν πρέπει ποτέ να σερβιριστεί stale.

Μη χρησιμοποιείς stale cache για authorization, balance ή one-time token. Μπορεί να είναι σωστό για public catalog ή rendered article. Το ίδιο store δεν σημαίνει ίδια freshness policy.

Σε rolling release, παλιός και νέος process μπορεί να παράγουν διαφορετικά bytes για την ίδια domain version. Βάλε representation schema ή renderer version μέσα στον validator όταν η αλλαγή δεν είναι byte-compatible. Αλλιώς ένας client μπορεί να πάρει 304 από νέο process για body που δημιούργησε η παλιά έκδοση με διαφορετική σημασία.

SOURCE TRACE
Στο Sinatra 4.2.1 δες `Helpers#cache_control`, `#expires`, `#last_modified`, `#etag` και `#etag_matches?` στο `lib/sinatra/base.rb`. Το final removal body και content headers για 304 γίνεται στο `Sinatra::Response#finish`.

ΠΑΓΙΔΑ
Validator check έξω από την write transaction δεν προστατεύει optimistic update. Χρειάζεται atomic compare-and-write στο durable owner της version.

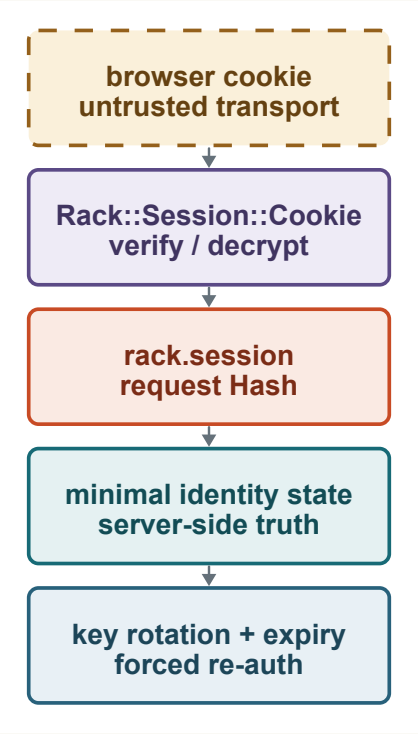
ΜΗΧΑΝΙΣΜΟΣ
Cache storage, freshness, validation και write preconditions είναι διαφορετικά mechanisms. Ένας ETag έχει αξία μόνο αν η version identity του είναι σωστή.

PROBE
Γράψε matrix για GET και PUT με absent, matching και stale `If-None-Match`, `If-Match` και dates. Έλεγξε 200, 304, 412, body presence, content headers και Vary. Τρέξε δύο concurrent conditional updates και assert ότι μόνο ένα κάνει durable transition.

ΚΕΦΑΛΑΙΟ 16

Sessions, cookies και secret rotation

Cookie session είναι client-carried encrypted state. Έχει size, rotation, revocation και compatibility constraints που δεν φαίνονται στο Hash API.



Σχήμα 16.1 · Το cookie αποκωδικοποιείται σε request session και επιστρέφει versioned.

Με `enable :sessions`, το Sinatra βάζει το configured session middleware πριν από Rack::Protection. Το default store είναι `Rack::Session::Cookie`. Το session data ταξιδεύει στον browser και επιστρέφει σε κάθε request. Στην τρέχουσα rack-session 2.1.2, configured secrets ενεργοποιούν authenticated encryption. Το πρώτο secret γράφει νέα cookies και η λίστα secrets δοκιμάζεται κατά το decrypt, άρα υποστηρίζει rotation.

Το Sinatra δημιουργεί random `session_secret` στο class load αν δεν δώσεις δικό σου. Αυτό είναι ασφαλές ως randomness, αλλά αλλάζει ανά process boot και διαφέρει ανά instance. Σε cluster οι users θα φαίνονται logged out τυχαία. Production secret πρέπει να είναι κοινό, stable και τουλάχιστον 64 bytes όπως ζητά το rack-session encryptor contract.

```ruby
current = ENV.fetch("SESSION_SECRET_CURRENT")
previous = ENV["SESSION_SECRET_PREVIOUS"]
secrets = [current, previous].compact.freeze

class WebApp < Sinatra::Base
  set :session_secret, current
  set :sessions, {
    key: "seatly.session",
    secrets: secrets,
    httponly: true,
    secure: production?,
    same_site: :lax,
    expire_after: 86_400
  }
end
```

Το example διαβάζει values χωρίς να τα τυπώνει. Σε πραγματικό project το composition code πρέπει να επιβεβαιώνει length και presence, όχι να κάνει fallback σε γνωστό string.

Rotation χωρίς μαζικό logout

Βήμα 1: deploy `[new, old]` σε όλους τους processes. Νέα responses γράφουν με new, παλιά cookies διαβάζονται με old. Περίμενε τουλάχιστον το μέγιστο session lifetime και το deployment overlap. Βήμα 2: αφαίρεσε old. Αν υπάρχει rollback πιθανότητα, η προηγούμενη έκδοση πρέπει επίσης να μπορεί να διαβάσει cookies που γράφτηκαν με new ή το rollback θα κάνει logout.

Secret rotation δεν ανακαλεί επιλεγμένο session. Client-carried state δεν έχει server-side row για revoke, εκτός αν αποθηκεύσεις version ή denylist σε server-side store. Για high-risk session, κράτα opaque session id στον cookie και authoritative state server-side.

Μικρό payload, μικρό blast radius

Το rack-session cookie έχει πρακτικό όριο περίπου 4 KB μαζί με overhead. Αν το encoded data ξεπεράσει το όριο, write μπορεί να αποτύχει και να γραφτεί warning στο `rack.errors`. Μην αποθηκεύεις model objects, permissions list ή large flash data. Κράτα subject id, authentication time, session version και λίγα bounded flags.

Encryption κρύβει content από client και authentication αποτρέπει modification χωρίς key. Δεν εγγυάται freshness. User role που άλλαξε στη database δεν ενημερώνεται επειδή παλιό cookie είναι cryptographically valid. Φόρτωσε authoritative authorization state ή βάλε revocable version.

Cookie deserialization είναι deployed protocol. Αλλαγή serializer ή field shape πρέπει να δουλεύει σε mixed release. H current Rack encryptor μπορεί να χρησιμοποιήσει JSON serialization. Primitive JSON shapes είναι ευκολότερα για compatibility από arbitrary Ruby objects.

Cookie attributes είναι security policy

`Secure` απαιτεί HTTPS transport. `HttpOnly` εμποδίζει normal JavaScript access, όχι κάθε browser attack. `SameSite=Lax` μειώνει πολλά cross-site sends, αλλά δεν αντικαθιστά CSRF token για state-changing browser flows. Domain-wide cookie μοιράζεται με subdomains και μεγαλώνει το trust boundary. Host-only cookie είναι ασφαλέστερο default.

Όταν γίνεται login ή privilege elevation, καθάρισε παλιό state και ζήτησε session renewal σύμφωνα με το store API. Στο Rack request μπορείς να θέσεις `request.session_options[:renew] = true`. Logout πρέπει να κάνει clear/destroy και να μην αφήνει reusable bearer data σε άλλο cookie.

RUBY

```ruby
post "/login" do
  principal = Login.authenticate(json_document)
  halt 401 unless principal

  session.clear
  request.session_options[:renew] = true
  session[:subject_id] = principal.id
  session[:auth_version] = principal.auth_version
  status 204
end
```

Μην αποθηκεύεις password ή access token τρίτου provider στο session cookie.

Sessions outside Sinatra

Αν βάλεις session middleware στο `config.ru` χωρίς `enable :sessions`, το Sinatra δεν ενεργοποιεί αυτόματα session-based Rack::Protection. Πρέπει να δηλώσεις `set :protection, session: true` ή να συνθέσεις τα συγκεκριμένα protection middleware εξωτερικά. Η σειρά έχει σημασία: session πρέπει να είναι outer από protection που διαβάζει `rack.session`.

SOURCE TRACE
Στο Sinatra 4.2.1 δες `setup_sessions`, `setup_protection`, τα settings `sessions`, `session_store` και `session_secret`. Για encryption, secret list, 4 KB guard και rotation δες `Rack::Session::Cookie` 2.1.2 και `Rack::Session::Encryptor`.

ΠΑΓΙΔΑ
Το auto-generated Sinatra secret αλλάζει ανά boot και process. Είναι καλό random fallback για development, όχι shared production identity.

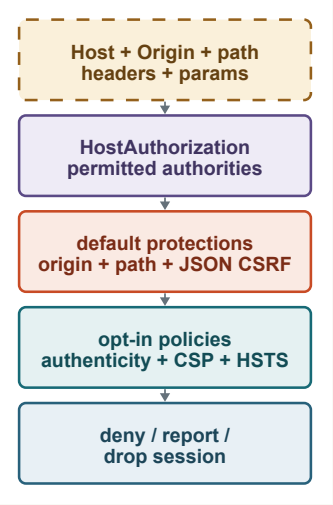
ΜΗΧΑΝΙΣΜΟΣ
Cookie session είναι encrypted client-carried cache της identity state. Authorization truth, selective revocation και compatibility χρειάζονται explicit server-side design.

PROBE
Ξεκίνα δύο app instances με διαφορετικά secrets και κάνε alternating requests με το ίδιο cookie. Μετά δοκίμασε rotation `[new, old]`, rollback reader και payload κοντά στο 4 KB. Έλεγξε Set-Cookie attributes χωρίς να καταγράψεις τα secret values ή ολόκληρο το cookie.

ΚΕΦΑΛΑΙΟ 17

Rack::Protection, Host και browser defenses

To Rack::Protection είναι σύνολο συγκεκριμένων middleware. Δεν είναι γενικό σήμα ότι η εφαρμογή είναι ασφαλής.



Σχήμα 17.1 · Το request περνά Host policy, default protections και opt-in defenses.

Με `protection` ενεργό, το Sinatra χτίζει `Rack::Protection` πριν από το HostAuthorization. Στο Rack::Protection 4.2.1, default-on layers είναι FrameOptions, HttpOrigin, IPSpoofing, JsonCsrf, PathTraversal, RemoteToken και XSSHeader, εκτός αν τα εξαιρέσεις. AuthenticityToken, ContentSecurityPolicy, CookieTossing, EscapedParams, FormToken, ReferrerPolicy, RemoteReferrer, SessionHijacking και StrictTransport είναι off by default και ενεργοποιούνται με `use` option.

Αυτός ο κατάλογος είναι version-specific. Μην γράψεις security claim `protection enabled` χωρίς να καταγράφεις ποια layers και options υπάρχουν στη δική σου έκδοση.

HostAuthorization χρειάζεται production allowlist

Το Sinatra προσθέτει `Rack::Protection::HostAuthorization` ανεξάρτητα από το γενικό protection toggle. Σε development επιτρέπει localhost, `.localhost`, `.test` και IP addresses. Σε άλλα environments το default options hash είναι κενό. Στο middleware, κενή `permitted_hosts` επιτρέπει οποιοδήποτε host.

```ruby
class PublicApp < Sinatra::Base
  set :host_authorization, {
    permitted_hosts: ["app.example.com", ".internal.example.com"]
  }
end
```

Leading dot επιτρέπει subdomains σύμφωνα με το Rack::Protection matcher. Μην βάζεις broad suffix αν untrusted parties μπορούν να αποκτήσουν subdomain. Το middleware ελέγχει origin Host και forwarded authority. Πίσω από proxy, η proxy normalization και trust policy πρέπει να συμφωνούν με την allowlist.

Host header επηρεάζει absolute URLs, password reset links και redirects. Το block δεν είναι μόνο DNS rebinding protection. Είναι origin construction boundary.

Reaction αλλάζει το failure contract

Το base protection middleware μπορεί να deny, report ή drop session ανάλογα με `reaction`. Το Sinatra επιλέγει `:drop_session` by default στο combined protection setup. Αν δεν υπάρχει session, το reaction μπορεί να μην επιστρέψει response και η request να συνεχίσει, ανάλογα με το συγκεκριμένο layer και options. Για critical policy προτίμησε explicit deny semantics και test το actual stack.

```ruby
set :protection, {
  reaction: :deny,
  session: true,
  use: [:authenticity_token, :content_security_policy, :referrer_policy],
  script_src: "'self'",
  style_src: "'self'",
  frame_ancestors: "'none'",
  referrer_policy: "strict-origin-when-cross-origin"
}
```

Το options hash περνά στα selected middleware. Επιβεβαίωσε τα supported option names στη pinned έκδοση. CSP που μπλοκάρει inline scripts μπορεί να σπάσει το UI, άρα ξεκίνα με report-only όταν αλλάζεις υπάρχον app και μέτρα violations με bounded reporting.

CSRF layers δεν είναι ένα πράγμα

`HttpOrigin`, `JsonCsrf` και `RemoteToken` καλύπτουν συγκεκριμένα cross-origin patterns. Το masked `AuthenticityToken` είναι opt-in. Με sessions, μπορείς να παράγεις token για form και να το στείλεις ως hidden field ή `X-CSRF-Token`.

```ruby
helpers do
  def csrf_token(path:, method:)
    Rack::Protection::AuthenticityToken.token(
      session,
      path: path,
      method: method
    )
  end
end
```

Per-form token δένεται σε normalized path και method. Form action, mount point και trailing slash πρέπει να συμφωνούν. Bearer-token API που δεν χρησιμοποιεί ambient cookies έχει άλλο CSRF model, αλλά CORS, token storage και XSS παραμένουν. Μην απενεργοποιείς layers συνολικά επειδή ένα endpoint είναι JSON.

Headers δεν κάνουν escaping

Frame options, CSP, nosniff και referrer policy περιορίζουν browser behavior. Δεν κάνουν HTML escape, SQL binding, authorization ή output validation. PathTraversal καθαρίζει route path patterns, όχι arbitrary path που περνάς σε `send_file`. IPSpoofing δεν αποφασίζει ποιο proxy εμπιστεύεσαι για business geolocation ή rate limits.

StrictTransport βάζει HSTS, αλλά μόνο σωστό HTTPS deployment πρέπει να το ενεργοποιεί. Με includeSubDomains ή preload μπορείς να επηρεάσεις άλλα hosts. Αυτό είναι domain ownership decision, όχι local header tweak.

Instrumentation και tests

Κάθε protection μπορεί να λάβει instrumenter. Σε failure γράφει attack name στο env και καλεί `instrument("rack.protection", env)`. Μην εξάγεις ολόκληρο env. Κράτα protection type, route class, status και bounded authority. Τα attack attempts μπορούν να έχουν υψηλή cardinality.

Γράψε negative tests για encoded paths, hostile Host και Forwarded, cross-site form, JSON CSRF shape, iframe embedding και content type sniffing. Security header snapshot μόνο στο happy path δεν αποδεικνύει blocking behavior.

SOURCE TRACE
Η σύνθεση γίνεται στα `Base.setup_protection` και `Base.setup_host_authorization` του Sinatra 4.2.1. Η ακριβής default-on και opt-in λίστα βρίσκεται στο `rack-protection/lib/rack/protection.rb` 4.2.1. Για reactions δες `Rack::Protection::Base` και για hosts το `HostAuthorization#accepts?`.

ΠΑΓΙΔΑ
Σε production το default `host_authorization` options hash είναι κενό και το middleware επιτρέπει κάθε host. Βάλε explicit allowlist όταν το Host επηρεάζει origin ή routing.

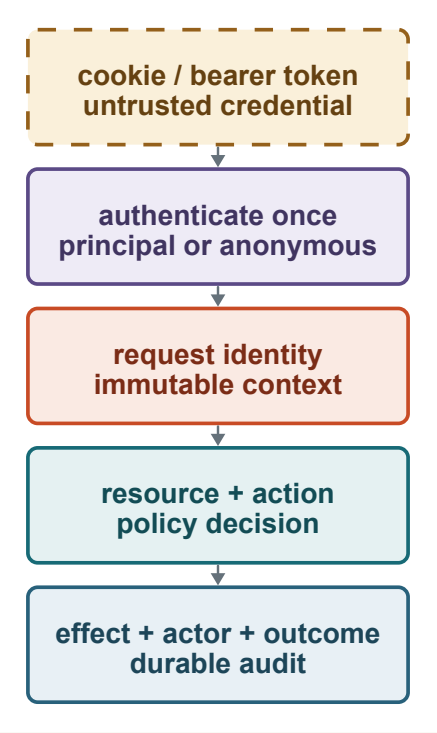
ΜΗΧΑΝΙΣΜΟΣ
Security middleware είναι layered policy με συγκεκριμένη threat model, ordering και reaction. Κανένα boolean setting δεν αντικαθιστά route-level validation και authorization.

PROBE
Εξήγαγε στο boot τη λίστα protection middleware και options χωρίς secrets. Στείλε hostile Host, Forwarded host, cross-origin form και JSON request. Για κάθε case κατέγραψε ποιο layer αντέδρασε, status, session effect και αν το endpoint εκτελέστηκε.

ΚΕΦΑΛΑΙΟ 18

Authentication και authorization χωρίς framework magic

Authentication παράγει principal. Authorization αποφασίζει αν αυτός ο principal μπορεί να κάνει συγκεκριμένη action πάνω σε συγκεκριμένο resource.



Σχήμα 18.1 · Το credential γίνεται immutable identity πριν από resource policy.

Το Sinatra δεν επιβάλλει authentication stack. Αυτό σε αναγκάζει να ξεχωρίσεις τα contracts. Credential extraction ανήκει στο HTTP edge. Verification ανήκει σε authenticator που ξέρει key material, issuer ή session store. Το αποτέλεσμα είναι immutable principal ή anonymous. Authorization ανήκει κοντά στο use case και παίρνει principal, action και resource facts.

```ruby
Principal = Data.define(:id, :roles, :auth_version) do
  def operator?
    roles.include?("operator")
  end
end

class Authentication
  def initialize(app, authenticator:)
    @app = app
    @authenticator = authenticator
  end

  def call(env)
    credential = env["HTTP_AUTHORIZATION"].to_s
    env["seatly.principal"] = @authenticator.call(credential)
    @app.call(env)
  end
end
```

Το middleware κάνει verification μία φορά. Ο authenticator πρέπει να επιστρέφει nil για absent credential και typed failure για malformed, expired ή invalid credential ώστε το outer policy να αποφασίζει 401 χωρίς να αποκαλύπτει λεπτομέρειες.

401, 403 και 404

401 σημαίνει ότι απαιτείται valid authentication και συνήθως συνοδεύεται από `WWW-Authenticate` για το scheme. 403 σημαίνει authenticated ή otherwise understood caller που δεν επιτρέπεται. 404 μπορεί να χρησιμοποιηθεί για να μη διαρρεύσει ύπαρξη resource, αλλά πρέπει να είναι consistent ώστε timing και error shape να μην αποκαλύπτουν τη διαφορά.

```ruby
get "/accounts/:account_id/reservations/:id" do |account_id, id|
  principal = env["seatly.principal"]
  halt 401, { "www-authenticate" => "Bearer" }, [] unless principal

  reservation = Reservations.find(account_id: account_id, id: id)
  halt 404 unless reservation
  halt 404 unless ReservationPolicy.read?(principal, reservation)

  present(reservation)
end
```

Το repository query περιλαμβάνει account scope για defense in depth και λιγότερη accidental cross-tenant exposure. Η policy παραμένει explicit.

Filters για coarse policy

Before filter μπορεί να απαιτήσει authentication για path family. Μην βάλεις όλη την authorization σε regex paths. Το route μπορεί να αλλάξει ή να κάνει mount κάτω από νέο prefix. Resource ownership δεν εκφράζεται αξιόπιστα μόνο με URL.

```ruby
before "/api/*" do
  next if request.path_info == "/api/public-key"

  halt 401, { "www-authenticate" => "Bearer" }, [] unless
    env["seatly.principal"]
end
```

Το `next` βγαίνει από filter block, δεν είναι `pass`. Κράτα allowlisted public routes μικρά και δοκίμασε ότι νέο route δεν γίνεται public κατά λάθος.

Session identity και revocation

Session cookie μπορεί να κρατά subject id και auth version. Σε κάθε sensitive request, φόρτωσε current user state ή χρησιμοποίησε short bounded cache. Αν password reset, account disable ή role change αυξάνει `auth_version`, παλιό session απορρίπτεται. Αυτό δίνει selective revocation χωρίς global secret rotation.

Bearer token χρειάζεται signature verification, allowed algorithm, issuer, audience, expiry και clock skew policy. Μην επιλέγεις algorithm από untrusted header χωρίς allowlist. Key rotation χρειάζεται key id με bounded lookup και old/new overlap. Token claims είναι assertions του issuer, όχι hydrated domain object.

Password και login boundary

Password verification πρέπει να χρησιμοποιεί password hashing library με κατάλληλο cost και constant-time verification path. Μην υλοποιείς crypto μέσα στο Sinatra route. Rate limiting χρειάζεται identity που υπάρχει πριν από login, όπως normalized account key και network signal, αλλά πρέπει να αποφεύγει user enumeration. Lockout policy να μην επιτρέπει attacker να κλειδώσει μαζικά λογαριασμούς.

Με successful login, ανανέωσε session id, καθάρισε προηγούμενα state και κατάγραψε authentication event χωρίς credential. Με logout, invalidation πρέπει να ταιριάζει στο store model. Διαγραφή browser cookie δεν ανακαλεί server-side bearer token που έχει αντιγραφεί αλλού.

Audit ως durable effect

Authorization denial μπορεί να είναι metric. Privileged state change χρειάζεται durable audit record στην ίδια transaction με το domain change ή μέσω outbox. Record actor id, effective subject, action, aggregate id, version και outcome. Μην αποθηκεύεις raw token, password ή full request.

Impersonation χρειάζεται δύο identities: actor και effective subject. Αν αντικαταστήσεις απλώς current user, το audit ψεύδεται. Expiry, reason και visible UI indicator είναι μέρος του contract.

Authorization TOCTOU

Αν φορτώσεις resource, κάνεις policy check και μετά update χωρίς version ή transaction, η state μπορεί να αλλάξει ενδιάμεσα. Για permission που εξαρτάται από current ownership ή status, check και write πρέπει να μοιράζονται durable boundary ή conditional update.

SOURCE TRACE
Το Sinatra παρέχει filters, request env, sessions και halt, όχι identity framework. Δες `Base#filter!`, `Helpers#session` και middleware composition στο `lib/sinatra/base.rb` 4.2.1. Το placement authentication πριν από routes είναι application design πάνω στο Rack contract.

ΠΑΓΙΔΑ
Μην αποθηκεύεις role list σε long-lived cookie και τη θεωρείς authorization truth. Είναι snapshot που παραμένει valid μέχρι expiry ακόμη κι αν τα rights άλλαξαν.

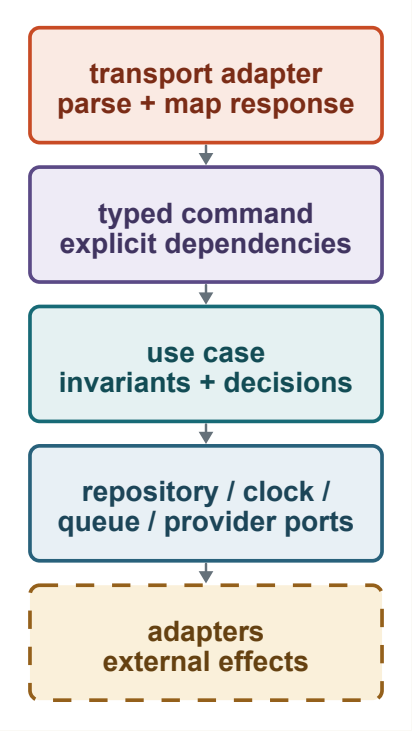
ΜΗΧΑΝΙΣΜΟΣ
Credential, principal, policy decision και audit record είναι τέσσερα διαφορετικά artifacts. Κάθε ένα έχει δικό του owner, lifetime και failure mode.

PROBE
Γράψε matrix absent, malformed, expired και valid credential, disabled user, changed role, cross-tenant id και concurrent ownership change. Assert 401/403/404 policy, `WWW-Authenticate`, audit identity και ότι denial δεν εκτελεί κανένα domain effect.

ΚΕΦΑΛΑΙΟ 19

Modular architecture και dependency ownership

Modular Sinatra σημαίνει composable Rack app. Application architecture σημαίνει ότι κάθε απόφαση και dependency έχει σαφή owner.



Σχήμα 19.1 · Το route μεταφράζει HTTP και το use case μιλά μέσω μικρών ports.

Η subclass του `Sinatra::Base` είναι modular ως Rack component. Αυτό δεν σημαίνει ότι business logic έγινε modular. Αν routes καλούν global constants, models στέλνουν webhooks και helpers ανοίγουν transactions, οι dependencies είναι implicit και οι failure boundaries μπερδεμένες.

Μια χρήσιμη δομή έχει τέσσερα layers χωρίς να χρειάζεται framework. Το transport layer κάνει parse και presentation. Το application layer εκτελεί use cases. Το domain layer κρατά invariants και state transitions. Οι adapters υλοποιούν database, queue, clock, ids και external providers.

```ruby
CreateReservation = Data.define(
  :reservations,
  :clock,
  :ids,
  :unit_of_work
) do
  def call(input)
    unit_of_work.call do
      reservation = Reservation.hold(
        id: ids.next,
        event_id: input.event_id,
        seat: input.seat,
        now: clock.now
      )
      reservations.add(reservation)
      reservation
    end
  end
end
```

Το use case δεν διαβάζει `settings`, `params` ή `Time.now`. Οι dependencies είναι constructor fields. Το transaction contract είναι explicit. Σε test μπορείς να δώσεις deterministic clock και in-memory repository που τηρεί το ίδιο interface.

Composition root μία φορά

Στο boot δημιουργείς adapters και use cases. Η Sinatra class παίρνει ένα immutable container ως setting. Routes διαλέγουν μόνο το use case που χρειάζονται.

```ruby
Container = Data.define(:create_reservation, :find_event)

def build_container(config)
  database = Database.connect(config.database_url)
  repositories = Repositories.new(database)
  Container.new(
    create_reservation: CreateReservation.new(
      reservations: repositories.reservations,
      clock: Clock.new,
      ids: SecureIds.new,
      unit_of_work: repositories.unit_of_work
    ),
    find_event: FindEvent.new(events: repositories.events)
  )
end
```

Δεν χρειάζεται generic registry με string keys. Typed container κάνει missing dependency boot failure. Αν ο container κρατά connection pool ή HTTP client, το object πρέπει να είναι thread-safe και να ξέρει fork lifecycle.

Για tests, αντί να μεταλλάξεις `App.settings.container` global μεταξύ cases, φτιάξε subclass ή app factory με δικό του container. Παράλληλα tests τότε δεν πατούν το ένα state του άλλου.

```ruby
def build_app(container)
  Class.new(Seatly::App).tap do |app|
    app.set :container, container
    app.set :environment, :test
  end
end
```

Results αντί για HTTP μέσα στο core

Use case δεν επιστρέφει Sinatra tuple. Επιστρέφει result με domain meaning. Το route mapping αποφασίζει status και representation.

```ruby
Result = Data.define(:kind, :value)

post "/reservations" do
  input = ReservationInput.from_json(json_document)
  result = settings.container.create_reservation.call(input)

  case result.kind
  when :created
    status 201
    present(result.value)
  when :seat_taken
    halt 409, json_error("seat_taken")
  else
    raise "unknown result kind: #{result.kind.inspect}"
  end
end
```

Unknown result είναι programmer mismatch και σηκώνει exception. Δεν το μετατρέπουμε σε 422. Έτσι contract evolution αποτυγχάνει εμφανώς σε test.

Όχι ένα route file ανά noun από συνήθεια

Split apps όταν χρειάζεσαι διαφορετικό middleware, authentication boundary, deployment lifecycle ή ownership. Το να κάνεις `UsersApp`, `EventsApp` και `ReservationsApp` μόνο επειδή υπάρχουν τρία nouns μπορεί να δημιουργήσει cross-app calls και duplicated filters. Ένα cohesive HTTP app με plain Ruby modules συχνά είναι απλούστερο.

Mount διαφορετικές apps όταν το boundary είναι πραγματικό:

```ruby
map "/api" do
  run PublicApi
end

map "/ops" do
  use OperatorNetworkPolicy
  run OperationsApi
end
```

Το `/ops` έχει διαφορετικό network και identity policy. Αυτό δικαιολογεί ξεχωριστό Rack subtree.

Dependency direction και adapters

Domain code δεν πρέπει να require Sinatra. Repository interface δεν χρειάζεται abstract base class. Αρκεί stable method contract και contract tests που τρέχουν σε κάθε adapter. Μην κάνεις mock every call sequence. Δοκίμασε observable behavior και failure categories.

External provider adapter μεταφράζει timeout, rejected request και ambiguous outcome σε typed results. Δεν αφήνει vendor exception classes να διασχίσουν όλο το application. Παράλληλα κρατά original cause για internal diagnostics.

Η απλότητα δεν είναι λιγότερα αρχεία. Είναι λιγότερες κρυφές διαδρομές. Ένα use case με πέντε explicit dependencies μπορεί να είναι πιο απλό από route δέκα γραμμών που καλεί πέντε globals.

SOURCE TRACE

Το Sinatra composition surface είναι `Sinatra::Base`, `.new`, `.build`, `helpers`, `register`, `use` και settings στο `lib/sinatra/base.rb` 4.2.1. Τα layers και ports είναι application design, όχι framework internals.

ΠΑΓΙΔΑ

Μη χρησιμοποιείς `settings` μέσα στο domain core. Μετατρέπει shared class configuration σε hidden service locator και δένει κάθε test στο Rack app.

ΜΗΧΑΝΙΣΜΟΣ

Το Sinatra app είναι transport shell και composition boundary. Plain Ruby use cases κρατούν decisions. Adapters κρατούν effects και version-specific APIs.

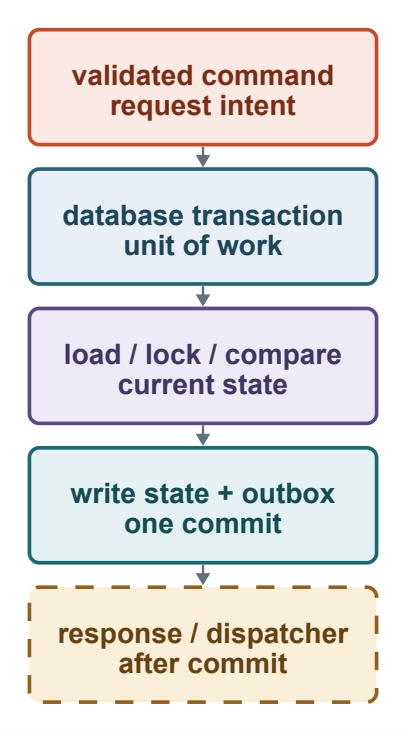
PROBE

Διάλεξε ένα route και απαρίθμησε κάθε global constant, clock, random id, database call και provider effect που αγγίζει. Κάν' τα explicit dependencies. Τρέξε το use case χωρίς Rack env. Αν χρειάζεται ακόμη Sinatra helper, το transport boundary δεν έχει κλείσει.

ΚΕΦΑΛΑΙΟ 20

Persistence, transactions και unit of work

Το Sinatra δεν ορίζει persistence lifecycle. Η εφαρμογή πρέπει να ορίσει transaction, connection ownership και τη στιγμή που ένα result γίνεται durable.



Σχήμα 20.1 · Το validated intent γράφει state και outbox μέσα σε ένα durable boundary.

Μπορείς να χρησιμοποιήσεις Sequel, Active Record, ROM, raw SQL ή remote data service. Το Sinatra δεν ανοίγει transaction και δεν επιστρέφει connection. Αυτό είναι πλεονέκτημα μόνο αν το application layer έχει explicit contract.

Ένα unit of work πρέπει να περιλαμβάνει όλα τα reads και writes που σχηματίζουν μία atomic απόφαση. Δεν πρέπει να περιλαμβάνει template rendering, slow HTTP call ή streaming response. Η transaction μένει μικρή και το αποτέλεσμα materialize πριν κλείσει.

```ruby
class ConfirmReservation
  def initialize(unit_of_work:, reservations:, outbox:)
    @unit_of_work = unit_of_work
    @reservations = reservations
    @outbox = outbox
  end

  def call(id:, expected_version:)
    @unit_of_work.call do
      reservation = @reservations.lock(id)
      reservation.confirm!(expected_version: expected_version)
      @reservations.save(reservation)
      @outbox.append(
        type: "reservation.confirmed",
        aggregate_id: reservation.id,
        version: reservation.version
      )
      reservation.snapshot
    end
  end
end
```

Το `snapshot` πρέπει να είναι detached value. Αν επιστρέψεις lazy dataset ή model που θα φορτώσει association αργότερα, μπορεί να κάνει query έξω από την transaction, σε άλλο connection και με άλλη state version.

Connection ownership ανά execution

Threaded Puma process εξυπηρετεί πολλά requests. Ο database adapter χρειάζεται bounded pool. Checkout πρέπει να συμβαίνει όσο πιο αργά γίνεται και release σε ensure. Pool size δεν ακολουθεί τυφλά τα Puma threads: background work, streaming code ή async clients μπορεί να προσθέτουν demand.

Αν ένα request κρατά connection ενώ περιμένει provider, άλλα threads μπορεί να κολλήσουν στο pool ακόμη κι αν database είναι idle. Μέτρα checkout wait ξεχωριστά από SQL time.

```ruby
class UnitOfWork
  def initialize(pool)
    @pool = pool
  end

  def call
    @pool.with_connection do |connection|
      connection.transaction { yield }
    end
  end
end
```

Το actual adapter API διαφέρει. Το contract είναι ότι exception ή explicit abort κάνει rollback και ότι connection επιστρέφει σε κάθε path. `halt` είναι throw, όχι exception, άρα μην το χρησιμοποιείς μέσα στο transaction block ως failure signal.

Constraints ανήκουν στη database

Application validation δίνει καλό error. Unique constraint, foreign key, not-null και check constraint προστατεύουν concurrent writers. Ένα `exists?` πριν από insert έχει race. Κάνε insert και μετάφρασε συγκεκριμένη constraint violation σε domain result.

Constraint names είναι compatibility surface μεταξύ schema και adapter. Δώσε stable names ώστε error mapping να μην εξαρτάται από database message text. Μην κάνεις rescue κάθε database exception ως conflict. Connection loss, serialization failure και programmer SQL error έχουν διαφορετική recovery.

Isolation και optimistic concurrency

Version column επιτρέπει conditional update:

```sql
UPDATE reservations
SET status = 'confirmed', version = version + 1
WHERE id = :id AND version = :expected_version;
```

Affected row count 1 σημαίνει success. Zero σημαίνει missing ή stale version και χρειάζεται δεύτερο bounded read αν το API ξεχωρίζει 404 από 412. Αυτό το compare-and-write είναι atomic στη database.

Pessimistic lock είναι σωστό όταν η απόφαση χρειάζεται current multi-row state και contention είναι bounded. Πάρε locks σε deterministic order. Μην κρατάς lock κατά το HTTP provider call. Deadlock ή serialization failure μπορεί να είναι retryable μόνο αν ξανατρέξεις ολόκληρο unit of work από fresh inputs και κανένα external effect δεν έχει ήδη φύγει.

Transaction callback δεν είναι message queue

Adapter after-commit hook κλείνει το window πριν από commit, αλλά όχι το window μετά το commit και πριν από enqueue. Αν ο process πέσει εκεί, η database state μένει χωρίς job. Transactional outbox γράφει intent στην ίδια transaction και ξεχωριστός dispatcher το παραδίδει.

Το HTTP response μπορεί να χαθεί αφού έγινε commit. Client retry πρέπει να είναι idempotent. Η transaction λύνει atomicity μέσα σε μία database, όχι exactly-once delivery πάνω από network.

Read consistency και replicas

Αν route γράψει primary και μετά redirect σε GET που διαβάζει replica, ο user μπορεί να δει παλιό state. Sticky primary window, causal token ή explicit read-your-write route είναι application policy. Sinatra δεν ξέρει ποιο connection role χρησιμοποιεί το repository.

Pagination πάνω σε changing data χρειάζεται stable order και cursor. Offset μπορεί να διπλασιάσει ή να παραλείψει rows όταν γίνονται inserts. Cursor με ordered unique tuple, όπως `(created_at, id)`, δίνει σαφέστερο contract.

> **SOURCE TRACE**
> Το Sinatra 4.2.1 δεν προσθέτει ORM ή transaction wrapper. Το route lifecycle φαίνεται στα `dispatch!`, `route!` και `invoke` του `lib/sinatra/base.rb`. Connection pools, isolation και callback semantics ανήκουν στον επιλεγμένο adapter και στη database documentation.

ΠΑΓΙΔΑ

Μην κρατάς database transaction ή connection μέχρι το Rack body close. Slow client μετατρέπεται σε pool exhaustion και lock retention.

ΜΗΧΑΝΙΣΜΟΣ

Unit of work κατέχει atomic database decision. External effects χρειάζονται durable intent και retry protocol. HTTP success μπορεί να χαθεί μετά το commit.

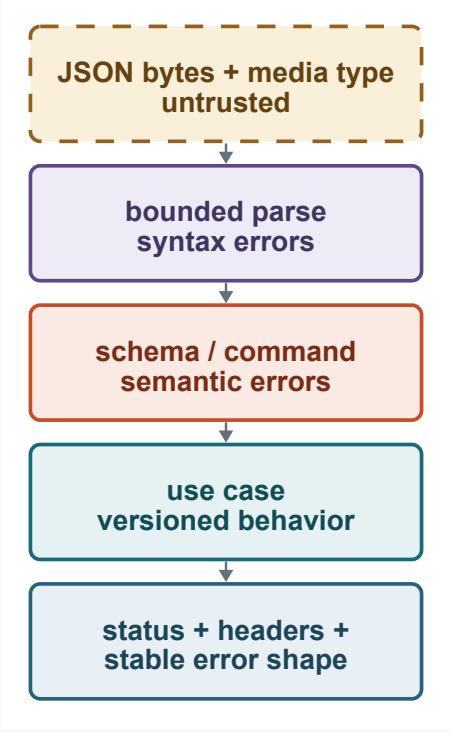
PROBE

Βάλε barriers σε δύο transactions που διεκδικούν το ίδιο seat. Assert ότι η database constraint ή version επιτρέπει ένα success. Έπειτα κάνε process-like failure μετά το commit και πριν από response. Το retry με ίδιο intent πρέπει να επιστρέψει το ίδιο outcome χωρίς δεύτερο effect.

ΚΕΦΑΛΑΙΟ 21

JSON APIs, validation και contract evolution

Ένα JSON endpoint είναι deployed protocol. Parser, schema, status, headers και error shape πρέπει να εξελίσσονται μαζί.



Σχήμα 21.1 · Τα bytes περνούν syntax και semantic validation πριν γίνουν response contract.

Το `JSON.parse` απαντά μόνο αν τα bytes είναι syntactically valid JSON. Δεν ξέρει required fields, types, ranges, cross-field invariants ή authorization. Το transport layer πρέπει να ξεχωρίζει parse failure από schema failure και domain conflict.

Ένα πρακτικό status model είναι: 400 για malformed request syntax, 401/403 για identity policy, 404 για missing resource, 415 για unsupported media type, 422 για well-formed document που δεν γίνεται valid command, 409 για conflict με current state και 412 για failed HTTP precondition. Δεν είναι θρησκεία, αλλά πρέπει να είναι consistent και documented.

```ruby
ErrorDocument = Data.define(:status, :code, :message, :fields)

helpers do
  def render_error(error)
    content_type :json
    status error.status
    JSON.generate(
      error: {
        code: error.code,
        message: error.message,
        fields: error.fields
      },
      request_id: env["seatly.request_id"]
    )
  end
end
```

Το `code` είναι stable machine identifier. Το message μπορεί να αλλάξει ή να μεταφραστεί. Τα field paths είναι bounded και δεν περιέχουν raw values. Το request id βοηθά support χωρίς να εκθέτει trace ή secret.

Media type πριν από parser

Αν endpoint δέχεται JSON, απαίτησε `application/json` ή versioned vendor type. Το charset για JSON είναι συνήθως UTF-8. Μην προσπαθείς να μαντέψεις form ή JSON από πρώτο byte. Content negotiation του response είναι διαφορετικό από request content type.

Limit body bytes πριν το parse. Limit nesting, collection sizes και string lengths στο schema. Ένα 500 KB JSON με εκατοντάδες χιλιάδες μικρά nodes μπορεί να έχει πολύ μεγαλύτερο allocation footprint.

Allow known fields και preserve compatibility

Για command endpoints, unknown fields μπορούν να είναι error ώστε typo να μην αγνοείται. Για event consumers, tolerant reader συχνά αγνοεί άγνωστα additive fields. Η policy εξαρτάται από direction και ownership. Μην εφαρμόζεις strict parser παντού επειδή ακούγεται ασφαλής.

```ruby
class ReservationInput
  ALLOWED = %w[event_id seat note].freeze

  def self.from_document(document)
    raise InputError, "object_required" unless document.is_a?(Hash)
    unknown = document.keys - ALLOWED
    raise InputError, "unknown_fields" unless unknown.empty?

    new(
      event_id: Integer(document.fetch("event_id"), 10),
      seat: String(document.fetch("seat")),
      note: document["note"]
    )
  end

  def initialize(event_id:, seat:, note:)
    @event_id = event_id
    @seat = seat
    @note = note
    freeze
  end
end
```

`String(value)` δέχεται objects με `to_str` και αριθμούς μέσω conversion. Αν θέλεις μόνο JSON String, έλεγξε `is_a?(String)`. Coercion policy πρέπει να είναι intentional.

Μην serializeάρεις domain object τυφλά

`JSON.generate(model)` ή generic `to_h` μπορεί να προσθέσει νέα internal fields στο public response μετά από refactor. Φτιάξε presenter που δηλώνει exact shape, types και nullability. Timestamp format, decimal representation και id type είναι contract.

Lists χρειάζονται stable ordering και pagination. Opaque cursor μπορεί να περιέχει signed versioned payload. Μην βάζεις raw SQL offset ή internal primary key αν δεν θέλεις να γίνει public guarantee. Links πρέπει να παράγονται με σωστό mount point και trusted origin.

Evolution στο overlap window

Rolling deploy σημαίνει old και new producers/consumers μαζί. Add optional field πρώτα. Κάνε readers tolerant. Μετά άρχισε write. Μόνο αφού λήξει το maximum client και queued-message horizon μπορείς να αφαιρέσεις old shape.

Rename δεν είναι atomic. Για transition, διάβαζε old και new, γράφε canonical new και, αν χρειάζεται, dual-write old field για περιορισμένο διάστημα. Μέτρα χρήση old field πριν το αφαιρέσεις. API version στο path δεν λύνει μόνο του database ή job payload overlap.

Error contracts σε partial failure

Μην επιστρέφεις provider message αυτούσιο. Map timeout σε stable code και βάλε retry hint μόνο όταν operation είναι safe. Αν external outcome είναι unknown, το response πρέπει να το λέει, ίσως 202 με status resource, όχι να προσποιείται clean 500 που ενθαρρύνει blind retry.

Batch endpoint χρειάζεται per-item results ή atomic all-or-nothing statement. Μη χρησιμοποιείς 200 με mixed implicit failures χωρίς schema. Όρισε ordering, duplicate handling και maximum batch size.

SOURCE TRACE
Το request και response shell χρησιμοποιεί `Sinatra::Request`, `Helpers#content_type`, `#status`, `#headers`, `#halt` και `Response#finish` στο Sinatra 4.2.1. JSON schema και version evolution είναι application protocol, όχι built-in Sinatra feature.

ΠΑΓΙΔΑ
Generic model serialization κάνει internal refactor public API change. Δήλωσε exact presenter shape και δοκίμασε το serialized document ως contract.

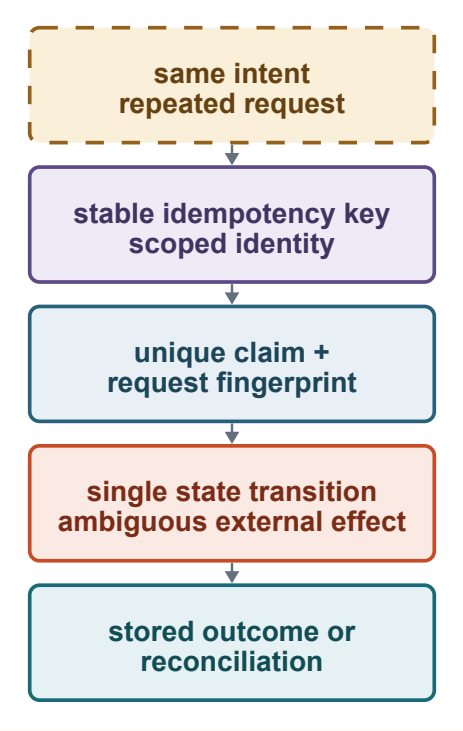
ΜΗΧΑΝΙΣΜΟΣ
Syntax parse, shape validation, domain decision και HTTP presentation είναι τέσσερις ξεχωριστές μεταφράσεις. Τα stable error codes είναι μέρος του API.

PROBE
Δημιούργησε contract fixtures από oldest supported client και newest server. Δοκίμασε unknown fields, missing fields, wrong scalar/container types, max sizes, duplicate request και mixed-version response reader. Κάνε JSON parse στο παραγόμενο response και assert exact keys, όχι substring.

ΚΕΦΑΛΑΙΟ 22

Idempotency, optimistic concurrency και race windows

Idempotency απαντά τι γίνεται όταν επαναληφθεί το ίδιο intent. Optimistic concurrency απαντά τι γίνεται όταν άλλαξε το state που είδε ο caller.



Σχήμα 22.1 · Το stable key διεκδικεί μία intent row και οδηγεί σε replay ή repair.

Network timeout δεν λέει αν ο server εκτέλεσε την operation. Client, proxy ή load balancer θα επαναλάβει request. Αν το endpoint δημιουργεί charge, reservation ή webhook, το retry πρέπει να αναγνωρίζει το ίδιο intent.

Idempotency key χρειάζεται scope. Το ίδιο string από δύο users δεν είναι ίδιο intent. Χρειάζεται επίσης request fingerprint. Αν caller επαναχρησιμοποιήσει key με διαφορετικό amount ή seat, ο server πρέπει να επιστρέψει conflict, όχι το παλιό result πάνω σε νέα parameters.

```ruby
IdempotentRequest = Data.define(:scope, :key, :fingerprint)

post "/reservations" do
  document = json_document
  request_identity = IdempotentRequest.new(
    scope: env.fetch("seatly.principal").id,
    key: request.get_header("HTTP_IDEMPOTENCY_KEY").to_s,
    fingerprint: RequestFingerprint.call(document)
  )
  result = Reservations.create_once(request_identity, document)
  present(result)
end
```

Το fingerprint πρέπει να βασίζεται σε canonical accepted fields, όχι raw JSON whitespace ή key order. Το key έχει bounded grammar και expiry policy.

Claim row και concurrent callers

Database table μπορεί να έχει unique constraint σε `(scope, key)`, fingerprint, state, result reference και timestamps. Πρώτος caller δημιουργεί claim. Ο δεύτερος με ίδιο fingerprint βλέπει processing ή stored outcome. Με άλλο fingerprint παίρνει 409.

Το read-then-insert χωρίς unique constraint έχει race. Το unique index είναι ο arbiter. Μετά από conflict, ξαναδιάβασε existing row. Αν state είναι processing, διάλεξε bounded wait, 202 status resource ή 409 retry-later policy. Μην κρατάς Puma thread επ' αόριστον.

Success response πρέπει να αποθηκεύει domain outcome, όχι απαραίτητα raw HTTP bytes. Αν presentation schema αλλάξει, μπορείς να ξαναπαρουσιάσεις stable result. Για operation όπου exact response replay είναι contract, version το stored response.

Crash windows

Αν business write και idempotency row είναι στην ίδια database, γράψε τα στην ίδια transaction. Αν external effect μεσολαβεί, υπάρχουν unknown outcomes:

1. provider δέχτηκε effect, response χάθηκε
2. local process έπεσε πριν γράψει receipt
3. retry βλέπει processing claim χωρίς owner

Χρειάζεσαι provider idempotency key ή lookup endpoint. Χωρίς αυτά, automatic retry μπορεί να διπλασιάσει effect. Mark state `unknown`, βάλε reconciliation και human review όταν το business risk το απαιτεί.

Optimistic concurrency είναι διαφορετικό key

Idempotency key αναγνωρίζει command. Version ή ETag αναγνωρίζει resource state που είδε ο caller. Update μπορεί να είναι idempotent αλλά stale: ίδιο request key δεν πρέπει να παρακάμψει failed `If-Match` για διαφορετική state version.

```ruby
patch "/reservations/:id" do |id|
  expected = request.get_header("HTTP_IF_MATCH").to_s
  command = ReservationPatch.from_document(json_document)
  result = Reservations.patch_if_version(id, command, expected)
  halt 412 unless result.updated?
  etag result.etag
  present(result.reservation)
end
```

Το repository κάνει atomic conditional update. Ruby mutex γύρω από route δεν προστατεύει άλλους processes, jobs ή direct database writers.

Retry boundary και budget

Retry μόνο transient failure και μόνο ολόκληρη replayable unit. Βάλε maximum attempts, exponential backoff με jitter και deadline. Connection timeout πριν send μπορεί να είναι safe. Read timeout μετά send μπορεί να είναι ambiguous. Μην ταξινομείς και τα δύο ως ίδιο generic timeout.

Idempotency records έχουν retention horizon τουλάχιστον όσο το maximum client retry window. Αν τα διαγράψεις νωρίτερα, delayed retry γίνεται νέο intent. Unbounded retention γίνεται storage cost. Product protocol πρέπει να δηλώνει πόσο καιρό key παραμένει valid.

In-process dedupe δεν αρκεί

Hash ή mutex στο Sinatra process μπορεί να μειώσει duplicate work μέσα σε έναν worker, αλλά δεν είναι correctness boundary. Restart χάνει state και cluster έχει πολλά copies. Χρησιμοποίησέ το μόνο ως optimization πάνω από durable unique claim.

Metrics: new claims, replayed successes, fingerprint conflicts, processing waits, stale claims, unknown outcomes και reconciliation age. Μην βάζεις raw idempotency key ως metric label.

SOURCE TRACE
Το Sinatra παρέχει request headers, route execution και response helpers, όχι idempotency store. Το request μπορεί να εκτελεστεί παράλληλα επειδή `Base.call` δεν κλειδώνει όταν `lock` είναι false. Η durable claim και conditional update ανήκουν στην database ή provider contract.

ΠΑΓΙΔΑ
`Mutex` και in-memory cache προστατεύουν μόνο ένα process. Δεν λύνουν duplicate requests σε Puma cluster, restart ή queue redelivery.

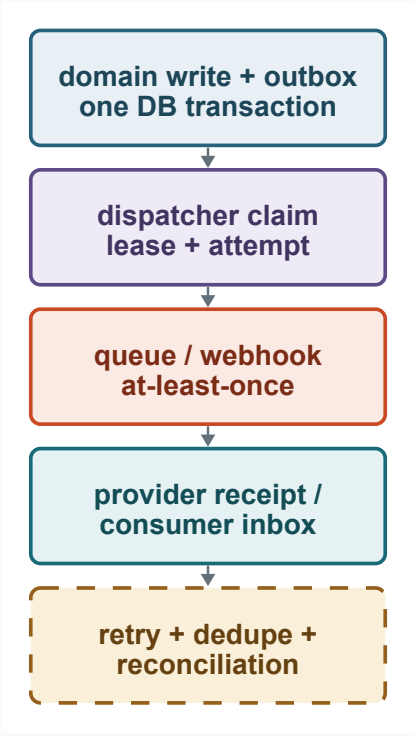
ΜΗΧΑΝΙΣΜΟΣ
Idempotency identity είναι `(scope, key, fingerprint)`. Resource concurrency χρειάζεται ξεχωριστή expected version. External ambiguity χρειάζεται lookup ή reconciliation.

PROBE
Στείλε 20 concurrent POSTs με ίδιο key και body, μετά ίδιο key με άλλο body. Βάλε crash points πριν claim, μετά claim, μετά business commit και μετά provider success. Assert ένα domain transition, stable replay, visible unknown state και bounded recovery.

ΚΕΦΑΛΑΙΟ 23

Jobs, outbox και webhooks

To 202 Accepted είναι αληθινό μόνο όταν υπάρχει durable intent που άλλος process μπορεί να συνεχίσει μετά τον θάνατο του request worker.



Σχήμα 23.1 · Η outbox κλείνει το local commit window και η reconciliation τα υπόλοιπα.

Το Sinatra δεν έχει built-in job system και αυτό είναι καλό για το mental model. Η route πρέπει να ξέρει τι ακριβώς εγγυάται το queue adapter. In-memory thread ή `Queue` χάνεται σε restart. Redis list, database queue ή broker έχουν διαφορετικό acknowledgment, redelivery και ordering contract.

Αν route γράψει database και μετά κάνει enqueue, process loss ανάμεσα στα δύο αφήνει committed state χωρίς job. Αν κάνει enqueue πρώτα, worker μπορεί να τρέξει πριν γίνει commit ή το database write να αποτύχει. Transactional outbox γράφει domain state και message intent στην ίδια database transaction.

```ruby
unit_of_work.call do
  reservation = reservations.confirm(id)
  outbox.append(
    event_id: SecureRandom.uuid,
    event_type: "reservation.confirmed",
    aggregate_id: reservation.id,
    aggregate_version: reservation.version,
    payload: ReservationConfirmedPayload.build(reservation)
  )
end
```

Μετά το commit, dispatcher claimάρει unpublished rows, στέλνει και σημειώνει outcome. Αν πέσει μετά το send και πριν το mark, θα ξαναστείλει. Το boundary είναι at-least-once. Consumer χρειάζεται inbox ή domain idempotency.

Claim, lease και poison messages

Πολλοί dispatchers χρειάζονται atomic claim. Row lock with skip-locked, conditional update ή queue-specific lease μπορεί να το δώσει. Claim έχει owner και expiry ώστε orphan work να ξαναγίνει eligible. Μικρό expiry προκαλεί duplicate concurrent sends. Μεγάλο expiry καθυστερεί recovery.

Κάθε attempt καταγράφει bounded error category και next attempt time. Permanent contract error δεν πρέπει να retryάρει για πάντα. Poison row πάει σε visible terminal state με operator action. Μην αποθηκεύεις unbounded provider response ή secrets στην exception text.

Payload ως temporal API

Job ή event μπορεί να εκτελεστεί μετά από deploy. Class names και arbitrary Ruby serialization είναι brittle. Χρησιμοποίησε primitive versioned document με event type, schema version, stable aggregate identity και occurred time.

Reference payload φορτώνει current state και μπορεί να χάσει historical meaning. Snapshot payload κρατά event-time facts αλλά μεγαλώνει compatibility surface. Διάλεξε ανά field. Για `reservation.confirmed`, ticket number και price snapshot ίσως είναι event facts, ενώ display name μπορεί να φορτωθεί current αν το product το επιτρέπει.

Webhook delivery protocol

Webhook είναι outbound message πάνω από unreliable HTTP. Κάθε delivery έχει stable event id και attempt id. Signature καλύπτει timestamp και exact raw body. Consumer ελέγχει signature constant-time, timestamp window και event-id dedupe πριν εφαρμόσει effect.

```ruby
def sign_webhook(secret:, timestamp:, body:)
  data = "#{timestamp}.#{body}"
  OpenSSL::HMAC.hexdigest("SHA256", secret, data)
end
```

Canonical JSON reserialization στον consumer μπορεί να αλλάξει bytes και να σπάσει signature. Verify raw bytes πρώτα, parse μετά. Secret rotation χρειάζεται key id ή old/new verification window. Μην βάζεις secret σε query string.

HTTP 2xx μπορεί να θεωρηθεί acknowledgment, αλλά consumer μπορεί να απαντήσει πριν το effect γίνει durable. Αυτό είναι δικό του contract. Για high-value integration κράτα reconciliation endpoint ή periodic comparison με canonical state.

Inbound inbox

Consumer γράφει unique `(provider, event_id)` receipt και domain effect στην ίδια local transaction. Αν duplicate event έρθει, επιστρέφει το stored outcome χωρίς δεύτερο effect. Αν provider δεν δίνει stable id, φτιάξε fingerprint από document μόνο όταν η business semantics το επιτρέπει.

Ordering πρέπει να είναι scoped. Global order πάνω σε distributed deliveries σπάνια υπάρχει. Aggregate version επιτρέπει στον consumer να απορρίψει stale event, να bufferάρει gap ή να κάνει re-fetch. Η σωστή επιλογή εξαρτάται αν κάθε intermediate transition έχει σημασία.

Request και worker capacity χωριστά

Μην τρέχεις durable dispatcher ως ad hoc thread μέσα σε κάθε Sinatra web process χωρίς leader election και shutdown protocol. Web scale-out τότε αλλάζει worker count. Προτίμησε ξεχωριστό process role ή queue adapter με σαφή supervision.

Health endpoint δεν πρέπει να λέει healthy μόνο επειδή web routes απαντούν. Παρακολούθησε oldest outbox age, ready jobs, retry age, poison count και worker heartbeat. Backlog count χωρίς age μπορεί να κρύψει stuck critical message μέσα σε πολλά φρέσκα low-priority messages.

SOURCE TRACE
Το Sinatra request lifecycle ολοκληρώνεται στο `Base#call!` και δεν παρέχει durable enqueue hook. Τα job, outbox και webhook guarantees είναι adapter και application protocols. Το Rack body close ή after filter δεν αποτελεί durable commit boundary.

ΠΑΓΙΔΑ
Μην απαντάς 202 αφού έβαλες work μόνο σε process memory. Accepted σημαίνει ότι το intent επιβιώνει worker loss και υπάρχει owner που θα το ξαναβρεί.

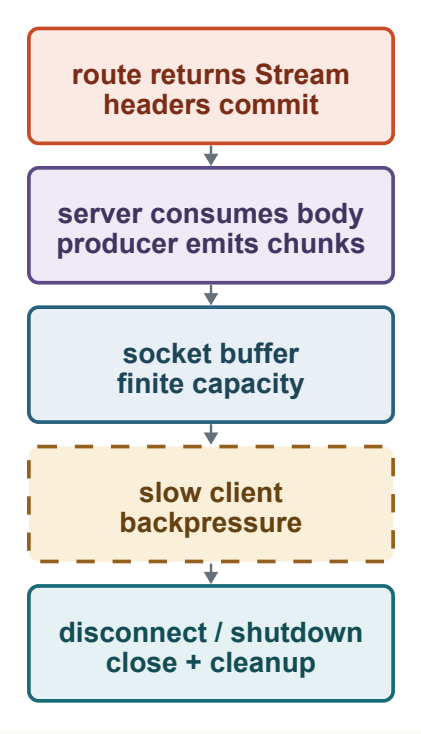
ΜΗΧΑΝΙΣΜΟΣ
Outbox προστατεύει local database intent. Inbox προστατεύει local consumption. At-least-once transport, idempotency και reconciliation κλείνουν τα υπόλοιπα crash windows.

PROBE
Βάλε deterministic crash μετά το domain commit, μετά το claim, μετά το webhook send και πριν το mark. Στον consumer, crash μετά το receipt και μετά το effect. Assert eventual terminal state, duplicate tolerance, gap visibility και bounded retry. Μέτρα oldest unresolved age, όχι μόνο row count.

ΚΕΦΑΛΑΙΟ 24

Streaming, SSE και backpressure

Streaming δεν σημαίνει ότι το request έγινε φθηνό ή ασύγχρονο. Σημαίνει ότι το response body παράγει bytes σε χρόνο που δεν συμπίπτει με το route block.



Σχήμα 24.1 · Το streaming path χρειάζεται bounded buffer, disconnect handling και δικό του capacity budget.

Στο Rack το response body δεν είναι απλώς ένα string. Είναι object που απαντά σε `each`, κάνει yield strings και, όταν έχει `close`, πρέπει τελικά να κλείσει. Τα headers και το status αποφασίζονται πριν καταναλωθεί όλο το body. Αν η παραγωγή αποτύχει αφού φύγει το πρώτο chunk, δεν μπορείς πια να αλλάξεις καθαρά το response σε JSON 500. Η αποτυχία ανήκει στο stream protocol.

Το `stream` helper του Sinatra επιστρέφει `Sinatra::Helpers::Stream`. Το `Stream#each` ξεκινά τον producer μέσω scheduler, το `<<` παραδίδει chunks και το `close` τρέχει callbacks. Αν το Rack environment έχει `async.callback`, το Sinatra επιλέγει EventMachine scheduler. Διαφορετικά χρησιμοποιεί τον δικό του άμεσο scheduler. Άρα η πραγματική παραλληλία, το buffering και το αν δεσμεύεται server thread εξαρτώνται από handler και deployment.

Για finite response το contract είναι καθαρό:

```ruby
get "/exports/:id" do |id|
  content_type "application/x-ndjson"
  attachment "export-#{id}.ndjson"

  stream do |out|
    Exports.each_row(id) do |row|
      break if out.closed?
      out << JSON.generate(row) << "\n"
    end
  end
end
```

Το route δεν πρέπει να κρατά ανοιχτή database transaction όσο γράφει αργά στο δίκτυο. Διάβασε με bounded pages ή cursor του οποίου τη συνέπεια έχεις ορίσει. Αν απαιτείται point-in-time snapshot, κάνε το κόστος και το timeout ρητό. Διαφορετικά ένα export μπορεί να δει διαφορετικές versions καθώς προχωρά.

SSE είναι application protocol

Server-Sent Events χρειάζεται `text/event-stream`, records χωρισμένα με κενή γραμμή και συνήθως `id`, `event` και `data`. Data με πολλές γραμμές χρειάζεται ένα `data:` prefix ανά γραμμή. Μην κάνεις interpolation raw user text μέσα στο wire format. Φτιάξε encoder και χρησιμοποίησε JSON ως payload.

Κάθε event έχει monotonically ordered identity μέσα σε σαφές scope. Ο browser μπορεί να ξανασυνδεθεί με `Last-Event-ID`. Αν τα events είναι μόνο process memory, reconnect σε άλλο worker δημιουργεί κενό. Durable log ή snapshot-plus- cursor endpoint επιτρέπει replay. Όταν το requested id έχει λήξει, απάντησε με ορισμένη resync συμπεριφορά, όχι με σιωπηλή απώλεια.

Heartbeat comment όπως `: ping\n\n` κρατά visible μια ήσυχη σύνδεση και αποκαλύπτει disconnect, αλλά δεν θεραπεύει proxy timeout αν ο proxy κάνει buffering. Έλεγξε reverse proxy, CDN και load balancer για buffering, idle timeout και maximum response duration. Το `X-Accel-Buffering: no` είναι χρήσιμο μόνο σε υποδομή που το καταλαβαίνει.

Ο αργός consumer είναι capacity problem

Αν producer παράγει γρηγορότερα από όσο γράφει ο client, unbounded queue γίνεται memory leak. Bounded queue χρειάζεται policy: block producer, drop παλιό event, drop νέο event ή κλείσε τη σύνδεση και απαίτησε replay. Για durable business events, το κλείσιμο και reconnect από stable cursor είναι συνήθως καθαρότερο από silent drop.

Το backpressure πρέπει να περνά μέχρι τον πραγματικό producer. Το ότι το `out << chunk` επιστρέφει δεν αποδεικνύει ότι ο client παρέλαβε το chunk ή ότι δεν αντιγράφηκε σε ενδιάμεσο buffer. Μέτρα queue depth, oldest queued age, bytes pending και disconnect reason. Βάλε per-connection byte limit, όχι μόνο event count, επειδή τα payloads έχουν διαφορετικό μέγεθος.

Keep-open χωρίς κρυφή υπόθεση

Το `stream(:keep_open)` αποτρέπει το αυτόματο close, όμως η fallback διαδρομή του Sinatra, όταν δεν υπάρχει async scheduler, επανακαλεί το producer block μέχρι να κλείσει το stream. Block που απλώς

κάνει subscribe και επιστρέφει μπορεί να κάνει repeated subscriptions. Δοκίμασε το ακριβές server adapter και μην θεωρήσεις ότι το chat example ορίζει production semantics για τον Puma.

Για μακρόβιο SSE προτίμησε adapter με τεκμηριωμένο streaming contract ή δικό σου Rack body που έχει bounded mailbox, σαφή `each` και idempotent `close`. Αυτό το body μπορεί να δεσμεύει Puma thread. Δεν είναι λάθος αν το capacity model το περιλαμβάνει. Είναι λάθος όταν χιλιάδες connections εμφανίζονται σαν δωρεάν επειδή το API ονομάζεται stream.

Cleanup πρέπει να συμβαίνει και σε normal end και σε client disconnect. Subscription, timer, cursor και προσωρινό file κλείνουν idempotently. Exception στο cleanup δεν πρέπει να κρύψει την πρώτη αποτυχία. Το `after` filter τρέχει πριν ολοκληρωθεί η κατανάλωση ενός streaming body, άρα δεν είναι σωστό σημείο για stream-lifetime cleanup.

SOURCE TRACE

Στο Sinatra 4.2.1 τα `Helpers::Stream#each`, `#<<`, `#close` και `#callback` ορίζουν το stream lifecycle. Το `stream` επιλέγει scheduler από το `env['async.callback']` και έχει ειδική fallback συμπεριφορά για keep-open. Το Rack contract απαιτεί string chunks και τελικό body close όπου υπάρχει.

ΠΑΓΙΔΑ

Μην κρατάς transaction, checkout από connection pool ή request-scoped object μέχρι να φύγει ο τελευταίος SSE client. Ένα αργό socket μπορεί να το κρατήσει για ώρες.

ΜΗΧΑΝΙΣΜΟΣ

Σχεδίασε stream ως ξεχωριστό protocol: replay identity, bounded buffering, disconnect cleanup, proxy behavior, capacity limit και graceful drain.

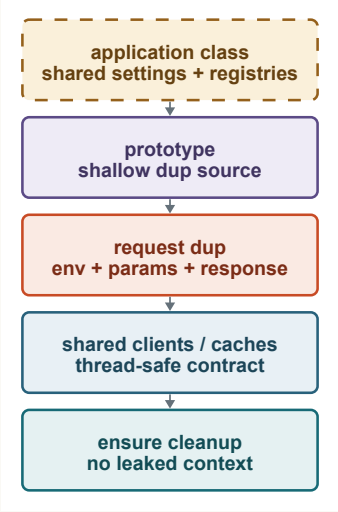
PROBE

Τρέξε real-server test με client που διαβάζει ένα byte ανά δευτερόλεπτο, client που κόβει TCP στη μέση και reconnect με παλιό `Last-Event-ID`. Κάνε deploy ενώ υπάρχουν streams. Assert bounded memory, released subscriptions, ορατό gap και τερματισμό μέσα στο shutdown deadline.

ΚΕΦΑΛΑΙΟ 25

Thread safety και shared mutable state

Το Sinatra απομονώνει το request object. Δεν απομονώνει αυτόματα οτιδήποτε έβαλες σε settings, constants, clients, caches ή closures.



Σχήμα 25.1 · Κάθε request έχει instance state, ενώ οι ρητές dependencies και caches ανήκουν στο process.

Το `Sinatra::Base#call` εκτελεί `dup.call!`. Κάθε request δουλεύει σε shallow copy του prototype και παίρνει δικά του `@request`, `@response`, `@params` και route temporaries. Το shallow είναι η κρίσιμη λέξη. Objects που δείχνουν οι instance variables του prototype, τα class settings ή Ruby constants δεν γίνονται deep copy.

Ένα client object μπορεί να είναι thread-safe, thread-confined ή να απαιτεί pool. Ένα plain Hash cache μπορεί να δεχτεί concurrent mutations. Ένα mutable array σε setting μπορεί να αλλάξει ενώ άλλο request το διαβάζει. Η ασφάλεια πρέπει να δηλώνεται ανά dependency.

```ruby
Dependencies = Data.define(:users, :clock, :events)

class AccountsApp < Sinatra::Base
  set :dependencies, Dependencies.new(
    users: UserRepositoryPool.new(size: 6),
    clock: Process.method(:clock_gettime),
    events: EventPublisher.new
  ).freeze

  get "/accounts/:id" do |id|
    account = settings.dependencies.users.with do |repo|
      repo.fetch(id)
    end
    present(account)
  end
end
```

Το frozen dependency container εμποδίζει αλλαγή references, όχι εσωτερική mutation των objects. Κάθε object χρειάζεται δικό του contract. Repository pool παραχωρεί exclusive handle και το επιστρέφει με `ensure`. Publisher είτε είναι concurrency-safe είτε έχει pool/serialized owner.

GVL δεν είναι transaction

Στο CRuby το Global VM Lock εμποδίζει δύο threads να εκτελούν Ruby bytecode ταυτόχρονα για μεγάλα διαστήματα, αλλά I/O και native code μπορούν να παραλληλίζονται. Ακόμη και μια λογικά σύνθετη operation πάνω σε Hash είναι σειρά βημάτων. Check-then-act, memoization και read-modify-write έχουν race. Άλλες Ruby implementations έχουν διαφορετικό runtime.

```ruby
class PriceCache
  def initialize
    @values = {}
    @mutex = Mutex.new
  end

  def fetch(key)
    @mutex.synchronize do
      return @values.fetch(key) if @values.key?(key)
      @values[key] = yield
    end
  end
end
```

Αυτό δίνει μία computation ανά key μόνο επειδή κρατά global mutex και κατά το slow `yield`. Είναι σωστό contract αλλά μπορεί να γίνει convoy. Production cache μπορεί να χρειάζεται per-key single-flight, timeout, bounded size, negative-cache policy και eviction εκτός critical section. Πρώτα γράψε τη semantics, μετά διάλεξε primitive.

`lock` είναι χοντρό εργαλείο

Το Sinatra setting `lock` μπορεί να βάλει mutex γύρω από κάθε `call!` μέσα σε ένα application instance. Αυτό σειριοποιεί requests στο συγκεκριμένο process. Δεν προστατεύει άλλο Puma worker, job process ή external writer. Μπορεί να είναι χρήσιμο για legacy adapter που αποδεδειγμένα δεν είναι thread-

safe, αλλά μετατρέπει τα max threads σε ουρά αναμονής. Δεν είναι distributed lock ούτε διόρθωση database race.

Προτίμησε να περιορίσεις το serialization γύρω από τον συγκεκριμένο resource. Αν ο resource δεν μπορεί να μοιραστεί, δώσε έναν ανά thread ή checkout από pool. Βάλε timeout στο checkout ώστε saturation να γίνει ελεγχόμενη 503 και όχι ανεξήγητο request timeout.

Context και reuse worker threads

Puma threads εξυπηρετούν πολλά requests διαδοχικά. Οτιδήποτε μπαίνει σε thread-local ή fiber-local storage πρέπει να καθαρίζει με `ensure`. Αλλιώς principal, request id ή locale μπορεί να διαρρεύσει στο επόμενο request. Το Puma 8 έχει προαιρετικό `fiber_per_request`, που δίνει καθαρό fiber storage, αλλά είναι defense in depth. Η εφαρμογή οφείλει ακόμη να καθαρίζει resources.

Εδώ υπάρχει lifecycle trap. Το `ensure` γύρω από `app.call(env)` τελειώνει μόλις επιστραφεί το Rack triple. Streaming body καταναλώνεται αργότερα. Αν το body χρειάζεται context, middleware πρέπει να τυλίξει και `each` και `close` ή να περάσει immutable context μέσα στο body. Μην βασίζεσαι σε ambient thread state μετά το route.

Request-local memoization σε instance variable route helper είναι συνήθως ασφαλής επειδή το request instance είναι διαφορετικό. Memoization σε class method, module singleton ή object stored στα settings είναι process-shared. Closure που δημιουργήθηκε στο boot μπορεί επίσης να κλείσει πάνω από mutable array ή client.

Fork safety

Με Puma workers, process fork δημιουργεί νέο address space. Connections, background threads και locks που δημιουργήθηκαν πριν το fork μπορεί να μην είναι έγκυρα στο child. Immutable code και read-only data μπορούν να προφορτωθούν για copy-on-write. Sockets, pools και thread-owning clients ανοίγουν στο worker boot και κλείνουν στο worker shutdown.

Η διαφορά thread-safety και fork-safety πρέπει να υπάρχει στο dependency inventory. Σημείωσε owner, creation phase, concurrency contract, checkout limit, reset method και shutdown method για κάθε external client.

SOURCE TRACE
Το Sinatra 4.2.1 υλοποιεί `Base#call` με shallow `dup` και αρχικοποιεί request state στο `call!`. Το optional `lock` καλεί το request μέσα από mutex. Τα settings και το prototype παραμένουν class/process-level objects.

ΠΑΓΙΔΑ
Μην αποδεικνύεις thread safety επειδή ένα concurrent test πέρασε μία φορά. Το test μπορεί απλώς να μην πέτυχε το race window.

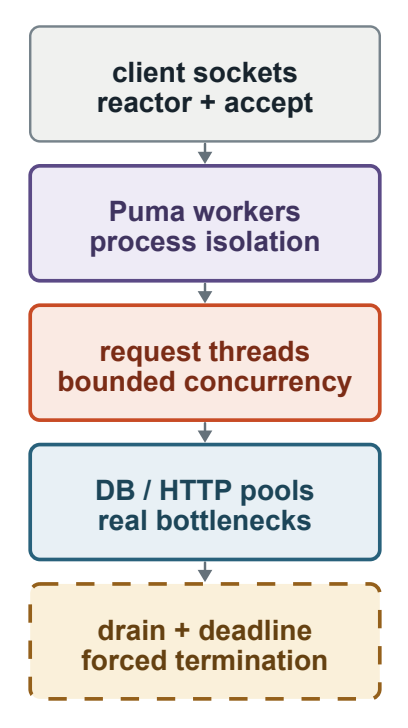
ΜΗΧΑΝΙΣΜΟΣ
Κάθε mutable object έχει owner και concurrency protocol. Request instance, thread, pool, process και durable store είναι διαφορετικά ownership scopes.

PROBE
Βάλε barriers ανάμεσα σε read και write για να αναγκάσεις δύο requests να μπουν στο ίδιο window. Τρέξε με πολλά threads και μετά με δύο workers. Assert pool timeout, context cleanup, fork reconnect και bounded cache. Επανάλαβε με streaming body που ολοκληρώνεται αφού επιστρέψει το `app.call`.

ΚΕΦΑΛΑΙΟ 26

Puma capacity, processes και graceful shutdown

Η ρύθμιση Puma δεν είναι tuning με τυχαίους αριθμούς. Είναι λογιστικό φύλλο για threads, connections, memory, latency και shutdown time.



Σχήμα 26.1 · Η πραγματική χωρητικότητα περιορίζεται από το στενότερο pool και από το shutdown budget.

Ένας Puma worker είναι process. Μέσα του υπάρχει thread pool. Τα workers δίνουν process isolation και CPU parallelism, ενώ τα threads αξιοποιούν concurrent I/O. Στο CRuby, CPU-heavy Ruby work δεν γίνεται πιο γρήγορο επειδή αύξησες threads. Μπορεί μάλιστα να μεγαλώσει queueing και tail latency.

Ξεκίνα από workload classes. Ένα route περιμένει database, άλλο κάνει μεγάλο JSON encode, άλλο κρατά SSE connection. Αν μοιράζονται το ίδιο pool, το ένα μπορεί να πνίξει τα άλλα. Separate process role ή endpoint isolation είναι μερικές φορές πιο απλό από περίπλοκο priority system.

```ruby
min_threads = Integer(ENV.fetch("PUMA_MIN_THREADS", "0"))
max_threads = Integer(ENV.fetch("PUMA_MAX_THREADS", "5"))
web_workers = Integer(ENV.fetch("WEB_CONCURRENCY", "2"))

threads min_threads, max_threads
workers web_workers
preload_app!

before_fork do
  Dependencies.close_prefork_resources
end

on_worker_boot do
  Dependencies.open_worker_resources(pool_size: max_threads)
end

on_worker_shutdown do
  Dependencies.close_worker_resources
end

force_shutdown_after 25
```

Οι αριθμοί είναι παράδειγμα εκκίνησης, όχι recommendation για κάθε host. Με 2 workers και μέχρι 5 request threads έχεις μέχρι 10 concurrent request slots. Αν κάθε request μπορεί να κρατά database connection, το web connection budget είναι τουλάχιστον 10 συν headroom για health checks και management work. Πρόσθεσε job workers, console και migrations πριν συγκρίνεις με το database maximum.

Pool μικρότερο από thread count μπορεί να είναι σκόπιμο admission control. Τότε όμως το checkout timeout πρέπει να είναι μικρότερο από request deadline και η saturation να μετριέται. Pool πολύ μεγαλύτερο από usable concurrency σπαταλά connections και επιτρέπει burst που γονατίζει τη database.

Queueing και overload

Το Puma έχει accept/reactor path και thread pool. Με `queue_requests true`, που είναι default, μπορεί να δεχτεί και να κάνει queue requests πριν τα πάρουν handler threads. Αυτό βελτιώνει throughput σε αρκετά workloads, αλλά κρύβει overload ως latency. Με `queue_requests false`, κάθε worker δέχεται όσα μπορεί να χειριστεί ταυτόχρονα, απενεργοποιούνται HTTP keep-alives και slow clients καταλαμβάνουν handler threads. Η επιλογή εξαρτάται και από το reverse proxy.

Κάθε ουρά χρειάζεται όριο ή upstream admission policy. Αν το service μπορεί να ολοκληρώνει 100 requests/s και φτάσουν 1.000/s, η μεγάλη ουρά δεν δημιουργεί capacity. Μετατρέπει γρήγορο 503 σε αργό timeout. Παρακολούθησε pool capacity, backlog, busy threads και request queue time χωριστά από service time.

Deadline πρέπει να διασχίζει τα layers. Proxy timeout, app timeout, database statement timeout και provider timeout δεν είναι τέσσερις άσχετοι αριθμοί. Ο inner operation χρειάζεται χρόνο να αποτύχει και να καθαρίσει πριν ο outer caller εγκαταλείψει. Retry μέσα σε request καταναλώνει το ίδιο συνολικό budget.

Preload και copy-on-write

Το `preload_app!` φορτώνει application code στο master πριν τα worker forks. Read-mostly pages μπορούν να μοιραστούν μέσω copy-on-write. Mutation μετά το fork αντιγράφει pages και αυξάνει RSS. Αυτό είναι optimization, όχι άδεια να κληρονομήσεις live database socket ή background thread.

Κάθε dependency χαρακτηρίζεται prefork-safe ή worker-only. Random generator, connection pool, telemetry exporter και thread scheduler θέλουν ξεχωριστή σκέψη. Hooks πρέπει να είναι idempotent επειδή restart και boot failure μπορούν να αφήσουν μερικώς ανοιγμένα resources.

Graceful σημαίνει bounded drain

Σε deploy, ο παλιός worker σταματά να παίρνει νέα work, ολοκληρώνει in-flight requests και κλείνει resources. Αν μπορεί να περιμένει για πάντα, δεν είναι graceful protocol αλλά leak. Το `force_shutdown_after` βάζει τελικό όριο. Διάλεξέ το μεγαλύτερο από το αποδεκτό request deadline συν cleanup margin και μικρότερο από το orchestrator kill deadline.

Το Puma προσφέρει `drain_on_shutdown`, το οποίο μπορεί να αδειάσει pending connections από το accept socket πριν σταματήσει. Αυτό αλλάζει πόσο work αναλαμβάνει ο παλιός process κατά shutdown. Μην το ενεργοποιείς μηχανικά. Σε rolling deploy ίσως θέλεις το load balancer να μεταφέρει γρήγορα νέο traffic στα νέα instances.

Long streams χρειάζονται ειδική πολιτική. Μπορείς να στείλεις reconnect event, να κλείσεις stream και να βασιστείς σε durable cursor. Αν αφήσεις κάθε stream να ζήσει φυσικά, deploy duration γίνεται ίσο με τη μακρύτερη σύνδεση.

Readiness αφαιρεί το instance από routing πριν αρχίσει drain. Liveness δεν πρέπει να σκοτώνει process επειδή η dependency έχει προσωρινή επιβράδυνση. Startup check πρέπει να αποδεικνύει ότι routes και κρίσιμες dependencies έχουν αρχικοποιηθεί, όχι απλώς ότι άνοιξε TCP port.

SOURCE TRACE
Στο Puma 8.0.2 τα defaults περιλαμβάνουν 5 max threads στο CRuby και `queue_requests true`. Το DSL εκθέτει `threads`, `workers`, `preload_app!`, worker hooks, `fiber_per_request`, `drain_on_shutdown` και `force_shutdown_after`. Η σωστή τιμή παραμένει property του workload.

ΠΑΓΙΔΑ
Μην υπολογίζεις database pool ανά instance και ξεχνάς τον πολλαπλασιασμό των workers. Κάθε fork έχει δικό του pool και δική του μνήμη.

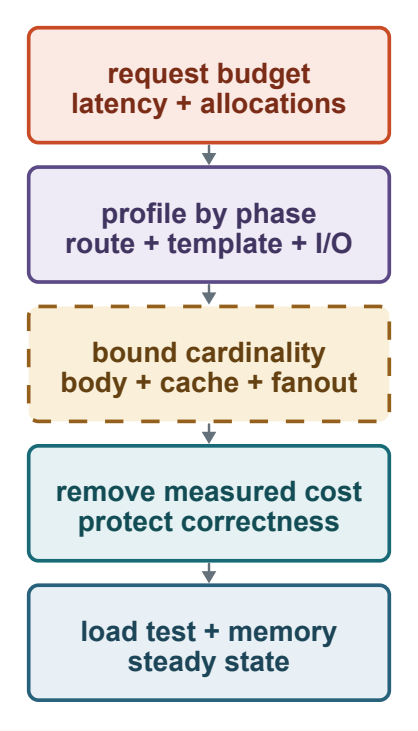
ΜΗΧΑΝΙΣΜΟΣ
Capacity είναι `workers x usable concurrency`, περιορισμένο από CPU, memory, database, downstreams και queueing policy. Shutdown είναι μέρος του ίδιου μοντέλου.

PROBE
Κάνε load test με I/O route, CPU route και slow stream μαζί. Μετά ξεκίνα rolling shutdown στο peak. Μέτρα queue time, p99, connection usage, RSS, forced requests και reconnect recovery. Επανάλαβε με ένα downstream στο όριο του timeout.

ΚΕΦΑΛΑΙΟ 27

Performance, allocation budgets και cache ownership

Optimization χωρίς phase timing και resource budget είναι αλλαγή πίστης. Η Sinatra εφαρμογή είναι αρκετά μικρή ώστε να μετρήσεις ακριβώς πού πήγε ο χρόνος.



Σχήμα 27.1 · Latency, allocations και cache growth χρειάζονται ξεχωριστά budgets και κοινό request context.

Το request latency χωρίζεται σε queue time, middleware, route matching, authorization, data access, serialization και body delivery. Measurement μόνο γύρω από το route block χάνει queueing και streaming. Measurement μόνο γύρω από `app.call` χάνει το χρόνο που ο server καταναλώνει το body.

Ένα Rack middleware μπορεί να καταγράψει application-return time και να τυλίξει το body για consumption time. Το wrapper πρέπει να διατηρεί το `close` contract:

```ruby
class TimedBody
  def initialize(body, clock:, observer:)
    @body = body
    @clock = clock
    @observer = observer
  end

  def each
    started = @clock.call(Process::CLOCK_MONOTONIC)
    @body.each { |chunk| yield chunk }
  ensure
    @observer.observe_body(@clock.call(Process::CLOCK_MONOTONIC) - started)
  end

  def close
    @body.close if @body.respond_to?(:close)
  end
end
```

Monotonic clock είναι για durations. Wall clock είναι για timestamps που συσχετίζονται μεταξύ συστημάτων. NTP adjustment μπορεί να κάνει wall-clock duration αρνητικό ή απότομα μεγάλο.

Το timing wrapper δεν πρέπει να καταπιεί exception ούτε να διπλοκλείσει body. Σε production βάλε histogram με bounded route labels. Raw path όπως `/users/93847` δημιουργεί unbounded cardinality. Το `env['sinatra.route']` δίνει το selected route αφού εκτελεστεί το match, όμως routes που αποτυγχάνουν πριν την επιλογή χρειάζονται fallback label.

Allocation budget

Ruby latency συχνά έρχεται από allocations και GC. Μέτρα allocated objects και bytes όπου επιτρέπει το profiler, αλλά θυμήσου ότι process-wide GC counters είναι noisy όταν τρέχουν concurrent requests. Για ακριβές before/after profile χρησιμοποίησε isolated process με representative fixture και αρκετό warmup.

Συνηθισμένες πηγές είναι repeated JSON transforms, regex creation μέσα σε route, μεγάλα intermediate arrays και template partials που ξαναφορτώνουν τα ίδια δεδομένα. Μην αντικαταστήσεις καθαρό code με mutable object reuse πριν δεις allocation flame graph. Ένα retained object είναι συνήθως χειρότερο από ένα short-lived object επειδή αυξάνει live heap.

Budget δεν είναι μόνο average. Όρισε p50, p95 και p99 ανά workload, maximum response bytes, allocation range και peak RSS. Δώσε χωριστό budget σε warm process και cold boot. Πρώτο request μπορεί να πληρώνει template compilation, autoload ή connection establishment.

Cache σημαίνει ownership

Κάθε cache χρειάζεται owner, key space, freshness rule, maximum size, eviction, invalidations και observability. Αν λείπει maximum, δεν είναι cache αλλά δεύτερη βάση δεδομένων που μεγαλώνει μέχρι restart.

Το Sinatra template cache είναι process-local Hash-like object. Στην 4.2.1 δηλώνεται ρητά non-thread-safe και unbounded. Τα compiled templates είναι χρήσιμο warm cache όταν το template set είναι finite. Αν template name ή options προέρχονται από request, το key space μπορεί να γίνει unbounded.

Σε development το `reload_templates` καθαρίζει το cache ανά request, κάτι που αλλάζει πλήρως τα performance characteristics.

Application cache πάνω από mutable records χρειάζεται versioned key ή explicit invalidation μετά το commit. Invalidation πριν το commit ανοίγει window όπου άλλο request ξαναγεμίζει stale value. Invalidation μετά το commit έχει μικρό window stale read. Αν το contract απαιτεί read-your-write, process cache ίσως δεν είναι σωστό boundary.

HTTP cache συχνά είναι φθηνότερη από server-side object cache. Strong ETag και `Cache-Control` αφήνουν client ή proxy να επαναχρησιμοποιήσει bytes χωρίς να κρατάς arbitrary Ruby objects. Όμως private identity-specific responses δεν πρέπει να γίνουν shared κατά λάθος.

Load test που λέει κάτι

Representative load έχει mix routes, payload sizes, cache hit ratios και think time. Closed-loop benchmark με σταθερούς clients μπορεί να κρύψει overload επειδή οι clients επιβραδύνουν όταν ο server επιβραδύνει. Open-loop arrival model αποκαλύπτει queue growth, αρκεί ο generator να μπορεί πραγματικά να κρατήσει το rate.

Warmup μέχρι να σταθεροποιηθούν code paths και pools. Κατάγραψε exact commit, Ruby, gems, Puma config, host shape, dataset και command. Άλλαξε μία υπόθεση κάθε φορά. Σύγκρινε confidence intervals ή τουλάχιστον πολλά runs, όχι ένα εντυπωσιακό best result.

Τα downstream limits μπαίνουν στο ίδιο report. Γρήγορος Sinatra worker που σπρώχνει περισσότερα concurrent queries σε saturated database μπορεί να μειώσει system throughput. Optimization είναι επιτυχία όταν βελτιώνει end-to-end service objective χωρίς να μεταφέρει μη ορατό κόστος αλλού.

SOURCE TRACE
Το Sinatra 4.2.1 κρατά template cache πάνω στο application prototype και το καθαρίζει όταν `reload_templates` είναι ενεργό. Το Rack response body μπορεί να καταναλωθεί μετά το `app.call`, άρα complete latency απαιτεί body-aware instrumentation.

ΠΑΓΙΔΑ
Μην βάζεις object ids, raw URLs, user ids ή exception messages σε metric labels. Η cardinality θα μετατρέψει την observability σε νέο outage.

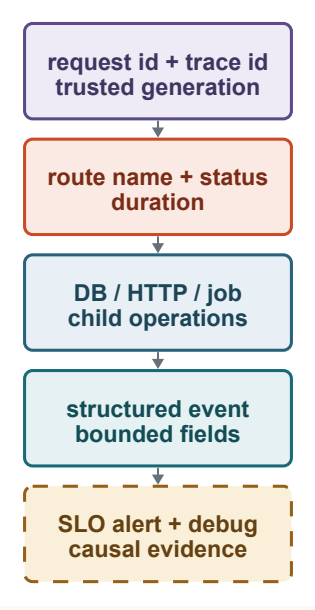
ΜΗΧΑΝΙΣΜΟΣ
Κάθε optimization έχει baseline, workload, budget και rollback. Κάθε cache έχει owner, όριο και freshness contract.

PROBE
Τρέξε cold και warm profiles, μετά steady open-loop load μέχρι saturation. Κατέγραψε queue time, service time, body time, allocations, GC, RSS, cache size και downstream utilization. Άλλαξε cache maximum και worker threads για να δεις αν το bottleneck μετακινείται αντί να εξαφανίζεται.

ΚΕΦΑΛΑΙΟ 28

Observability με request causality

Ένα log line δεν είναι observability. Χρειάζεσαι causal chain από το inbound request μέχρι database attempt, job, webhook και τελικό αποτέλεσμα.



Σχήμα 28.1 · Το request context περνά σε logs, metrics, traces και asynchronous work χωρίς sensitive payloads.

Ξεκίνα με request id. Αν δέχεσαι id από trusted proxy, έλεγξε μήκος και grammar. Αλλιώς δημιούργησε νέο. Επέστρεψέ το σε response header και πέρασέ το ρητά στα events. Μην αφήνεις arbitrary header να γίνει log injection ή τεράστιο field.

RUBY

```ruby
class RequestIdentity
  VALID = /\A[a-zA-Z0-9_.:-]{1,96}\z/

  def initialize(app)
    @app = app
  end

  def call(env)
    supplied = env["HTTP_X_REQUEST_ID"].to_s
    request_id = VALID.match?(supplied) ? supplied : SecureRandom.uuid
    env["request.id"] = request_id
    status, headers, body = @app.call(env)
    headers["x-request-id"] = request_id
    [status, headers, body]
  end
end
```

Σε πολλαπλά trust zones ίσως κρατήσεις και edge id και service-local id. Trace context ακολουθεί standard propagation, ενώ request id παραμένει βολικό για operator search. Δεν χρειάζεται να τα κάνεις το ίδιο identifier.

Structured event έχει σταθερό schema: timestamp, level, service, version, environment, request id, trace id, normalized route, method, status, duration, principal kind και error category. Το principal id μπορεί να είναι προσωπικό δεδομένο. Κατέγραψέ το μόνο αν υπάρχει συγκεκριμένη operational ανάγκη και retention policy. Ποτέ password, session cookie, authorization header, raw request body ή signed webhook secret.

Route identity και cardinality

Το Sinatra γράφει το selected route στο `env['sinatra.route']` αμέσως πριν εκτελέσει το route block. Η τιμή είναι method μαζί με compiled pattern. Είναι καλύτερο metric dimension από raw path, αλλά πρέπει να επιβεβαιώσεις ότι custom matchers έχουν bounded string representation. Ένα 404 δεν έχει selected route. Ένα middleware error πριν το Sinatra επίσης όχι.

Status μόνο δεν εξηγεί outcome. Ένα 409 expected concurrency conflict δεν έχει ίδια σημασία με 500 bug. Χρησιμοποίησε μικρό error taxonomy όπως validation, unauthorized, forbidden, conflict, dependency_timeout, dependency_rejected και internal. Exception class μένει σε logs/traces, όχι απαραίτητα ως metric label.

Πότε τελείωσε το response

Middleware code μετά το `@app.call` βλέπει status και headers, όχι απαραίτητα ολοκληρωμένο body. Για non-streaming application timing αυτό μπορεί να αρκεί. Για bytes sent, disconnect και body error χρειάζεται body wrapper ή server hook.

Το Rack 3 environment μπορεί να περιέχει `rack.response_finished`, array από callables που ο server καλεί μετά την επεξεργασία του response, σε reverse order. Callback παίρνει env, status, headers και error και δεν πρέπει να σηκώνει exception. Επειδή η παρουσία και η χρονική ακρίβεια εξαρτώνται από τον server, χρησιμοποίησέ το ως capability που ελέγχεται στο integration test. Μην το εφεύρεις όταν το key λείπει.

Ένα body wrapper μετρά bytes που έγιναν yield προς τον server. Δεν αποδεικνύει ότι ο remote client τα παρέλαβε. Server/proxy telemetry χρειάζεται για socket write failures και upstream disconnects. Αυτή η διάκριση αποτρέπει ψεύτικο «response delivered» event.

Traces, metrics και logs έχουν διαφορετική δουλειά

Metrics δείχνουν αν υπάρχει πρόβλημα: rate, errors, duration, saturation. Traces δείχνουν πού πέρασε χρόνος μέσα σε sampled causal path. Logs εξηγούν discrete state transitions με αρκετό context. Αν γράφεις κάθε SQL statement ως metric label ή κάθε request field ως log, χρησιμοποιείς λάθος εργαλείο.

Span boundaries μπαίνουν γύρω από πραγματικές remote operations, queue wait, transaction και serialization. Μην δημιουργείς δεκάδες spans για trivial Ruby helpers. Κατέγραψε retry attempt, timeout budget και peer. Error span δεν σημαίνει πάντα failed HTTP request, επειδή fallback μπορεί να πέτυχε.

Για jobs, αποθήκευσε causation id και correlation id στο versioned message. Νέο job execution παίρνει νέο trace ή linked span ανάλογα με το tracing model, αλλά διατηρεί τη σχέση με το initiating request. Webhook attempt έχει event id και attempt id. Έτσι duplicate delivery δεν φαίνεται σαν άσχετο incident.

Health, SLO και alerts

Liveness απαντά αν πρέπει να ξαναξεκινήσει το process. Readiness αν μπορεί να λάβει νέο traffic. Dependency diagnostics είναι ξεχωριστό, authenticated operator surface. Μην κάνεις liveness να εξαρτάται από κάθε downstream, γιατί μια database διακοπή θα προκαλέσει restart storm.

Service-level indicator μετρά user-visible outcome στο σωστό boundary. Για sync API μπορεί να είναι successful non-cancelled requests κάτω από latency threshold. Για async command είναι ο χρόνος μέχρι durable terminal state, όχι μόνο το γρήγορο 202. Alert πάνω σε burn rate και saturation έχει περισσότερο νόημα από alert σε κάθε μεμονωμένο exception.

Dashboard πρέπει να οδηγεί σε runbook. Από αυξημένο p99 ο operator βλέπει queue time, busy threads, database checkout, downstream latency, deploy marker και top bounded error categories. Αν χρειάζεται να μαντέψει ποιο chart σχετίζεται, η causal chain δεν έχει ολοκληρωθεί.

SOURCE TRACE
Το Sinatra 4.2.1 θέτει `env['sinatra.route']` όταν επιλέγεται route. Το Rack 3.2.7 ορίζει προαιρετικό `rack.response_finished` callback contract. Ο χρόνος μετά το `app.call` δεν ισούται από μόνος του με ολοκληρωμένη αποστολή body.

ΠΑΓΙΔΑ
Observability data είναι production data. Redaction, access control, retention και deletion policy ισχύουν όπως και στην κύρια βάση.

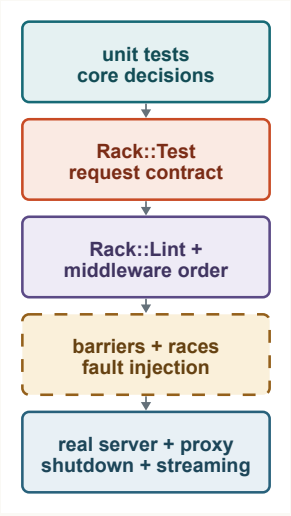
ΜΗΧΑΝΙΣΜΟΣ
Χτίσε bounded identifiers και taxonomy μία φορά και πέρασέ τα ρητά από request σε transaction, job και webhook. Τα τρία telemetry σήματα μοιράζονται context, όχι cardinality.

PROBE
Προκάλεσε route 404, validation 422, database timeout, body exception, client disconnect και duplicate job. Από ένα request id βρες όλη την causal chain. Assert ότι secrets και payloads δεν εμφανίζονται, ότι metric label count μένει bounded και ότι alert οδηγεί σε συγκεκριμένο runbook action.

ΚΕΦΑΛΑΙΟ 29

Tests, contract suites και fault injection

Τα δύσκολα bugs του Sinatra system ζουν στα boundaries και στα crash windows. Εκεί πρέπει να είναι deterministic και τα tests.



Σχήμα 29.1 · Η test matrix ανεβαίνει από pure policy σε Rack contract, real server και controlled failure.

Γρήγορα unit tests καλύπτουν parsers, policies, serializers και service objects χωρίς Rack. Request tests περνούν πραγματικό Rack environment μέσα από όλη τη middleware στοίβα. Adapter contract suites τρέχουν το ίδιο behavior πάνω σε fake και production adapter. Real-server tests ελέγχουν όσα δεν μπορεί να μοντελοποιήσει ένα in-process request helper: sockets, streaming, disconnect, signals και proxy headers.

Κράτα το application factory explicit ώστε κάθε test να παίρνει νέο app class και δικές του dependencies:

```ruby
def build_app(accounts:)
  Class.new(Sinatra::Base) do
    set :environment, :test
    disable :show_exceptions
    set :accounts, accounts

    get "/accounts/:id" do |id|
      account = settings.accounts.fetch(id)
      content_type :json
      JSON.generate(id: account.id, name: account.name)
    end
  end
end

repository = FakeAccounts.new("42" => Account.new("42", "Ada"))
response = Rack::MockRequest.new(build_app(accounts: repository)).get(
  "/accounts/42"
)
raise "unexpected status" unless response.status == 200
```

Anonymous class αποφεύγει routes και settings που μένουν από προηγούμενο test. Fake adapter δεν πρέπει να είναι απλώς Hash αν το production repository έχει transactions, version conflicts ή not-found semantics. Η κοινή contract suite ορίζει accepted input, return types, error taxonomy και atomicity.

Rack contract ως executable boundary

Βάλε `Rack::Lint` σε test composition. Θα βρει invalid status, header names, body chunks και lifecycle παραβάσεις. Κατανάλωσε όλο το body και κάλεσε `close` σε `ensure`, επειδή test που κοιτά μόνο το triple μπορεί να χάσει exception στην παραγωγή ή cleanup leak.

Κάθε route contract δοκιμάζει method, path, content type, auth state, accepted media type, valid body, malformed body, boundary size και expected error. Για HEAD έλεγξε empty delivered body και headers αντίστοιχα με GET. Για conditional request έλεγξε validators και 304 semantics. Για files έλεγξε traversal, symlink policy, missing file και Range αν το υποστηρίζεις.

Snapshot ολόκληρου JSON response είναι χρήσιμο μόνο για public schema. Για dynamic timestamps ή unordered maps γίνεται brittle. Προτίμησε parsed document assertions και explicit required/forbidden fields. Ένα schema compatibility test πρέπει να αποτυγχάνει όταν αφαιρεθεί field ή αλλάξει meaning χωρίς version.

Deterministic concurrency

Το `sleep` δεν ανοίγει αξιόπιστα race window. Βάλε barrier hook στο ακριβές σημείο μετά το read και πριν το conditional write. Δύο threads περιμένουν μέχρι να φτάσουν και τα δύο, μετά συνεχίζουν. Το production object παίρνει no-op hook, ενώ το test ελέγχει interleaving χωρίς probabilistic timing.

Database unique constraint και compare-and-swap δοκιμάζονται και με πραγματική database. In-memory fake με global mutex μπορεί να κρύψει το race. Εκτέλεσε το ίδιο idempotency key ταυτόχρονα, άλλο fingerprint με ίδιο key και δύο updates με ίδια expected version. Assert domain state, stored receipt και response για κάθε caller.

Με πολλούς Puma workers χρειάζεται process-level test ή external load driver. Thread-only test δεν αποδεικνύει ότι in-memory dedupe λειτουργεί σε cluster.

Fault injection με ονόματα

Κάθε reliability protocol έχει named cut points: `after_claim`, `after_domain_commit`, `after_provider_accept`, `before_outbox_mark`. Test hook στο σημείο σηκώνει controlled exception ή τερματίζει child process. Μετά restartάρεις και αφήνεις recovery να δουλέψει. Assert όχι μόνο άμεσο HTTP status, αλλά eventual durable state και maximum recovery age.

Timeout fake πρέπει να ξεχωρίζει connect timeout, read timeout πριν bytes και ambiguous timeout μετά remote acceptance. Generic `raise Timeout::Error` παντού δεν δοκιμάζει τη retry policy. Provider fake αποθηκεύει received idempotency key ώστε reconciliation να βρει το αποτέλεσμα.

To real-server layer

Εκκίνησε Puma σε ephemeral port με production-like config. Χρησιμοποίησε real HTTP client και, για lifecycle tests, raw socket. Έλεγξε slow upload, oversized headers, keep-alive, chunked body, client disconnect και graceful shutdown. Για SSE κράτα connection ανοιχτή, κάνε deploy signal και reconnect με cursor.

Reverse proxy behavior χρειάζεται ξεχωριστό environment ή containerized test μόνο αν αυτό είναι το production path. Unit test ενός header δεν αποδεικνύει ότι buffering έχει απενεργοποιηθεί. Το ίδιο ισχύει για forwarded host και scheme. Test trusted proxy chain και spoofed direct header.

Η suite πρέπει να είναι repeatable. Freeze wall clock μέσω injected clock, seed random data, χρησιμοποίησε monotonic deadlines και τύπωσε seed στην αποτυχία. Parallel tests δεν πρέπει να μοιράζονται global Sinatra app, ports ή database rows χωρίς namespace.

Από test σε release evidence

CI κρατά artifact με dependency lock, audit result, contract matrix και failing seed. Smoke test μετά το deploy ελέγχει version endpoint, critical write/read και telemetry emission χωρίς destructive production mutation. Canary metrics συγκρίνονται με baseline πριν μεγαλώσει traffic.

Ένα test suite δεν αποδεικνύει απουσία bugs. Δίνει επαναλήψιμες αποδείξεις για δηλωμένα contracts. Όταν βρεθεί incident, πρώτα γράψε characterization στο σωστό layer, μετά fix. Έτσι το production failure γίνεται μόνιμο μέρος του correctness packet.

SOURCE TRACE
Το Sinatra προτείνει Rack-based testing και το Rack παρέχει `MockRequest` και `Lint`. Το Rack body contract απαιτεί κατανάλωση και close. Οι process, socket και shutdown συμπεριφορές παραμένουν ευθύνη real handler tests.

ΠΑΓΙΔΑ
Fake που είναι πιο atomic, πιο γρήγορο ή πιο συνεπές από τον production adapter μπορεί να κάνει τη suite πράσινη για ακριβώς το bug που θα συμβεί.

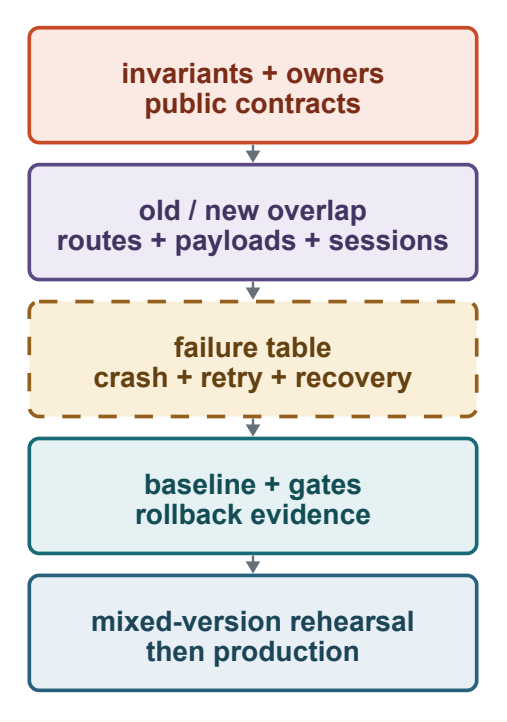
ΜΗΧΑΝΙΣΜΟΣ
Test στο χαμηλότερο layer που αποδεικνύει το contract, αλλά κράτα real-server tests για lifecycle. Κάνε τα race και crash windows ελεγχόμενα, όχι τυχαία.

PROBE
Τρέξε όλη τη fault matrix σε loop με fixed seeds και parallel CI. Μετά σκότωσε πραγματικό worker στα τέσσερα named cut points. Assert body close, pool return, idempotent recovery, bounded duplicate effects και traceable evidence για κάθε case.

ΚΕΦΑΛΑΙΟ 30

Extensions, upgrades και production correctness packet

Τελειωμένο Sinatra system δεν είναι αυτό που δεν αλλάζει. Είναι αυτό που μπορεί να αλλάξει framework, dependency και topology χωρίς να χάσει τα contracts του.



Σχήμα 30.1 · Το correctness packet συνδέει source, contracts, tests, operations και rollout evidence.

Το Sinatra extension model έχει δύο βασικά εργαλεία. `helpers` περιλαμβάνει instance methods που είναι διαθέσιμα σε routes και templates. `register` επεκτείνει την application class με module και καλεί `registered(app)` αν υπάρχει. Extensions μπορούν επίσης να ακούσουν hooks όπως `route_added`.

Ένα extension πρέπει να προσθέτει μικρό, τεκμηριωμένο public surface. Μην αντιγράψεις private methods από `Sinatra::Base` για να γλιτώσεις πέντε γραμμές. Αυτά είναι versioned implementation details.

```ruby
module JsonErrors
  module Helpers
    def json_error(code, message, status:)
      content_type :json
      halt status, JSON.generate(error: { code: code, message: message })
    end
  end

  def self.registered(app)
    app.helpers Helpers
    app.error Domain::NotFound do
      json_error("not_found", "Resource not found", status: 404)
    end
  end
end

class Api < Sinatra::Base
  register JsonErrors
end
```

Το module δεν κρατά mutable global state και μπορεί να registered σε πολλά app classes. Αν extension χρειάζεται state, βάλ' το σε setting του συγκεκριμένου app. Πρόσεξε ότι setting value που είναι mutable μπορεί να κληρονομηθεί ή να μοιραστεί. Φτιάξε νέο owner ανά app, πάγωσε configuration και όρισε cleanup.

Το `route_added` παίρνει verb, original path και block, αλλά όχι application ως argument. Extension singleton που θυμάται «το τελευταίο app» σπάει όταν registered σε δύο classes. Για route inventory είναι συχνά καθαρότερο να συλλέξεις `app.routes` μετά το boot σε read-only diagnostic, ή να δημιουργήσεις ξεχωριστό extension instance ανά app.

Public DSL ή middleware

Cross-cutting behavior που αφορά κάθε Rack app, όπως request id, body limits ή timing, ανήκει συνήθως σε middleware. Behavior που χρειάζεται Sinatra route scope, helpers και error handlers ταιριάζει σε extension. Domain policy δεν ανήκει σε κανένα από τα δύο. Μένει plain object ώστε να δοκιμάζεται χωρίς web framework.

Middleware ordering είναι μέρος του extension contract. Extension που κάνει σιωπηλά `app.use` μπορεί να βρεθεί μέσα ή έξω από security middleware ανάλογα με registration time. Τεκμηρίωσε required order και κάνε boot assertion όταν είναι κρίσιμο. Μην αλλάζεις route registry αφού αρχίσει traffic.

Upgrade ως controlled source migration

Ένα framework upgrade ξεκινά από exact current και target versions. Διάβασε changelog, gemspec constraints και source diff στα code paths που χρησιμοποιεί η εφαρμογή. Για Sinatra αυτά συνήθως είναι dispatch, settings inheritance, middleware build, sessions, protection, templates και streaming. Dependency upgrade του Rack ή Mustermann μπορεί να αλλάξει observable behavior χωρίς να αλλάξει πολύ το Sinatra code.

Πριν το update, πάρε characterization baseline:

1. route inventory και precedence cases
2. πλήρη middleware order με arguments

3. settings ανά environment και subclass
4. representative Rack triples και error bodies
5. cookie/session compatibility
6. template keys, cache growth και static file boundaries
7. concurrency, stream και shutdown probes

Μετά άλλαξε ένα dependency set, τρέξε contract suites και σύγκρινε behavior. Warnings και deprecations είναι work items, όχι output που κρύβεται. Αν αλλάζει cookie serializer ή encryption secret, rollout πρέπει να υποστηρίζει overlap: new code διαβάζει old και new, γράφει new, περιμένει retention window και μετά αφαιρεί old.

Canary παίρνει μικρό πραγματικό traffic με ίδιο topology. Παρακολούθησε route status mix, error taxonomy, p99, allocations, RSS, database pool, session failures και body errors. Rollback έχει exact trigger και δοκιμασμένη εντολή. Αν new code γράφει ασύμβατο durable data, binary rollback μόνο του δεν είναι ασφαλές. Χρειάζεται expand-contract data protocol.

Το production correctness packet

Το packet δεν είναι τεράστιο wiki. Είναι μικρό, versioned σύνολο evidence που συνοδεύει κάθε release.

1. System contract. Supported methods, media types, authentication, authorization, request limits, response schemas, idempotency και consistency.

2. Execution map. Route inventory, precedence, middleware order, error pipeline, file boundaries και settings provenance.

3. State protocols. Transaction boundaries, unique constraints, outbox, inbox, retry taxonomy, crash windows, secret rotation και data compatibility.

4. Capacity envelope. Puma workers/threads, pool sizes, downstream limits, request deadlines, queue policy, memory budget και streaming cap.

5. Operational proof. SLOs, dashboards, alerts, runbooks, health semantics, graceful shutdown result, backup restore evidence και reconciliation age.

6. Release evidence. Locked versions, source diff review, audit output, contract tests, fault matrix, load baseline, canary comparison και rollback conditions.

Κάθε statement δείχνει executable test, config ή measured result. «Thread-safe» χωρίς inventory και concurrency probe είναι ετικέτα. «Highly available» χωρίς failure mode και recovery time είναι ευχή. Γράψε και ό,τι δεν εγγυάται το σύστημα, όπως global event ordering ή exactly-once external effect.

Πότε έχεις φτάσει μέχρι τέλους

Δεν τελειώνεις όταν ξέρεις όλα τα Sinatra helpers. Τελειώνεις όταν μπορείς να προβλέψεις τι γίνεται σε κάθε boundary: ποιο middleware βλέπει το request, ποιο route κερδίζει, πότε δεσμεύεται το status, ποιος κλείνει το body, πού γίνεται commit, τι επιβιώνει crash, πόσο work χωρά, πώς γίνεται drain και ποια evidence αποδεικνύουν την απάντηση.

Το μικρό framework παραμένει πλεονέκτημα. Μπορείς να διαβάσεις το source που τρέχει, να σχεδιάσεις τη δική σου αρχιτεκτονική χωρίς κρυφό container και να κρατήσεις το production model αρκετά μικρό ώστε να ελεγχθεί. Η πειθαρχία δεν έρχεται από περισσότερο framework. Έρχεται από explicit contracts.

SOURCE TRACE
Στο Sinatra 4.2.1 τα `helpers`, `register`, `registered` και `route_added` αποτελούν το extension surface. Τα registries, το prototype και το middleware build παραμένουν implementation που πρέπει να ελέγχεται ξανά σε upgrade.

ΠΑΓΙΔΑ
Μην κάνεις monkey patch private Sinatra method ως μόνιμη πλατφόρμα. Αν είναι αναπόφευκτο, κλείδωσε exact version, βάλε characterization test και αφαίρεσέ το με συγκεκριμένο upgrade plan.

ΜΗΧΑΝΙΣΜΟΣ
Production mastery είναι η ικανότητα να μετατρέπεις source behavior σε explicit contracts και τα contracts σε επαναλήψιμη evidence πριν και μετά από αλλαγή.

PROBE
Πάρε το correctness packet μιας release και δώσ' το σε engineer που δεν έγραψε το service. Ζήτησέ του να προβλέψει route choice, crash recovery, maximum capacity και rollback. Μετά εκτέλεσε τα αντίστοιχα probes σε canary. Κάθε διαφορά είναι κενό στο packet ή στο σύστημα.