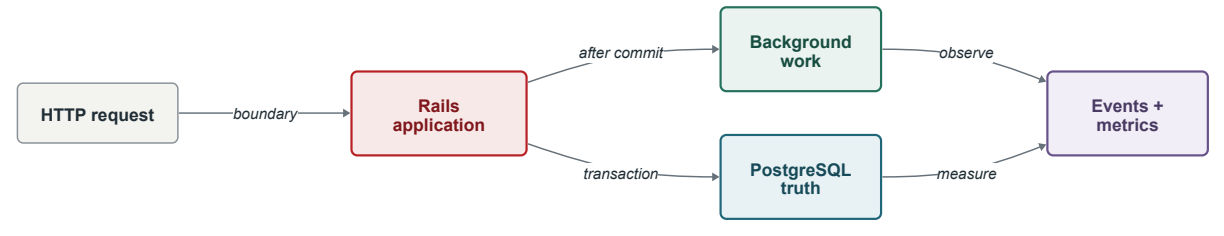
ΕΝΑ ΒΙΒΛΙΟ ΓΙΑ PRODUCTION RAILS

Rails πέρα

από το CRUD

Mid και advanced concepts για εφαρμογές που δέχονταιπταυτόχρονα requests, jobs και αλλαγές χωρίς downtime.



21 **Από το request στο deploy**

ΚΕΦΑΛΑΙΑ · RAILS 8.1 · ΕΚΔΟΣΗ 1.0

ΠΡΙΝ ΑΡΧΙΣΕΙΣ

Το framework δεν είναιπο δύσκολο κομμάτι.

Ξέρεις ήδη να φτιάχνεις ένα model, ένα controller και ένα migration. Το δύσκολο μέρος αρχίζει όταν δύο requests βλέπουν την ίδια τελευταία θέση, όταν ένα job τρέχει ξανά μετά από timeout ή όταν το νέο release γράφει δεδομένα που το παλιό δεν καταλαβαίνει.

Αυτό το βιβλίο οργανώνει εκείνες τις στιγμές. Δεν προσθέτει layers για να φαίνεται η εφαρμογή πιο σοβαρή. Δείχνει πρώτα την εγγύηση που χρειαζόμαστε και μετά το μικρότερο Rails, Ruby ή database mechanism που μπορεί να την κρατήσει.

Πώς να το διαβάσεις

Τα κεφάλαια 1 έως 10 είναι το mid επίπεδο. Δίνουν καθαρό mental model για requests, autoloading, Active Record, jobs και cache. Τα κεφάλαια 11 έως 21 μπαίνουν στα failure modes της production. Concurrency, replicas, outbox, observability, capacity και migrations χωρίς downtime.

Τα παραδείγματα ακολουθούν την ίδια εφαρμογή κρατήσεων. Δεν χρειάζεται να τη χτίσεις ολόκληρη. Σε κάθε κεφάλαιο υπάρχει ένας έλεγχος που μπορείς να κάνεις στη δική σου εφαρμογή.

Έκδοση αναφοράς

Η έκδοση 1.0 ελέγχθηκε με Rails 8.1.3.1 documentation τον Σεπτέμβριο του 2026. Όπου ένα feature είναι νέο στο Rails 8.1, σημειώνεται. Όπου η συμπεριφορά ανήκει στον PostgreSQL, στον queue adapter ή στην υποδομή, δεν αποδίδεται στο framework.

Το βιβλίο χρησιμοποιεί απλό κώδικα, συγκεκριμένα failure modes και επίσημες πηγές. Οι μετρήσεις μιας δικής σου εφαρμογής έχουν πάντα τον τελευταίο λόγο.

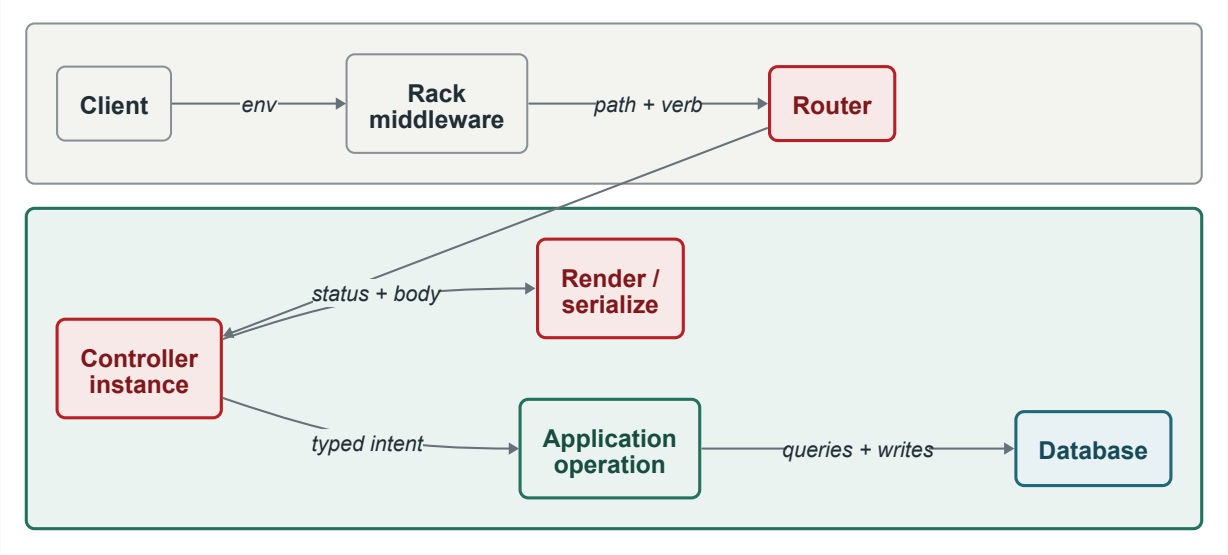
Ο ΧΑΡΤΗΣ ΤΟΥ ΒΙΒΛΙΟΥ

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1

Το request ως διαδρομή

Ένα controller action είναι ένα μικρό σημείο μέσα σε μεγαλύτερη διαδρομή. Αν δεν βλέπεις τη διαδρομή, βάζεις ευθύνες στο πρώτο αρχείο που έτυχε να ανοίξεις.



Σχήμα 1.1 · Η διαδρομή ενός request από το Rack μέχρι το render.

Πριν από το controller

Ο application server παραδίδει στο Rack ένα `env`. Το middleware stack μπορεί να αναγνωρίσει session, cookies, request id, host, method και content type πριν τρέξει ο router. Ο router αντιστοιχίζει verb και path σε controller action. Μόνο τότε δημιουργείται το controller instance του συγκεκριμένου request.

Αυτή η σειρά έχει πρακτική σημασία. Ένα authorization middleware δεν έχει απαραίτητα το domain object που θα φορτώσει το action. Ένα `before_action` έχει τα controller params αλλά μπορεί να αρχίσει άθελά του queries. Ένα model δεν πρέπει να ξέρει αν η εντολή ήρθε από HTTP, job ή console.

Δες την πραγματική στοίβα αντί να τη μαντεύεις.

TERMINAL

```
bin/rails middleware
bin/rails routes --expanded
```

Το middleware είναι κατάλληλο για cross-cutting συμπεριφορά που ισχύει σε κάθε σχετικό request. Request ids, compression, host checks και γενική καταγραφή ταιριάζουν εκεί. Η απόφαση «επιτρέπεται αυτός ο χρήστης να επιβεβαιώσει αυτή την κράτηση;» χρειάζεται φορτωμένα domain δεδομένα και ανήκει πιο κοντά στο action.

Το controller μεταφράζει πρωτόκολλα

Ένα καθαρό controller μετατρέπει HTTP input σε application input και το application result ξανά σε HTTP. Δεν αποφασίζει πώς κλειδώνεται μια θέση, δεν συντονίζει πέντε writes και δεν στέλνει email μέσα στο request.

```ruby
class ReservationsController < ApplicationController
  def confirm
    result = Reservations::Confirm.call(
      reservation_id: params.require(:id),
      actor: Current.user,
      idempotency_key: request.request_id
    )

    if result.success?
      render json: { reservation_id: result.reservation.id }, status: :ok
    else
      render json: { error: result.code }, status: result.http_status
    end
  end
end
```

Το action παραμένει αρκετά συγκεκριμένο ώστε να φαίνεται το HTTP contract. Δεν κρύβει το status code σε callback και δεν επιστρέφει Active Record object που ο serializer θα περιηγηθεί ανεξέλεγκτα. Η operation μπορεί να κληθεί και από job ή command line χωρίς ψεύτικο request.

Τα params παραμένουν μη έμπιστα ακόμα και μετά το routing. Το `params.expect` ή το `params.require(...).permit(...)` ορίζει το επιτρεπτό σχήμα. Δεν κάνει authorization και δεν μετατρέπει από μόνο του ένα string σε έγκυρο domain value. Το `"0"`, μια άγνωστη timezone και ένα τεράστιο array εξακολουθούν να χρειάζονται σαφή χειρισμό.

Render, redirect και control flow

Το `render` και το `redirect_to` ορίζουν την απόκριση. Δεν αποτελούν γενικό `return` από τη μέθοδο. Αν υπάρχει κώδικας που δεν πρέπει να τρέξει μετά από redirect, σταμάτησέ τον ρητά.

```ruby
class Admin::EventsController < AdminController
  def publish
    return redirect_to(events_path, alert: "Forbidden") unless allowed?

    Events::Publish.call(event: Event.find(params.require(:id)))
    redirect_to events_path, notice: "Published"
  end

  private

  def allowed?
    Current.user&.admin?
  end
end
```

Τα `around_action` είναι ισχυρά επειδή τυλίγουν όλο το action. Αυτό είναι και ο λόγος να χρησιμοποιούνται φειδωλά. Transaction γύρω από ολόκληρο request κρατά connection και locks όσο γίνεται rendering ή εξωτερικό I/O. Συνήθως η transaction boundary πρέπει να βρίσκεται γύρω από τα συγκεκριμένα database writes στην application operation.

Request-local state

Το `ActiveSupport::CurrentAttributes` βοηθά να περνά request-local context όπως user, account και request id. Δεν κάνει μια κρυφή global εξάρτηση καλή. Αν ένα domain operation χρειάζεται actor, προτίμησε να τον περάσεις ρητά. Κράτα το `Current` για context που πραγματικά ακολουθεί όλο το request και φρόντισε να μην αποθηκεύεις mutable objects που θα τροποποιηθούν από διαφορετικά σημεία.

Rails τυλίγει κάθε request με τον Executor. Ανάμεσα στα άλλα, αυτό οργανώνει το query cache και την επιστροφή database connections στο pool. Ένα χειροποίητο `Thread.new` δεν αποκτά αυτόματα σωστό lifecycle επειδή ξεκίνησε μέσα από controller. Αν υπάρχει πραγματική ανάγκη για δικό σου thread, ο κώδικας της εφαρμογής πρέπει να τυλιχτεί με `Rails.application.executor.wrap`. Στις περισσότερες περιπτώσεις ένα job είναι καθαρότερο boundary.

```ruby
class ParallelProbe
  def self.call
    Thread.new do
      Rails.application.executor.wrap do
        HealthSnapshot.capture
      end
    end
  end
end
```

Errors ως μέρος του contract

Το `rescue_from` είναι χρήσιμο όταν μια κατηγορία application error έχει μία σταθερή HTTP μετάφραση. Δεν είναι χώρος για να καταπιείς άγνωστα exceptions. Ένα `ActiveRecord::RecordNotFound` μπορεί να γίνει 404. Ένα invariant violation μπορεί να γίνει 409. Ένα απρόσμενο `NoMethodError` πρέπει να μείνει 500 και να αναφερθεί.

RUBY

```ruby
class ApplicationController < ActionController::Base
  rescue_from Reservations::Conflict do |error|
    render json: { error: error.code }, status: :conflict
  end
end
```

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Το controller κατέχει το HTTP contract. Η application operation κατέχει το use case. Η database κατέχει τις εγγυήσεις που πρέπει να ισχύουν για κάθε writer.

Ο πρακτικός χάρτης

Για ένα αργό ή μπερδεμένο endpoint, γράψε τη διαδρομή σε επτά γραμμές. Middleware, route, filters, action, operation, database calls και render. Σημείωσε πού κρατιέται connection, πού γίνεται authorization και πού μπορεί να υπάρξει εξωτερικό I/O. Συχνά το πρόβλημα φαίνεται πριν ανοίξεις profiler.

Η επίσημη βάση για αυτή τη διαδρομή είναι το Rails on Rack, το Action Controller Overview και το Threading and Code Execution.

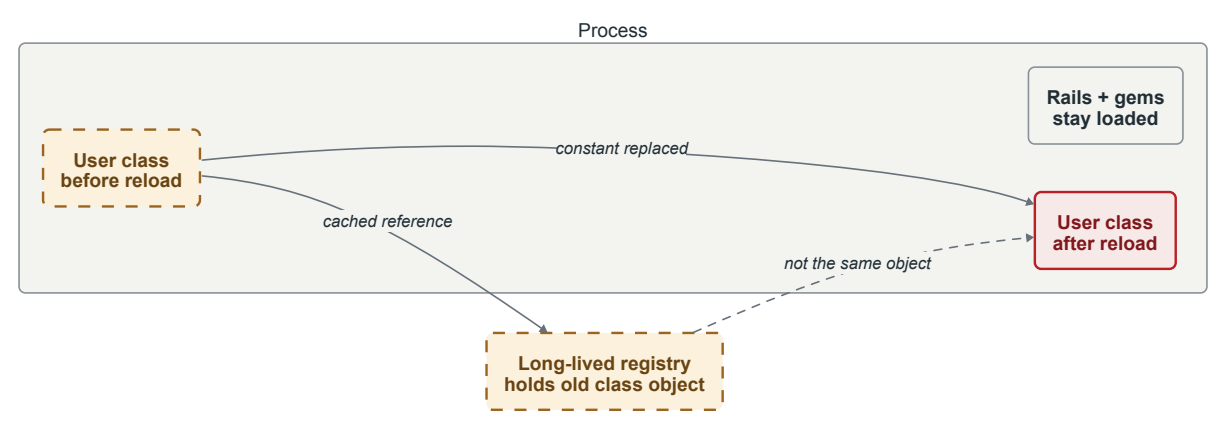
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Διάλεξε ένα endpoint που αλλάζει δεδομένα. Μέτρα πόσα callbacks και queries τρέχουν πριν μπεις στο action, αν υπάρχει εξωτερικό I/O μέσα σε transaction και αν το HTTP status αποφασίζεται σε ένα ορατό σημείο. Γράψε τη διαδρομή χωρίς να αλλάξεις ακόμα κώδικα.

ΚΕΦΑΛΑΙΟ 2

Zeitwerk, boot και reload

To autoloading δεν είναι μαγεία. Είναι συμβόλαιο ανάμεσα στο path, στο constant name και στη διάρκεια ζωής του process.



Σχήμα 2.1 · Ένα reload δημιουργεί νέο class object ενώ ένα παλιό registry μπορεί να κρατά το προηγούμενο.

Το path είναι μέρος του API

Στο Zeitwerk mode το `app/services/reservations/confirm.rb` πρέπει να ορίζει `Reservations::Confirm`. Ο φάκελος είναι namespace και το filename καθορίζει το constant. Ένα mismatch μπορεί να μένει κρυφό όσο ένα αρχείο δεν φορτώνεται στο development και να εμφανιστεί όταν το production κάνει eager load.

RUBY

```ruby
# app/services/reservations/confirm.rb
module Reservations
  class Confirm
    def self.call(...)
      new(...).call
    end
  end
end
```

Οι νέοι υποφάκελοι κάτω από `app/` γίνονται autoload paths αυτόματα. Δεν χρειάζεται να προσθέσεις `app/services` με το χέρι. Το `lib/` έχει διαφορετικό ρόλο επειδή συχνά περιέχει tasks, generators και αρχεία που δεν είναι reloadable application code. Σε σύγχρονο Rails χρησιμοποίησε `config.autoload_lib(ignore: ...)` όταν θέλεις να το εντάξεις ρητά.

```ruby
module Seatly
  class Application < Rails::Application
    config.autoload_lib(ignore: %w[assets tasks generators])
  end
end
```

Το πιο φθηνό production check είναι επίσης ένα από τα καλύτερα CI checks.

```terminal
bin/rails zeitwerk:check
```

Reloadable δεν σημαίνει mutable global

Στο development το Rails αφαιρεί constants εφαρμογής και τα φορτώνει ξανά. Το νέο `User` είναι διαφορετικό Ruby class object από το παλιό `User`. Αν ένα gem, initializer ή global registry κράτησε reference στο παλιό object, το reload δεν μπορεί να αλλάξει εκείνο το reference.

Αυτό εξηγεί bugs που μοιάζουν παράλογα. Ένα serializer registry συνεχίζει να καλεί παλιά μέθοδο. Ένα decorator φαίνεται να αγνοεί την τελευταία αλλαγή. Ένα `User.new.class == User` μπορεί θεωρητικά να γίνει false όταν ανακατεύονται αντικείμενα πριν και μετά το reload.

Κώδικας που εκτελείται στο boot και αναφέρεται σε reloadable constants πρέπει να μπει σε `Rails.application.config.to_prepare`. Το block τρέχει στο boot και ξανά πριν από reload. Πρέπει να είναι idempotent γιατί μπορεί να εκτελεστεί δύο φορές κατά την εκκίνηση για ιστορικούς λόγους σε ορισμένες διαμορφώσεις.

```ruby
Rails.application.config.to_prepare do
  ReservationPresenter.reset_registry!
  ReservationPresenter.register(ConfirmedReservationPresenter)
end
```

Η εναλλακτική είναι να βάλεις κώδικα που πρέπει να μείνει σταθερός σε `autoload_once_paths`. Αυτό ταιριάζει σε class objects που αποθηκεύονται μέσα σε framework registries τα οποία δεν κάνουν reload, όπως custom Active Job serializers. Δεν είναι γενική λύση για κάθε περίεργο reload bug.

Boot, eager load και αρχική κατάσταση

Στο production το συνηθισμένο setup έχει `config.enable_reloading = false` και `config.eager_load = true`. Όλα τα application constants φορτώνονται πριν το process δεχτεί traffic. Αυτό αποκαλύπτει naming errors νωρίς και επιτρέπει καλύτερη συμπεριφορά σε copy-on-write process models.

Ένα initializer πρέπει να ρυθμίζει framework και gem configuration. Δεν πρέπει να κάνει αργό network call, να διαβάζει μεταβαλλόμενα business rows ή να ξεκινά thread χωρίς ξεκάθαρο shutdown lifecycle. Κάθε τέτοια ενέργεια επαναλαμβάνεται σε console, rake task, web process και πιθανώς job worker.

Αν χρειάζεσαι δεδομένα κατά το boot, ρώτησε πρώτα αν είναι πραγματικά configuration. Τα credentials και environment settings είναι configuration. Η λίστα ενεργών accounts είναι application data και πρέπει να διαβάζεται όταν χρειάζεται ή να ενημερώνεται μέσω cache με σαφές invalidation.

Threads και Executor

Ο Puma εξυπηρετεί πολλά requests σε threads μέσα στο ίδιο process. Τα class objects και τα globals είναι κοινά. Το controller instance είναι ξεχωριστό για κάθε request. Η ασφάλεια προκύπτει όταν η κοινή κατάσταση είναι immutable ή προστατεύεται σωστά, όχι επειδή «το Rails είναι thread-safe» ως γενική φράση.

Rails τυλίγει requests, jobs και τα περισσότερα entry points με τον Executor. Ο Executor επιστρέφει connections στο pool, διαχειρίζεται query-cache lifecycle και συνεργάζεται με το reload interlock. Library code που καλεί application code από δικό του thread πρέπει να χρησιμοποιεί αυτό το boundary.

```ruby
class FeedConsumer
  def consume(message)
    Rails.application.executor.wrap do
      IncomingMessages::Apply.call(message: message)
    end
  end
end
```

Μην χρησιμοποιείς το Reloader σε child thread. Ένα reload χρειάζεται να περιμένει να τελειώσουν τα threads που εκτελούν application code. Αν το child περιμένει reload ενώ ο parent περιμένει το child, έχεις deadlock από το ίδιο το lifecycle που προσπάθησες να προστατεύσεις.

> **ΠΡΟΣΟΧΗ**
>
> Το `require_dependency` δεν είναι καθημερινό αντίδοτο για λάθος filenames ή λάθος namespaces. Διόρθωσε πρώτα το Zeitwerk contract. Χειροκίνητο loading χρειάζεται μόνο σε ειδικά integration boundaries.

Ένα καθαρό loading audit

Τρέξε `bin/rails zeitwerk:check` και μετά boot με production configuration σε ασφαλές local ή CI περιβάλλον. Αναζήτησε initializers που κάνουν queries, network calls ή αποθηκεύουν application classes σε constants. Έλεγξε custom threads και registries. Το ζητούμενο δεν είναι να μην υπάρχει κανένα. Είναι να είναι ξεκάθαρο ποιος τα δημιουργεί, ποιος τα σταματά και αν κρατούν reloadable objects.

Η πλήρης επίσημη αναφορά βρίσκεται στο Autoloading and Reloading Constants και στο Threading and Code Execution.

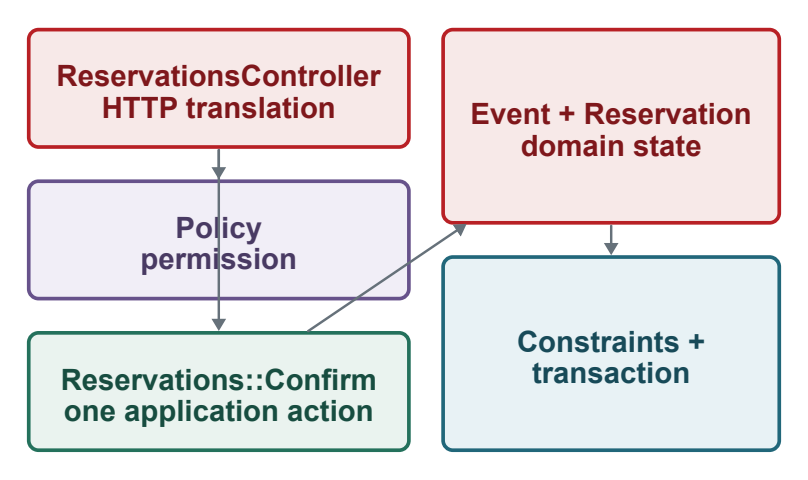
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Βρες ένα custom directory κάτω από `app`, ένα initializer που αναφέρει application constant και ένα long-lived registry. Επιβεβαίωσε το filename με `zeitwerk:check`. Μετά σημείωσε ποιο από τα τρία πρέπει να είναι reloadable και ποιο πρέπει να φορτώνεται μόνο μία φορά.

ΚΕΦΑΛΑΙΟ 3

Καθαρά application boundaries

Δεν χρειάζεσαι περισσότερα layers. Χρειάζεσαι ένα ορατό σημείο για κάθε σημαντική απόφαση.



Σχήμα 3.1 · Ένα application operation συνδέει HTTP, permission, domain state και database guarantees.

Το πρόβλημα με το «fat»

Το «fat model, skinny controller» ήταν χρήσιμη διόρθωση σε controllers που έκαναν τα πάντα. Δεν είναι αρχιτεκτονική για κάθε εφαρμογή. Ένα Active Record model μπορεί να καταλήξει να στέλνει email, να καλεί payment provider, να αποφασίζει permissions και να γνωρίζει πέντε άσχετα use cases. Μετά ένα απλό `reservation.save!` αποκτά αόρατες συνέπειες.

Το αντίθετο άκρο είναι ο φάκελος `services` με εκατοντάδες κλάσεις που έχουν μόνο `call`. Αν κάθε γραμμή Ruby γίνει service object, η πλοήγηση χειροτερεύει χωρίς να εμφανίζεται νέο boundary.

Ξεκίνα από την application action. «Επιβεβαίωσε αυτή την κράτηση για αυτόν τον actor με αυτό το idempotency key». Η action έχει αρχή, αποτέλεσμα και transaction boundary. Αυτό δικαιολογεί ένα object.

RUBY

```ruby
module Reservations
  class Confirm
    Result = Data.define(:reservation, :code) do
      def success?
        code == :confirmed
      end

      def http_status
        success? ? :ok : :unprocessable_entity
```

```
      end
    end

    def self.call(reservation_id:, actor:, idempotency_key:)
      new(
        reservation_id: reservation_id,
        actor: actor,
        idempotency_key: idempotency_key
      ).call
    end

    def initialize(reservation_id:, actor:, idempotency_key:)
      @reservation_id = reservation_id
      @actor = actor
      @idempotency_key = idempotency_key
    end

    def call
      reservation = Reservation.find(@reservation_id)
      return Result.new(reservation:, code: :forbidden) unless allowed?
(reservation)

      reservation.confirm!(idempotency_key: @idempotency_key)
      Result.new(reservation:, code: :confirmed)
    end

    private

    def allowed?(reservation)
      reservation.user_id == @actor.id
    end
  end
end
```

Το παράδειγμα δεν είναι ολοκληρωμένη concurrency λύση. Αυτό θα έρθει στο κεφάλαιο 11. Δείχνει όμως ιδιοκτησία. Το controller δεν αποφασίζει permission και το model δεν γνωρίζει status codes.

Behavior στο model, orchestration στην operation

Ένα model είναι καλό μέρος για behavior που ανήκει στο ίδιο το entity και πρέπει να ισχύει όπου κι αν κληθεί. Η μετάβαση από `pending` σε `confirmed` και η άρνηση μετάβασης από `cancelled` είναι domain behavior. Η σειρά «κλείδωσε event, επιβεβαίωσε reservation, γράψε outbox event» είναι orchestration πολλών objects και ταιριάζει στην operation.

RUBY

```
class Reservation < ApplicationRecord
  class InvalidTransition < StandardError; end

  def confirm!(idempotency_key:)
    raise InvalidTransition, "cancelled reservation" if cancelled?

    update!(status: :confirmed, idempotency_key: idempotency_key)
  end
end
```

Το model API πρέπει να κάνει την έγκυρη ενέργεια ευκολότερη από την τυχαία μεταβολή πεδίων. Αυτό δεν σημαίνει ότι θα απαγορεύσεις κάθε setter. Σημαίνει ότι οι σημαντικές μεταβάσεις έχουν όνομα και tests.

Form objects και Active Model

Ένα form object είναι χρήσιμο όταν το input δεν αντιστοιχεί σε ένα row. Μπορεί να συνδυάζει timezone, terms acceptance και στοιχεία δύο records χωρίς να φορτώνει το Active Record model με transient fields. Με `ActiveModel::Model` αποκτά validations και error messages χωρίς persistence.

```ruby
class ReservationRequest
  include ActiveModel::Model
  include ActiveModel::Attributes

  attribute :event_id, :integer
  attribute :seats, :integer
  attribute :terms_accepted, :boolean

  validates :event_id, presence: true
  validates :seats, inclusion: { in: 1..6 }
  validates :terms_accepted, acceptance: true
end
```

Το form object προστατεύει την ποιότητα του μηνύματος προς τον χρήστη. Η database constraint εξακολουθεί να προστατεύει την τελική invariant από άλλους writers και races.

Query objects που παραμένουν composable

Ένα query object αξίζει όταν μια επιχειρηματική έννοια συνδυάζει joins, filters και permission scope σε πολλά endpoints. Αν επιστρέφει `ActiveRecord::Relation`, ο caller μπορεί να προσθέσει pagination ή ordering χωρίς να φορτώσει νωρίς τα rows.

```ruby
module Events
  class VisibleTo
    def self.call(user:, relation: Event.all)
      return relation.published if user.guest?

      relation.where(account_id: user.account_id)
        .merge(Event.published.or(Event.draft.where(owner_id: user.id)))
    end
  end
end
```

Το object δέχεται relation αντί να ξεκινά πάντα από `Event.all`. Αυτό το κάνει composable και ευκολότερο να δοκιμαστεί μέσα σε περιορισμένο scope. Αν ένα query είναι μία καθαρή, επαναχρησιμοποιήσιμη συνθήκη του model, ένα named scope παραμένει η απλούστερη επιλογή.

Dependencies στα εξωτερικά boundaries

Ένα gateway προς payment ή email provider πρέπει να φαίνεται ως dependency. Δεν χρειάζεται container. Ένα keyword argument με λογικό default αρκεί συχνά.

```ruby
module Notifications
  class Deliver
    def initialize(provider: EmailProvider.new)
      @provider = provider
    end

    def call(message:)
      @provider.deliver(message)
    end
  end
end
```

Στο test δίνεις fake που κρατά τις κλήσεις. Στην production δίνεις adapter που μεταφράζει provider errors σε μικρό, σταθερό application vocabulary. Έτσι μια αλλαγή provider δεν διαρρέει exceptions και response objects σε όλο το app.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Δημιούργησε object όταν μπορείς να του δώσεις μία συγκεκριμένη ευθύνη, σαφές input, σαφές αποτέλεσμα και λόγο να αλλάζει ανεξάρτητα. Το όνομα του φακέλου δεν δημιουργεί boundary από μόνο του.

Πώς κρίνεις μια διάσπαση

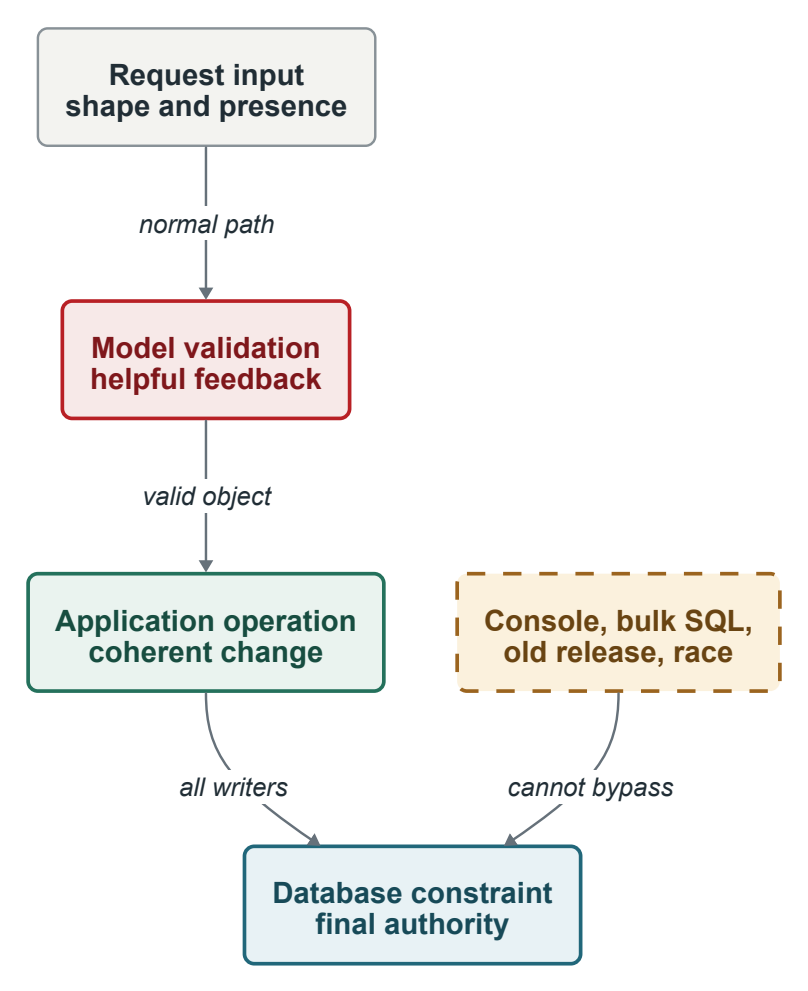
Μια καλή διάσπαση μειώνει τον αριθμό των πραγμάτων που πρέπει να κρατάς στο κεφάλι σου για να αλλάξεις ένα use case. Το controller test ελέγχει HTTP. Το operation test ελέγχει business branches. Το model test ελέγχει transitions. Το database integration test ελέγχει constraint και locking. Αν μετά τη διάσπαση κάθε αλλαγή απαιτεί να πειράξεις πέντε wrappers που απλώς προωθούν arguments, έχεις μεταφέρει γραμμές χωρίς να κερδίσεις ανεξαρτησία.

> **ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ**
>
> Πάρε ένα action με πάνω από ένα write. Γράψε σε μία πρόταση το use case, την invariant και το εξωτερικό effect. Αν δεν υπάρχουν ήδη ορατά boundaries, μετέφερε μόνο το orchestration σε μία operation. Μην αλλάξεις models, policy και serializers στην ίδια κίνηση.

ΚΕΦΑΛΑΙΟ 4

Invariants σε model και database

Validation σημαίνει «δώσε καλό feedback πριν γράψεις».
Constraint σημαίνει «αυτό δεν επιτρέπεται να υπάρχει ποτέ».



Σχήμα 4.1 · Η ίδια invariant περνά από input, model, operation και τελικά database constraint.

Ποιον writer εμπιστεύεσαι

Ένα Rails validation τρέχει όταν η συγκεκριμένη διαδρομή persistence το ζητήσει. Δεν τρέχει για `update_all`, `insert_all`, raw SQL, παλιότερο release ή διαφορετική εφαρμογή που γράφει στην ίδια βάση. Ακόμα και στην κανονική διαδρομή, δύο processes μπορούν να περάσουν το ίδιο uniqueness validation πριν γράψει κάποιο από τα δύο.

Η database βλέπει όλους τους writers. Για αυτό οι αδιαπραγμάτευτες invariants χρειάζονται constraint. Το model validation παραμένει χρήσιμο επειδή παράγει καλό μήνυμα χωρίς να περιμένει database exception.

RUBY

```ruby
class Reservation < ApplicationRecord
  belongs_to :event
  belongs_to :user

  validates :idempotency_key, presence: true, uniqueness: true
  validates :seats, numericality: { only_integer: true, greater_than: 0 }
end
```

RUBY

```ruby
class ProtectReservations < ActiveRecord::Migration[8.1]
  def change
    change_column_null :reservations, :idempotency_key, false
    add_index :reservations, :idempotency_key, unique: true
    add_check_constraint :reservations, "seats > 0", name:
"reservations_seats_positive"
    add_foreign_key :reservations, :events
    add_foreign_key :reservations, :users
  end
end
```

Το unique index είναι η εγγύηση. Η validation είναι η γρήγορη, φιλική πρώτη γραμμή. Ο application κώδικας πρέπει επίσης να μεταφράζει το πιθανό `ActiveRecord::RecordNotUnique` σε αποτέλεσμα που καταλαβαίνει ο caller. Το exception δεν είναι «απίθανο edge case». Είναι ο τρόπος με τον οποίο η βάση κερδίζει ένα πραγματικό race.

NULL δεν είναι απλό κενό

Στο SQL το `NULL` σημαίνει άγνωστη τιμή. Δεν είναι empty string και οι συγκρίσεις του ακολουθούν three-valued logic. Ένα unique index μπορεί να επιτρέπει πολλές NULL τιμές ανάλογα με τη database και τις επιλογές του index. Αν η business invariant λέει ότι κάθε confirmed reservation έχει reference, χρειάζεσαι `NOT NULL` στο σωστό στάδιο της μετάβασης.

Μην προσθέτεις `NOT NULL` σε μεγάλο table πριν βεβαιωθείς ότι τα παλιά rows έχουν τιμή και ότι όλα τα ενεργά releases γράφουν το πεδίο. Η ασφαλής σειρά είναι expand, backfill, verification και μετά constraint. Θα τη δούμε πλήρως στο κεφάλαιο 21.

Foreign keys και deletion semantics

Το `belongs_to :event` εκφράζει association στο Rails. Το foreign key εκφράζει referential integrity στη βάση. Χρειάζεσαι και τα δύο όταν ένα reservation χωρίς event είναι άκυρη κατάσταση.

Το `dependent: :destroy` εκτελεί Ruby callbacks για κάθε child και μπορεί να είναι αργό. Το `dependent: :delete_all` παρακάμπτει callbacks. Το database `ON DELETE CASCADE` εφαρμόζεται σε κάθε writer αλλά μεταφέρει την ευθύνη στη βάση. Δεν υπάρχει μία σωστή επιλογή για όλα. Γράψε ποιος κατέχει το deletion workflow, αν υπάρχουν external effects και πόσο μεγάλο μπορεί να γίνει το fan-out.

Σε polymorphic association η βάση δεν μπορεί να βάλει ένα απλό foreign key από ένα column προς πολλά πιθανά tables. Αυτό είναι πραγματικό integrity cost, όχι λόγος να μην χρησιμοποιήσεις ποτέ polymorphism. Αν η σχέση είναι κρίσιμη, ένα explicit join model με κανονικά foreign keys μπορεί να είναι καθαρότερο.

State ως μετάβαση, όχι ως string

Ένα enum κάνει τις τιμές ευανάγνωστες αλλά δεν ορίζει μόνο του επιτρεπτές μεταβάσεις. Το `pending -> confirmed` μπορεί να επιτρέπεται και το `cancelled -> confirmed` να απαγορεύεται. Δώσε όνομα στη μετάβαση και βάλε την απόφαση εκεί όπου φαίνεται.

RUBY

```ruby
class Reservation < ApplicationRecord
  enum :status, { pending: 0, confirmed: 1, cancelled: 2 }, default: :pending

  def confirm!
    raise InvalidTransition, "already cancelled" if cancelled?

    update!(status: :confirmed, confirmed_at: Time.current)
  end
end
```

Για πολύπλοκο state machine, ένα transition table ή ξεχωριστό domain object μπορεί να αξίζει περισσότερο από πολλά callbacks. Μην εγκαταστήσεις abstraction πριν δεις πραγματική πολυπλοκότητα. Δύο καθαρές methods είναι συχνά αρκετές.

Callbacks με περιορισμένη εμβέλεια

Τα callbacks ταιριάζουν σε αλλαγές που είναι εγγενείς στο lifecycle του ίδιου record. Normalization πριν από validation και ενημέρωση ενός derived local field είναι λογικά παραδείγματα. Η αποστολή email, το charge και η ενημέρωση search index ξεφεύγουν από τη database transaction και αποκρύπτουν external effects πίσω από ένα `save!`.

Το Rails προσφέρει `normalizes` για deterministic normalization. Πρόσεξε να μην καταστρέψεις πληροφορία που χρειάζεται ο χρήστης.

RUBY

```ruby
class User < ApplicationRecord
  normalizes :email, with: ->(value) { value.strip.downcase }
  validates :email, presence: true
end
```

Αν το normalized email πρέπει να είναι μοναδικό, ο unique index πρέπει να αντιστοιχεί στην αποθηκευμένη normalized μορφή. Διαφορετικά η Ruby και η βάση ελέγχουν διαφορετικές έννοιες.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Βάλε helpful rejection στο model και final rejection στη database. Μετά δοκίμασε ξεχωριστά και τις δύο διαδρομές.

Error vocabulary

Μην αφήνεις database adapter exceptions να ανεβαίνουν ως τυχαίο API contract. Rescue στο στενό σημείο όπου ξέρεις ποιο constraint παραβιάστηκε. Μετά επέστρεψε `duplicate_request`, `sold_out` ή άλλο application result. Μην κάνεις global rescue κάθε `StatementInvalid`, επειδή τότε κρύβεις schema bugs μαζί με αναμενόμενα conflicts.

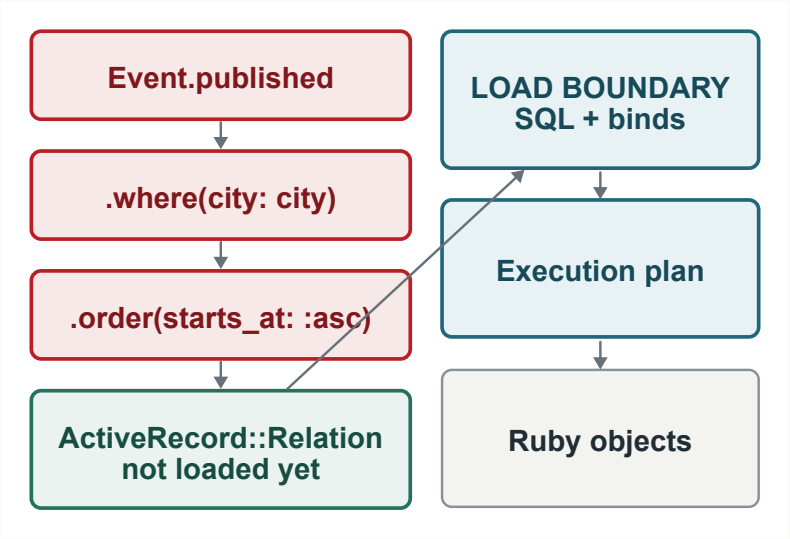
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Διάλεξε τρεις validations από κρίσιμο model. Για καθεμία βρες αν υπάρχει αντίστοιχο `NOT NULL`, unique, foreign-key ή check constraint. Αν δεν υπάρχει, γράψε ποιος άλλος writer ή ποιο race μπορεί να την παρακάμψει. Μην προσθέσεις constraint πριν ελέγξεις τα υπάρχοντα rows και τη συμβατότητα των releases.

ΚΕΦΑΛΑΙΟ 5

Οι relations είναι προγράμματα SQL

Μια `ActiveRecord::Relation` είναι περιγραφή μελλοντικού query. Όσο παραμένει relation, μπορείς να τη συνθέσεις. Όταν φορτωθεί, έχεις rows και κόστος.



Σχήμα 5.1 · Η relation συντίθεται στη Ruby και εκτελείται μόνο όταν ζητηθούν τα rows.

Lazy μέχρι να μην είναι

Οι κλήσεις `where`, `joins`, `order` και `limit` επιστρέφουν νέα relation χωρίς να στείλουν αμέσως SQL. Η εκτέλεση έρχεται με `each`, `to_a`, `load`, `first`, `count`, `pluck` και άλλες terminal operations. Αυτό επιτρέπει να χτίσεις query σε layers χωρίς να μεταφέρεις ενδιάμεσα arrays.

RUBY

```ruby
scope = Event.published
scope = scope.where(city: "Athens")
scope = scope.where(starts_at: Time.current..)
scope = scope.order(:starts_at).limit(20)

Rails.logger.debug(scope.to_sql)
events = scope.load
```

Το `to_sql` δείχνει τη μορφή αλλά όχι πάντα τις τελικές bind values ή το plan. Το log και το `explain` είναι καλύτερα evidence για το query που εκτελέστηκε.

Scopes που συντίθενται

Ένα scope πρέπει να επιστρέφει relation και να εκφράζει μία σταθερή έννοια. Το scope `published` είναι καλό. Το scope `for_homepage_with_everything` συνήθως μπλέκει permission, presentation και eager loading μιας συγκεκριμένης οθόνης.

```ruby
class Event < ApplicationRecord
  scope :published, -> { where(status: :published) }
  scope :upcoming, -> { where(starts_at: Time.current..) }
  scope :for_account, ->(account) { where(account_id: account.id) }
end

events = Event.published.upcoming.for_account(Current.account)
```

Το `default_scope` κρύβει συνθήκες σε κάθε query, μαζί με admin reports και associations. Μπορεί να αλλάξει update ή delete scope με τρόπους που δεν φαίνονται στο call site. Προτίμησε named scopes και explicit base relations. Multi-tenant isolation χρειάζεται επίσης authorization και συχνά database strategy. Ένα default scope δεν αποτελεί μόνο του security boundary.

`merge`, `or` και joins

Το `merge` επαναχρησιμοποιεί scope του joined model χωρίς να αντιγράφεις table conditions ως strings.

```ruby
reservations = Reservation.joins(:event)
  .merge(Event.published.upcoming)
  .where(status: :confirmed)
```

Το `or` χρειάζεται structurally compatible relations. Αν οι δύο πλευρές έχουν διαφορετικά joins ή select lists, η σύνθεση δεν είναι καλά ορισμένη. Συχνά είναι καθαρότερο να ξεκινήσεις από κοινό base scope και να αλλάξεις μόνο το `where`.

```ruby
base = Event.where(account_id: Current.account.id)
visible = base.published.or(base.where(owner_id: Current.user.id))
```

Ένα join μπορεί να πολλαπλασιάσει parent rows. Το `distinct` διορθώνει το σύμπτωμα όταν πραγματικά θέλεις μοναδικούς parents, αλλά έχει κόστος και μπορεί να κρύψει λάθος cardinality. Πριν το προσθέσεις, γράψε ποια σχέση είναι one-to-one και ποια one-to-many.

Διάλεξε το σωστό αποτέλεσμα

Δεν χρειάζεται πάντα να δημιουργήσεις Active Record objects.

Ερώτηση	Συνήθης επιλογή	Τι επιστρέφει
Υπάρχει τουλάχιστον ένα row	`exists?`	boolean
Θέλω μία scalar τιμή	`pick(:column)`	μία τιμή
Θέλω συγκεκριμένα columns	`pluck(:id, :name)`	arrays τιμών
Θέλω aggregation	`count`, `sum`, `group`	αριθμούς ή hash
Θέλω behavior model	`select` και `load`	model instances

Το `length` σε unloaded relation φορτώνει όλα τα records. Το `count` συνήθως εκτελεί `COUNT(*)`. Το `size` χρησιμοποιεί loaded records όταν υπάρχουν και διαφορετικά μετρά στη βάση. Η διαφορά δεν είναι trivia όταν η relation έχει χιλιάδες rows.

Μην χρησιμοποιείς Ruby `select` επειδή ξέχασες ότι υπάρχει SQL `where`. Το πρώτο φορτώνει rows και φιλτράρει στη μνήμη. Το δεύτερο επιτρέπει στη βάση να χρησιμοποιήσει index και να επιστρέψει μόνο ό,τι χρειάζεσαι.

NULL και αρνητικές συνθήκες

Το `where.not(status: "cancelled")` δεν σημαίνει πάντα «όλα τα άλλα» όταν το column επιτρέπει NULL. Στο SQL το `NULL != 'cancelled'` δεν είναι true. Είναι unknown και συνήθως δεν περνά το `WHERE`. Αν θέλεις να συμπεριλάβεις NULL, γράψε το ρητά ή κάνε το column `NOT NULL` αν αυτό ταιριάζει στο domain.

```ruby
active = Reservation.where.not(status: :cancelled)
  .or(Reservation.where(status: nil))
```

Bound values, όχι interpolation

Τα hash conditions και placeholders κρατούν values έξω από τη SQL σύνταξη. String interpolation από params ανοίγει SQL injection και χαλά plan reuse.

```ruby
safe = Event.where("starts_at >= ?", Time.zone.parse("2026-10-01"))
unsafe_sql = "starts_at >= '#{Time.current}'"
```

Το δεύτερο assignment είναι syntactically valid Ruby αλλά δεν πρέπει να περάσει σε `where`. Όταν χρειάζεσαι dynamic column ή direction, επίλεξέ τα από δικό σου allowlist. Bind parameters προστατεύουν values, όχι SQL identifiers.

Arel στο στενό άκρο

Το Arel βοηθά σε expressions που το public query API δεν εκφράζει καθαρά. Πολλά μέρη του Arel δεν είναι σταθερό public API. Κράτα τη χρήση σε ένα μικρό query object, με tests στο παραγόμενο SQL ή στη συμπεριφορά. Μην αφήσεις Arel nodes να περνούν σε controllers και serializers.

> **ΠΡΟΣΟΧΗ**
>
> Η ωραία αλυσίδα Ruby δεν αποδεικνύει καλό query. Cardinality, bind values, indexes και database statistics αποφασίζουν το plan.

Batches και σταθερή πρόοδος

Για μεγάλο dataset χρησιμοποίησε `find_each` ή `in_batches` αντί για ένα `to_a`. Η batch iteration βασίζεται σε cursor order και χρειάζεται σταθερό, μοναδικό tie-breaker. Rows που αλλάζουν τα cursor columns κατά τη διάρκεια του run μπορούν να παραλειφθούν ή να εμφανιστούν ξανά. Για data migration κράτα checkpoint και σχεδίασε idempotent update.

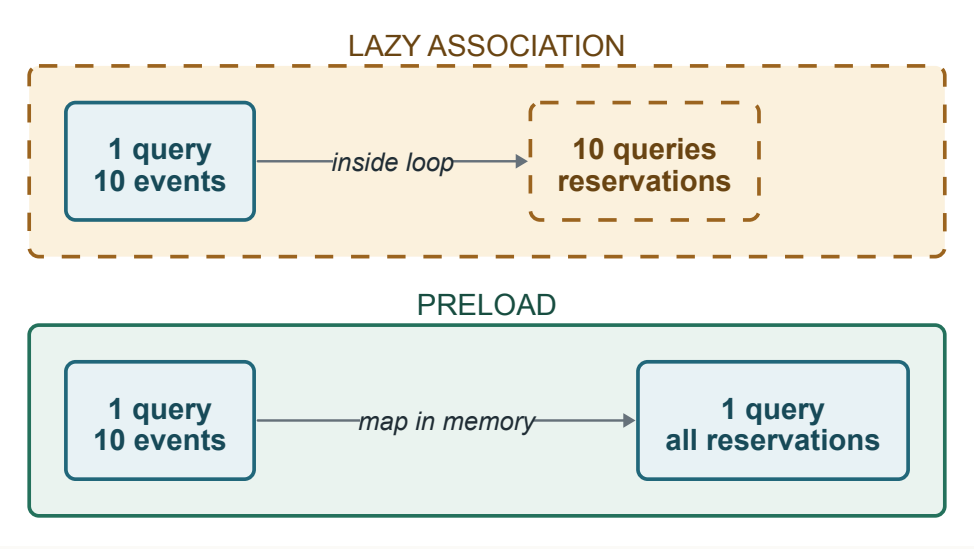
Η επίσημη λεπτομερής αναφορά είναι το Active Record Query Interface.

> **ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ**
>
> Βρες endpoint που επιστρέφει collection. Κατέγραψε πότε φορτώνεται η relation, αν μετατρέπεται νωρίς σε array και αν ο serializer προκαλεί νέα queries. Μετά σύγκρινε `to_sql`, query log και πλήθος instantiated records. Άλλαξε μόνο το σημείο με το καθαρότερο evidence.

ΚΕΦΑΛΑΙΟ 6

Associations χωρίς κρυφά queries

Το N+1 δεν είναι πρόβλημα «πολλών associations». Είναι πρόβλημα όταν η εκτέλεση ενός query κρύβεται μέσα σε iteration ή rendering.



Σχήμα 6.1 · Η lazy φόρτωση κάνει ένα child query ανά parent ενώ το preload τα συγκεντρώνει.

Από πού έρχεται το N+1

Ο controller φορτώνει δέκα events. Το view ζητά `event.reservations.size` για κάθε event. Αν η association δεν είναι loaded, εμφανίζεται ένα επιπλέον query ανά parent. Το template φαίνεται αθώο επειδή η database access είναι κρυμμένη πίσω από Ruby method.

RUBY

```ruby
class EventsController < ApplicationController
  def index
    @events = Event.published.order(:starts_at).limit(20)
  end
end
```

ERB

```erb
<% @events.each do |event| %>
  <p><%= event.name %>: <%= event.reservations.size %></p>
<% end %>
```

Η λύση δεν είναι να προσθέσεις `includes` παντού. Πρώτα αποφασίζεις ποια δεδομένα χρειάζεται η συγκεκριμένη representation και με ποια cardinality.

`preload`, `eager_load` και `includes`

Το `preload` εκτελεί χωριστό query για parents και χωριστό query ανά association τύπο. Είναι προβλέψιμο όταν δεν χρειάζεσαι filtering ή ordering από το joined table.

```ruby
events = Event.published
  .preload(:venue, reservations: :user)
  .order(:starts_at)
```

Το `eager_load` χρησιμοποιεί `LEFT OUTER JOIN`. Επιτρέπει συνθήκες πάνω στο joined table αλλά μπορεί να πολλαπλασιάσει SQL rows και να μεταφέρει πολύ περισσότερα bytes. Το Rails πρέπει μετά να ξανασυνθέσει μοναδικά parent objects.

```ruby
events = Event.eager_load(:venue)
  .where(venues: { city: "Athens" })
  .order(:starts_at)
```

Το `includes` επιλέγει στρατηγική ανάλογα με το query. Αυτό είναι βολικό αλλά σημαίνει ότι μια μεταγενέστερη `where` ή `references` μπορεί να αλλάξει ένα two-query load σε join. Όταν η στρατηγική έχει σημασία για performance ή semantics, προτίμησε το ρητό `preload` ή `eager_load`.

Ανάγκη	Επιλογή εκκίνησης	Κύριος κίνδυνος
Parents και children χωρίς child filter	`preload`	Μεγάλο child result set
Filter ή order σε joined table	`joins` ή `eager_load`	Row multiplication
Δεν ξέρεις ακόμα	`includes`	Αλλαγή στρατηγικής από σύνθεση

`joins` δεν σημαίνει loaded association

Το `joins(:venue)` χρησιμοποιεί το table για SQL filtering. Δεν γεμίζει από μόνο του το association cache. Αν μετά καλέσεις `event.venue`, μπορεί να κάνεις νέα queries. Συνδύασε `joins` με `preload` όταν χρειάζεσαι και filtering και loaded objects.

```ruby
events = Event.joins(:venue)
  .where(venues: { city: "Athens" })
  .preload(:venue)
```

Αυτό εκτελεί join query για να βρει events και δεύτερο query για να φορτώσει τα venues. Μπορεί να είναι καλύτερο από ένα μεγάλο eager-load join. Το σωστό εξαρτάται από cardinality και selectivity.

Strict loading ως feedback

Το `strict_loading` μετατρέπει απρόσμενο lazy load σε exception ή log, ανάλογα με configuration. Είναι ιδιαίτερα χρήσιμο σε serializers, views και critical collections όπου ένα νέο association call μπορεί να αλλάξει ένα endpoint από 3 queries σε 203.

```ruby
class Event < ApplicationRecord
  has_many :reservations, strict_loading: true
end
```

Μπορείς επίσης να εφαρμόσεις strict loading σε συγκεκριμένη relation. Αυτό είναι συχνά ασφαλέστερη αρχή από global enable σε παλιά εφαρμογή.

```ruby
events = Event.published.strict_loading.preload(:venue, :reservations)
```

Βάλε test που κάνει πραγματικό render ή serialization. Ένα model unit test που δεν περνά από το representation boundary δεν θα δει το κρυφό query.

`size`, counter cache και freshness

Το `association.size` χρησιμοποιεί loaded target αν υπάρχει. Διαφορετικά μπορεί να κάνει count query ή να χρησιμοποιήσει counter cache. Το `length` φορτώνει records. Το `count` ρωτά τη βάση και μπορεί να αγνοήσει unsaved objects που βρίσκονται ήδη στο association target.

Counter cache αξίζει όταν το count διαβάζεται πολύ συχνά και η μικρή write amplification είναι αποδεκτή. Χρειάζεται backfill και verification για παλιά rows. Είναι derived state. Αν χαλάσει από bulk write, πρέπει να μπορείς να το ξαναχτίσεις.

Pagination πριν από το preload

Περιόρισε parents πριν φορτώσεις associations. Αν κάνεις preload σε relation χωρίς bounded page, το N+1 μπορεί να εξαφανιστεί αλλά να αντικατασταθεί από τεράστιο allocation. Query count 2 δεν σημαίνει φθηνό endpoint όταν το δεύτερο query επιστρέφει 100.000 rows.

Για feeds που αλλάζουν συχνά, keyset pagination με σταθερό order και unique tie-breaker είναι πιο σταθερή από μεγάλο `OFFSET`. Για admin reports, offset pagination μπορεί να είναι αρκετή. Γράψε τις απαιτήσεις συνέπειας πριν διαλέξεις.

Async queries με πραγματικό budget

Το `load_async` μπορεί να επικαλύψει independent database waits. Δεν μειώνει το SQL work και καταναλώνει connection capacity από async executor ή pool. Σε μικρή βάση το επιπλέον concurrency μπορεί να χειροτερέψει latency. Μέτρα wall time, connection wait και database load μαζί.

RUBY

```ruby
events = Event.published.limit(20).load_async
counts = Reservation.confirmed.group(:event_id).count

events.each do |event|
  Rails.logger.debug([event.id, counts.fetch(event.id, 0)].inspect)
end
```

Στο παράδειγμα το aggregation δεν χρειάζεται να κατασκευάσει reservation objects. Το async load έχει νόημα μόνο αν το άλλο work γίνεται πραγματικά πριν χρειαστούν τα events.

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Βάλε query ownership στο boundary που ξέρει τη representation. Μετά έλεγξε και query count και row count. Το ένα χωρίς το άλλο είναι μισή μέτρηση.

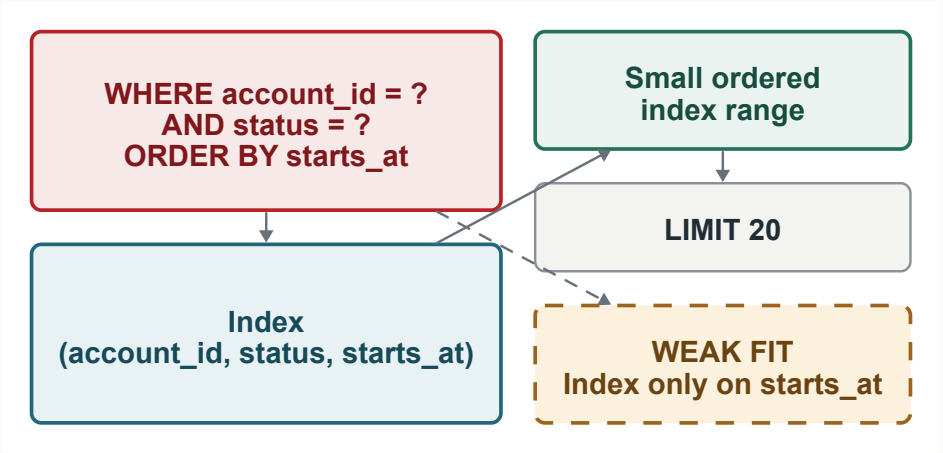
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Πάρε μία σελίδα λίστας και ενεργοποίησε strict loading στη base relation. Κατάγραψε κάθε association που λείπει. Για καθεμία διάλεξε preload, join με aggregation ή καθόλου model loading. Σύγκρινε SQL queries, returned rows και allocated model instances πριν κρατήσεις την αλλαγή.

ΚΕΦΑΛΑΙΟ 7

Indexes και execution plans

Δεν βάζουμε index σε column. Βάζουμε index για συγκεκριμένο query shape, cardinality και order.



Σχήμα 7.1 · Το composite index ακολουθεί equality predicates και μετά το order του query.

Ξεκίνα από το query

Ένα index είναι δεύτερη δομή δεδομένων που η database πρέπει να ενημερώνει σε κάθε σχετικό write. Έχει χώρο, write cost και maintenance. Το «βάλε index σε όλα τα foreign keys» είναι καλό baseline αλλά δεν αντικαθιστά τον σχεδιασμό των queries που έχουν πραγματικό latency ή throughput cost.

Έστω το endpoint που βρίσκει τις επόμενες confirmed κρατήσεις ενός account.

RUBY

```ruby
scope = Reservation.where(account_id: account.id, status: :confirmed)
  .where(starts_at: Time.current..)
  .order(:starts_at)
  .limit(20)
```

Σε PostgreSQL ένα πιθανό index είναι `(account_id, status, starts_at)`. Τα δύο equality predicates περιορίζουν πρώτα το range. Το `starts_at` υποστηρίζει range και order μέσα σε αυτό το prefix. Η σειρά δεν προκύπτει από το ποιο column «φαίνεται σημαντικό». Προκύπτει από τον τρόπο που ρωτά η εφαρμογή και από την κατανομή των τιμών.

```ruby
class IndexUpcomingReservations < ActiveRecord::Migration[8.1]
  disable_ddl_transaction!

  def change
    add_index :reservations,
      %i[account_id status starts_at],
      algorithm: :concurrently,
      name: "index_reservations_for_upcoming_list"
  end
end
```

Το `algorithm: :concurrently` είναι PostgreSQL-specific επιλογή του adapter και δεν επιτρέπεται μέσα στη συνηθισμένη migration transaction. Μειώνει το blocking σε writes αλλά χρειάζεται περισσότερο χρόνο και ειδικό failure handling. Αν το deploy system τρέχει migrations μόνο μία φορά, ένα failed concurrent build πρέπει να αφήνει σαφές operational signal.

Composite index και leftmost prefix

Ένα B-tree composite index είναι συνήθως πιο χρήσιμο όταν τα πρώτα columns ταιριάζουν στα equality predicates. Query μόνο με `starts_at` δεν αξιοποιεί αναγκαστικά αποτελεσματικά index που ξεκινά με `account_id`. Αυτό δεν σημαίνει ότι χρειάζεσαι κάθε πιθανή permutation. Σημαίνει ότι πρέπει να ξέρεις τα σημαντικά entry points.

Παρόμοια, sort direction και NULL ordering μπορούν να επηρεάσουν αν η database χρειάζεται ξεχωριστό sort. Ένα plan σε staging με 500 rows δεν αποδεικνύει τι θα γίνει στην production με 50 εκατομμύρια και διαφορετικά statistics.

Unique και partial indexes

Ένα unique index είναι concurrency primitive, όχι performance decoration. Αν κάθε account επιτρέπεται να έχει ένα ενεργό reservation ανά external key, η βάση πρέπει να απορρίψει τον δεύτερο writer.

```ruby
class AddReservationIdempotency < ActiveRecord::Migration[8.1]
  disable_ddl_transaction!

  def change
    add_index :reservations,
      %i[account_id idempotency_key],
      unique: true,
      algorithm: :concurrently,
      name: "index_reservations_on_account_and_idempotency"
  end
end
```

Partial index είναι χρήσιμο όταν το query και η invariant αφορούν σταθερό subset, όπως active rows. Είναι επίσης database-specific. Το predicate του query πρέπει να ταιριάζει αρκετά με το predicate του index ώστε ο planner να μπορεί να το χρησιμοποιήσει.

```ruby
class IndexPendingReservations < ActiveRecord::Migration[8.1]
  disable_ddl_transaction!

  def change
    add_index :reservations,
      %i[event_id created_at],
      where: "status = 0",
      algorithm: :concurrently,
      name: "index_pending_reservations_by_event"
  end
end
```

Το integer `0` πρέπει να συμφωνεί με την πραγματική enum storage. Αν αλλάξει το mapping, το schema object δεν αλλάζει αυτόματα. Σε κρίσιμη invariant προτίμησε σταθερές database values που δεν εξαρτώνται από τη θέση ενός enum array.

Διάβασε το plan

Το `relation.explain` ζητά από τη database execution plan για τα queries που χρειάζονται για τη relation. Με eager loading μπορεί να εκτελέσει περισσότερα από ένα queries για να πάρει τα plans τους. Το output θέλει context.

```ruby
plan = Reservation.where(account_id: 42, status: :confirmed)
  .order(:starts_at)
  .limit(20)
  .explain

Rails.logger.info(plan)
```

Κοίτα estimated rows, actual rows όταν χρησιμοποιείς ασφαλώς `EXPLAIN ANALYZE`, scan type, loops, sort και buffers. Μεγάλη διαφορά estimate και actual συχνά δείχνει stale statistics ή correlated columns. Το sequential scan δεν είναι πάντα κακό. Για μεγάλο ποσοστό table μπορεί να είναι η φθηνότερη επιλογή.

Το `EXPLAIN ANALYZE` εκτελεί το query. Σε `UPDATE`, `DELETE` ή ακριβό query μπορεί να αλλάξει δεδομένα ή να προκαλέσει πραγματικό φορτίο. Μην το τρέχεις τυφλά σε production console.

Κόστος model instantiation

Ακόμα και γρήγορο SQL μπορεί να γίνει αργό Ruby όταν επιστρέφει πολλές πλατιές γραμμές. Μέτρα `row_count` και `instantiation.active_record`. Χρησιμοποίησε `pluck`, aggregation ή περιορισμένο `select` όταν δεν χρειάζεσαι model behavior. Πρόσεξε ότι partially selected model μπορεί να σηκώσει `ActiveModel::MissingAttributeError` αν μεταγενέστερος κώδικας ζητήσει column που δεν φορτώθηκε.

Index review, όχι index συλλογή

Κάθε νέο index πρέπει να έχει owner query και λόγο. Κατέγραψε write rate, table size και αν καλύπτεται ήδη από άλλο composite index. Ένα index `(account_id, status)` μπορεί να γίνει περιττό μετά την προσθήκη `(account_id, status, starts_at)`, αλλά όχι πάντα. Unique semantics, sort και planner επιλογές διαφέρουν. Επιβεβαίωσε usage πριν αφαιρέσεις.

ΠΡΟΣΟΧΗ

Ένα benchmark με ζεστή cache και μία bind value δεν είναι γενική απόδειξη. Δοκίμασε αντιπροσωπευτικές τιμές, cold και warm συμπεριφορά, μαζί με write cost.

Η επίσημη αφετηρία είναι το τμήμα Running EXPLAIN και το Active Record Migrations.

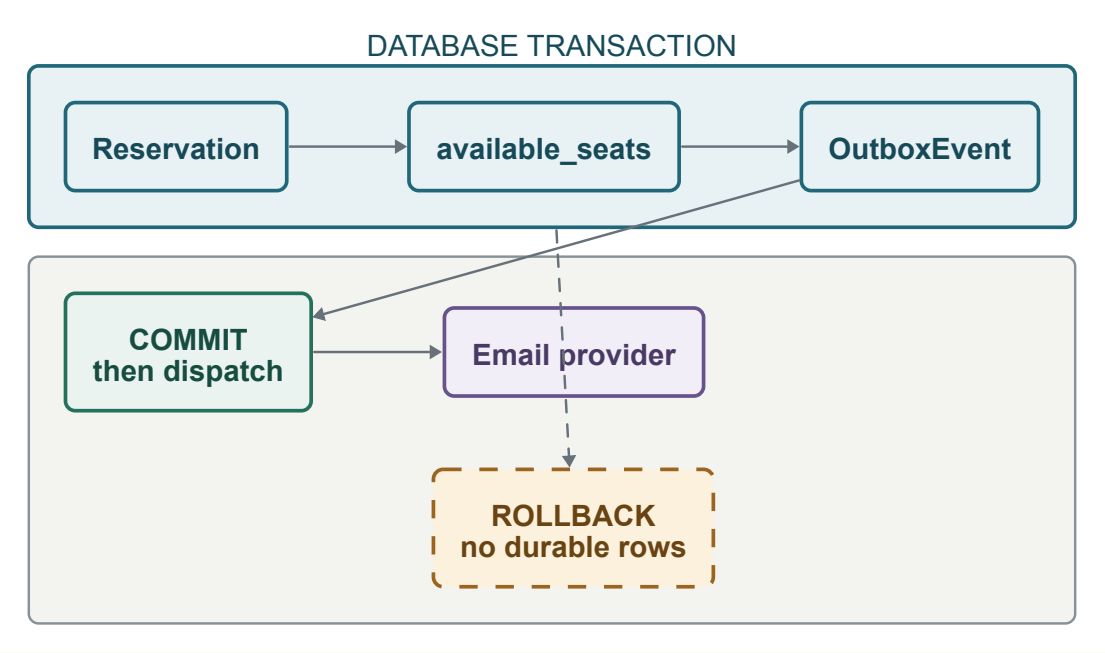
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Βρες τα τρία πιο αργά συχνά queries από πραγματικό monitoring. Για καθένα γράψε predicates, order, limit, estimated rows και actual returned rows. Σύνδεσε κάθε υπάρχον index με ένα query. Μην προσθέσεις index πριν εξηγήσεις ποιο μέρος του plan περιμένεις να αλλάξει.

ΚΕΦΑΛΑΙΟ 8

Transactions, callbacks και commit

H transaction προστατεύει writes σε μία database connection. Δεν αναιρεί email, HTTP calls ή mutable Ruby objects.



Σχήμα 8.1 · Τα database writes γίνονται atomic και τα εξωτερικά effects ξεκινούν μετά το commit.

Το πραγματικό όριο

`ActiveRecord::Base.transaction` δανείζεται μία connection και ορίζει ένα atomic boundary για SQL που περνά από αυτή. Τα models μέσα στο block δεν χρειάζεται να έχουν την ίδια class. Πρέπει όμως να χρησιμοποιούν την ίδια database connection. Δύο databases δεν αποκτούν distributed transaction επειδή έβαλες δύο model classes στο ίδιο Ruby block.

```ruby
module Reservations
  class Confirm
    def call(reservation:)
      Reservation.transaction do
        event = Event.lock.find(reservation.event_id)
        raise SoldOut if event.available_seats < reservation.seats

        event.update!(available_seats: event.available_seats -
reservation.seats)
        reservation.update!(status: :confirmed, confirmed_at: Time.current)
        OutboxEvent.create!(topic: "reservation.confirmed", record:
reservation)
      end
    end
  end
end
```

Αν ένα write αποτύχει, τα προηγούμενα SQL writes στο block κάνουν rollback. Τα in-memory objects δεν γυρίζουν αυτόματα στις προηγούμενες attribute values. Αν συνεχίσεις να χρησιμοποιείς ένα instance μετά από rollback, κάνε reload ή πέτα το object.

Exceptions και rollback

Κάθε κανονικό exception προκαλεί rollback και ξανασηκώνεται. Το `ActiveRecord::Rollback` κάνει rollback αλλά καταπίνεται από το transaction block. Είναι κατάλληλο όταν το rollback είναι ελεγχόμενο αποτέλεσμα που δεν πρέπει να γίνει exception προς τα έξω. Χρησιμοποίησέ το προσεκτικά γιατί μπορεί να κρύψει control flow.

Μην κάνεις rescue `ActiveRecord::StatementInvalid` μέσα στην ίδια transaction και μετά συνεχίζεις. Στον PostgreSQL ένα database error μπορεί να αφήσει την transaction aborted μέχρι το τέλος της. Κάνε retry ολόκληρο το transaction boundary όταν η συγκεκριμένη κατηγορία error είναι πράγματι retryable.

```ruby
def create_once!(attributes)
  Reservation.create!(attributes)
rescue ActiveRecord::RecordNotUnique
  Reservation.find_by!(
    account_id: attributes.fetch(:account_id),
    idempotency_key: attributes.fetch(:idempotency_key)
  )
end
```

Εδώ το rescue βρίσκεται έξω από την εσωτερική transaction του `create!`. Η unique constraint ορίζει τον νικητή. Το follow-up lookup επιστρέφει το durable row. Σε υψηλό replica lag το lookup πρέπει να πάει στον writer.

Nested transactions και savepoints

Μια nested `transaction` χωρίς επιλογές συνήθως συμμετέχει στην outer transaction. Ένα `ActiveRecord::Rollback` στο inner block δεν φτάνει απαραίτητα στο outer block. Για ανεξάρτητο

rollback χρησιμοποίησε `requires_new: true`, το οποίο οι περισσότερες databases υλοποιούν με savepoint.

```ruby
Reservation.transaction do
  reservation.update!(status: :confirmed)

  AuditEntry.transaction(requires_new: true) do
    AuditEntry.create!(action: "confirmed", auditable: reservation)
  end
end
```

Αν το audit write αποτύχει και το exception γίνει rescue έξω από το inner block, μπορείς τεχνικά να κρατήσεις την outer transaction. Ρώτησε όμως αν αυτό είναι σωστό domain αποτέλεσμα. Savepoint mechanics δεν αποφασίζουν business semantics.

`after_save` δεν είναι `after_commit`

Το `after_save` τρέχει πριν γίνει οριστικό το outer commit. Ένα job ή provider που ξεκινά εκεί μπορεί να δει row που τελικά θα γίνει rollback. Το `after_commit` τρέχει αφού η database έχει δεχτεί την αλλαγή.

```ruby
class Reservation < ApplicationRecord
  after_update_commit :enqueue_confirmation, if: :saved_change_to_confirmed_at?

  private

  def enqueue_confirmation
    SendReservationConfirmationJob.perform_later(id)
  end
end
```

Αυτό είναι ασφαλέστερο χρονικά αλλά εξακολουθεί να κρύβει application workflow στο model lifecycle. Αν μόνο ένα συγκεκριμένο use case πρέπει να στείλει την ειδοποίηση, γράψε outbox row μέσα στην operation αντί για γενικό callback.

Επίσης το «μετά το commit» δεν εγγυάται ότι το process θα ζήσει αρκετά για να κάνει enqueue. Υπάρχει μικρό crash window ανάμεσα στο database commit και στο external enqueue. Το transactional outbox του κεφαλαίου 15 υπάρχει ακριβώς για αυτό το κενό.

Κράτα την transaction μικρή

Μέσα στην transaction κάνε μόνο ό,τι χρειάζεται την κοινή atomicity. Μην καλείς HTTP provider, μην περιμένεις file upload και μην κάνεις μεγάλο rendering. Όσο κρατάς transaction ανοιχτή, κρατάς connection και πιθανώς row locks. Το latency τρίτου συστήματος μετατρέπεται σε contention της βάσης σου.

```ruby
def confirm_and_notify(reservation:)
  reservation_id = Reservation.transaction do
    confirm_in_database!(reservation)
    reservation.id
  end

  SendReservationConfirmationJob.perform_later(reservation_id)
end
```

Το παράδειγμα έχει crash window. Είναι αποδεκτό μόνο αν η ειδοποίηση είναι best-effort ή υπάρχει reconciliation. Για εγγυημένη τελική παράδοση χρειάζεται durable intent στην ίδια transaction.

Isolation και locks

Η transaction δεν σημαίνει ότι δύο callers εκτελούνται σειριακά. Το isolation level της database καθορίζει τι βλέπει κάθε transaction. Row lock προστατεύει συγκεκριμένα rows, unique constraint προστατεύει συγκεκριμένη μοναδικότητα και serializable isolation μπορεί να απαιτήσει retry. Θα συνθέσουμε αυτά τα εργαλεία στο κεφάλαιο 11.

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Γράψε πρώτα ποια rows πρέπει να αλλάξουν μαζί. Μετά βάλε transaction γύρω από αυτά τα writes. Κάθε effect που δεν μπορεί να κάνει rollback μένει έξω ή εκφράζεται ως durable outbox intent.

Η ακριβής συμπεριφορά nested transactions και callbacks καταγράφεται στο Active Record Transactions API.

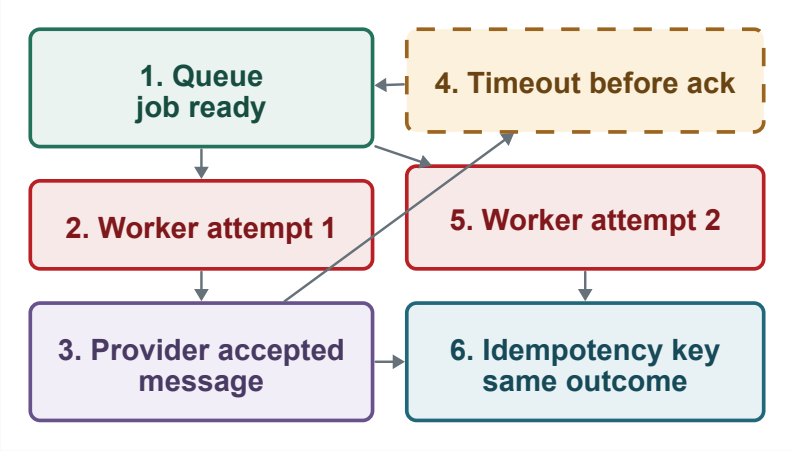
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Βρες transaction που περιέχει network call, rendering ή enqueue. Μέτρα πόσο κρατά connection και locks. Γράψε ποιο crash window δημιουργείται αν μεταφέρεις το effect μετά το commit. Αν το effect είναι κρίσιμο, σχεδίασε outbox row πριν πειράξεις τον κώδικα.

ΚΕΦΑΛΑΙΟ 9

Jobs, retries και idempotency

Ένα background job δεν είναι «κάν' το αργότερα». Είναι μήνυμα που μπορεί να καθυστερήσει, να εκτελεστεί δύο φορές ή να μείνει στη μέση.



Σχήμα 9.1 · Ένα timeout μετά το external effect οδηγεί σε δεύτερη προσπάθεια και απαιτεί idempotency.

Το delivery contract

Τα περισσότερα πρακτικά queue systems προσφέρουν at-least-once execution ή κάτι που πρέπει να αντιμετωπίζεται έτσι. Ο worker παίρνει job, εκτελεί work και μετά καταγράφει completion. Αν το process πεθάνει αφού ο provider δέχτηκε το request αλλά πριν γίνει acknowledge, η queue δικαίως θα ξαναδώσει το job.

Δεν υπάρχει γενικό «exactly once» ανάμεσα στη database σου και σε ανεξάρτητο external provider. Υπάρχει idempotent effect, deduplication key, durable state και reconciliation.

RUBY

```ruby
class SendReservationConfirmationJob < ApplicationJob
  queue_as :mailers

  retry_on Net::ReadTimeout, wait: 10.seconds, attempts: 5
  discard_on ActiveJob::DeserializationError

  def perform(reservation_id)
    reservation = Reservation.confirmed.find(reservation_id)
    ReservationsMailer.confirmation(reservation).deliver_now
  end
end
```

Το job δέχεται scalar id. Active Job μπορεί να serialize Active Record records μέσω GlobalID, αλλά όταν εκτελεστεί το job θα κάνει lookup της τότε κατάστασης. Το row μπορεί να έχει διαγραφεί ή να έχει αλλάξει. Το explicit id κάνει αυτή την επιλογή πιο ορατή, όχι διαφορετική από μόνο του.

Το argument είναι reference, όχι snapshot

Αν το email πρέπει να αντανακλά την τρέχουσα reservation, φόρτωσε το record όταν τρέχει το job. Αν πρέπει να στείλει ακριβώς την τιμή που εγκρίθηκε, γράψε immutable snapshot ή versioned payload σε durable table. Μην περάσεις τεράστιο hash στην queue επειδή μοιάζει εύκολο. Payload schema αλλάζει μεταξύ releases και μπορεί να περιέχει προσωπικά δεδομένα που δεν πρέπει να μένουν στα job records.

Ένα καλό argument set είναι μικρό, versionable και αρκετό για να ξαναβρεθεί η δουλειά. Ένα καλό job ελέγχει ξανά preconditions επειδή η κατάσταση μπορεί να άλλαξε όσο περίμενε.

```ruby
class ExpireReservationJob < ApplicationJob
  def perform(reservation_id, expected_lock_version)
    reservation = Reservation.find_by(id: reservation_id)
    return unless reservation&.pending?
    return unless reservation.lock_version == expected_lock_version

    reservation.cancel!(reason: :expired)
  end
end
```

Το early return είναι επιτυχής no-op όταν το intended state έχει ήδη αλλάξει. Δεν χρειάζεται retry για δουλειά που δεν ισχύει πλέον.

Retry μόνο για μεταβατικό failure

Το retry έχει νόημα για timeout, προσωρινό provider outage, deadlock ή άλλο failure που μπορεί να αλλάξει χωρίς αλλαγή input. Δεν λύνει invalid email, missing permission ή παραβίαση invariant. Αν κάνεις retry μόνιμο error, γεμίζεις την queue και κρύβεις το πραγματικό signal.

Χρησιμοποίησε συγκεκριμένες exception classes. Ένα `retry_on StandardError` κάνει retry και bugs του κώδικα. Το backoff πρέπει να προστατεύει το dependency από retry storm. Πρόσθεσε jitter όταν πολλές ίδιες δουλειές μπορεί να αποτύχουν μαζί, αν ο adapter ή ο δικός σου μηχανισμός το υποστηρίζει.

Μετά την τελευταία προσπάθεια, το job δεν πρέπει απλώς να εξαφανιστεί. Χρειάζεται error report, failed-job visibility και owner. Για business-critical work χρειάζεται επίσης reconciliation query που βρίσκει durable intents χωρίς ολοκληρωμένο effect.

Idempotency στο σωστό boundary

Το flag `email_sent_at` βοηθά αλλά μόνο αν η ανάγνωση, η απόφαση και η ενημέρωση συντονίζονται. Δύο workers μπορούν να διαβάσουν NULL ταυτόχρονα. Για external provider που δέχεται idempotency key, στείλε σταθερό key ανά logical effect και αποθήκευσέ το.

```ruby
class DeliverReceiptJob < ApplicationJob
  def perform(delivery_id)
    delivery = Delivery.find(delivery_id)
    return if delivery.delivered?

    response = ReceiptProvider.deliver(
      payload: delivery.payload,
      idempotency_key: "delivery-#{delivery.id}"
    )

    delivery.update!(delivered_at: Time.current, provider_id: response.id)
  end
end
```

Υπάρχει ακόμα crash window μετά τον provider και πριν το local update. Το σταθερό provider idempotency key κάνει τη δεύτερη κλήση να επιστρέψει το ίδιο logical result. Αν ο provider δεν υποστηρίζει deduplication, χρειάζεσαι δικό σου ledger ή αποδέχεσαι και χειρίζεσαι duplicates.

Enqueue και database commit

Job που μπαίνει σε queue πριν γίνει commit μπορεί να τρέξει αμέσως και να μη βρει το row. Το Rails και adapters έχουν ρυθμίσεις για enqueue after transaction commit. Στο Rails 8.1 η συμπεριφορά δεν πρέπει να θεωρείται ίδια για κάθε adapter και topology. Το Solid Queue είναι από default σε ξεχωριστή database, άρα δεν μοιράζεται αυτόματα atomic commit με τα application tables.

```ruby
class ApplicationJob < ActiveJob::Base
  self.enqueue_after_transaction_commit = true
end
```

Αυτό αποφεύγει early execution. Δεν κλείνει το crash window μετά το commit και πριν από το enqueue όταν οι δύο writes δεν είναι στην ίδια transaction. Για critical work η durable outbox παραμένει ισχυρότερη λύση.

Queues και capacity isolation

Χώρισε queues όταν χρειάζεσαι διαφορετικό capacity ή latency objective, όχι μόνο επειδή τα jobs έχουν διαφορετικά ονόματα. Ένα αργό import δεν πρέπει να κρατά πίσω password reset. Queue order και numeric priority έχουν adapter-specific semantics. Στο Solid Queue η σειρά των queues έχει προτεραιότητα και το numeric priority εφαρμόζεται μέσα στην queue.

Κατέγραψε queue latency, execution duration, retries και oldest job age. Το average execution time δεν αποκαλύπτει ότι ένα critical job περίμενε 20 λεπτά.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Για κάθε job γράψε τέσσερα πράγματα. Durable intent, idempotency strategy, retryable exceptions και reconciliation path. Αν λείπει ένα, υπάρχει άγνωστο failure mode.

Η επίσημη βάση είναι το Active Job Basics. Τα continuations και τα Solid Queue concurrency controls έρχονται στο κεφάλαιο 14.

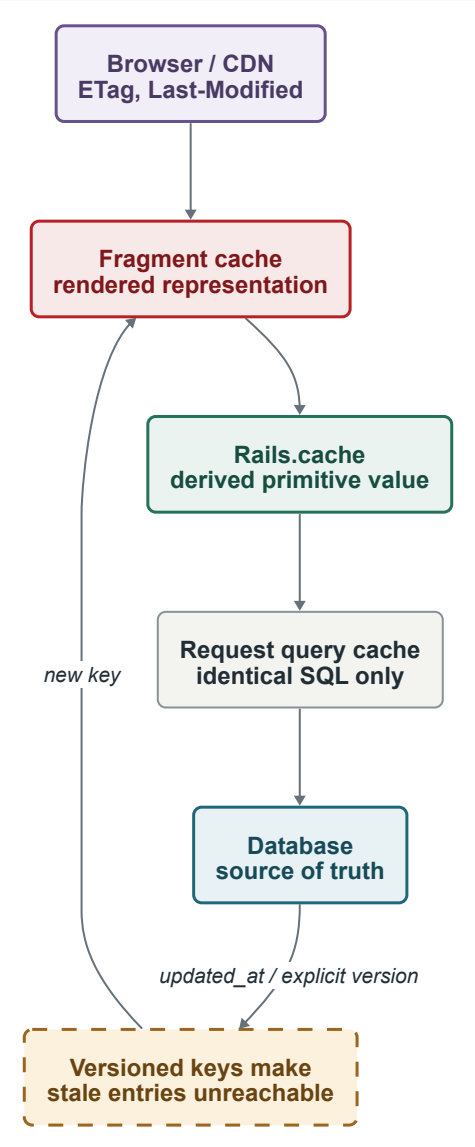
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Διάλεξε job που καλεί εξωτερικό σύστημα. Βάλε το process να «πεθαίνει» νοητά σε κάθε γραμμή. Για κάθε σημείο γράψε τι θα συμβεί στο retry. Αν βρίσκεις διπλό effect ή χαμένο intent, σχεδίασε stable idempotency key και reconciliation query.

ΚΕΦΑΛΑΙΟ 10

Cache και HTTP freshness

Η cache είναι παράγωγο δεδομένο με περιορισμένη διάρκεια. Αν γίνει δεύτερη πηγή αλήθειας, κάθε invalidation γίνεται business bug.



Σχήμα 10.1 · Τα cache layers βρίσκονται πάνω από τη database και έχουν διαφορετικό scope.

Τέσσερα διαφορετικά πράγματα

Το Rails χρησιμοποιεί τη λέξη cache για μηχανισμούς με διαφορετικό lifecycle.

Layer	Τι κρατά	Συνήθης διάρκεια
HTTP freshness	Απόκριση στον browser ή CDN	Πολλά requests
Fragment cache	Rendered representation	Μέχρι να αλλάξει key/version
Low-level cache	Derived primitive data	Ρητό expiry ή version
SQL query cache	Result ίδιου SQL	Ένα request ή execution unit

Το SQL query cache δεν είναι γενική application cache. Αποφεύγει δεύτερο ίδιο query μέσα στο ίδιο lifecycle και δημιουργεί ξανά model instances από το cached result. Δεν λύνει ακριβό query σε επόμενα requests.

Versioned keys αντί για delete choreography

Το πιο ασφαλές invalidation είναι συχνά να κάνει το παλιό key unreachable. Το `cache_key_with_version` περιλαμβάνει identity και version από το record. Όταν αλλάξει το `updated_at`, το νέο read ζητά νέο key.

RUBY

```ruby
class Event < ApplicationRecord
  def availability_summary
    Rails.cache.fetch(
      [cache_key_with_version, "availability-summary"],
      expires_in: 30.minutes
    ) do
      {
        confirmed: reservations.confirmed.sum(:seats),
        remaining: available_seats
      }
    end
  end
end
```

Μην cache-άρεις Active Record instances. Μπορεί να γίνουν stale, να αναφέρονται σε class object πριν από reload ή να έχουν association state που δεν ισχύει. Κράτα ids, strings, numbers και μικρά versioned hashes. Μετά φόρτωσε current records όταν χρειάζεται behavior.

Αν η τιμή εξαρτάται από reservations αλλά το key μόνο από το event, η αλλαγή reservation πρέπει να αλλάζει event version. Το `touch: true` μπορεί να κάνει αυτό το dependency explicit αλλά αυξάνει writes και contention στο parent row. Άλλη επιλογή είναι domain-specific cache version ή aggregation table.

Fragment και collection caching

Το fragment caching βάζει rendered HTML κάτω από key που περιλαμβάνει template digest και record version. Το collection rendering με `cached: true` μπορεί να κάνει multi-read αντί για ένα cache call ανά item.

```erb
<%= render partial: "events/event",
           collection: @events,
           cached: ->(event) { [I18n.locale, event] } %>
```

Η locale είναι μέρος της representation και άρα του key. Το ίδιο ισχύει για currency, feature variant ή permission level όταν αλλάζουν πραγματικά το output. Μην βάλεις ολόκληρο user object στο key χωρίς να ξέρεις πόσες εκδοχές θα δημιουργήσει.

Russian doll caching επαναχρησιμοποιεί inner fragments όταν αλλάζει outer container. Χρειάζεται σωστό dependency graph. Αν child update δεν αλλάζει το parent version, το outer fragment μπορεί να μείνει stale. Το template digest βλέπει template dependencies, όχι κάθε helper ή external setting που επηρεάζει το αποτέλεσμα.

HTTP conditional requests

Η φθηνότερη απόκριση μπορεί να είναι 304 χωρίς body. Τα `fresh_when` και `stale?` συνδέουν ETag ή Last-Modified με το representation. Αυτό αφήνει τον browser ή CDN να επαναχρησιμοποιήσει απόκριση χωρίς να μεταφέρεις ξανά HTML ή JSON.

```ruby
class EventsController < ApplicationController
  def show
    event = Event.find(params.require(:id))

    if stale?(event, public: true)
      render json: EventSerializer.new(event).as_json
    end
  end
end
```

Το `public: true` είναι σοβαρή δήλωση. Αν η representation περιέχει user-specific ή private δεδομένα, shared cache μπορεί να τα δείξει σε λάθος χρήστη. Το cache key και τα HTTP headers πρέπει να περιλαμβάνουν κάθε διάσταση που αλλάζει permission ή output.

Stampede και race condition TTL

Όταν λήξει δημοφιλές key, πολλά requests μπορεί να υπολογίσουν μαζί την ίδια ακριβή τιμή. Το `race_condition_ttl` επιτρέπει σε έναν caller να ανανεώσει ενώ οι υπόλοιποι παίρνουν προσωρινά την παλιά τιμή, ανάλογα με το store.

```ruby
value = Rails.cache.fetch(
  "homepage/event-stats/v3",
  expires_in: 5.minutes,
  race_condition_ttl: 10.seconds
) do
  EventStats.calculate
end
```

Αυτό ταιριάζει σε derived value που επιτρέπεται να είναι λίγα δευτερόλεπτα stale. Δεν ταιριάζει σε διαθέσιμες θέσεις τη στιγμή επιβεβαίωσης. Εκεί η database transaction αποφασίζει. Η cache μπορεί να δείξει ένδειξη, όχι να κρατήσει inventory invariant.

Solid Cache και topology

Από Rails 8 το Solid Cache είναι default επιλογή για νέα apps και αποθηκεύει entries σε database-backed store. Μεγάλη χωρητικότητα και απλούστερη υποδομή μπορούν να είναι καλό tradeoff. Παραμένει όμως workload πάνω σε database. Κράτα συνήθως ξεχωριστή cache database ή τουλάχιστον ξεκάθαρο connection budget ώστε cache miss storm να μην ανταγωνίζεται critical writes.

Αν το Solid Cache χρησιμοποιεί το ίδιο Active Record pool και connection, cache write μέσα σε application transaction μπορεί να συμμετέχει σε εκείνο το transaction. Μη βασίζεις semantics σε αυτό χωρίς explicit topology. Μια μελλοντική μεταφορά σε ξεχωριστή database αλλάζει τη συμπεριφορά.

Cache failures

Αποφάσισε αν cache outage κάνει fail-open ή fail-closed. Για product list συνήθως επιστρέφεις στη βάση και προστατεύεις την απότομη αύξηση με rate ή concurrency limit. Για authorization data, stale ή missing cache δεν πρέπει να παρακάμπτει permission check. Η source of truth αποφασίζει.

Μέτρα hit ratio ανά namespace, fetch latency, entry size και backend errors. Ένα υψηλό global hit ratio μπορεί να κρύβει ότι το ακριβότερο key χάνει πάντα.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Για κάθε cache γράψε source of truth, key dimensions, staleness budget, invalidation mechanism και fallback. Αν δεν μπορείς να τη διαγράψεις και να ξαναχτιστεί, δεν είναι απλή cache.

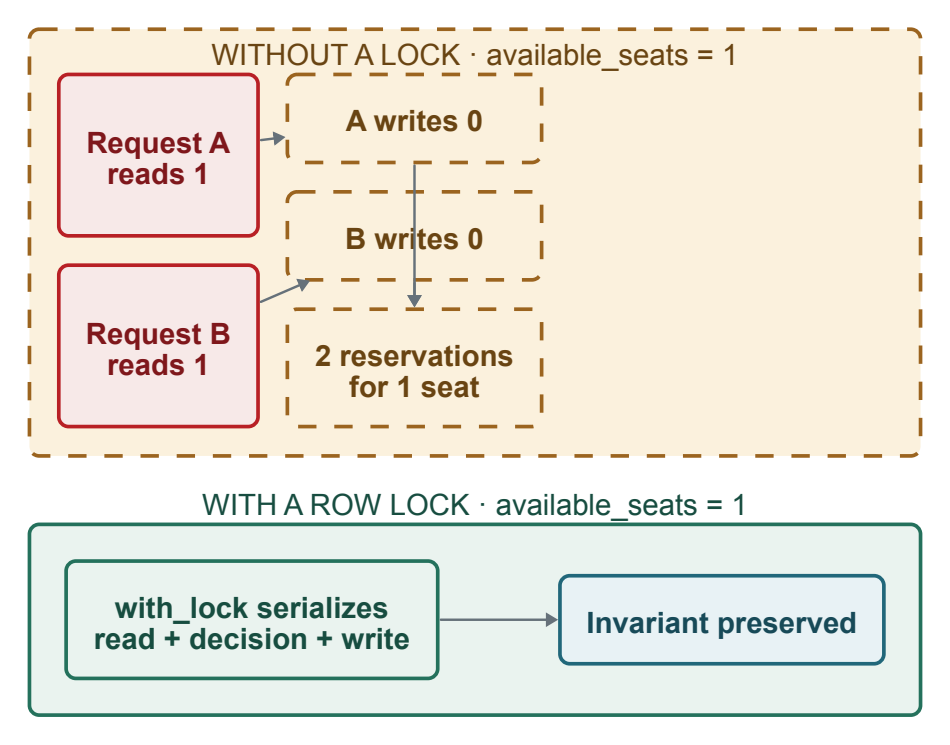
Η επίσημη αναφορά είναι το Caching with Rails.

> **ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ**
>
> Διάλεξε ένα `Rails.cache.fetch`. Γράψε όλα τα δεδομένα που επηρεάζουν το output και σύγκρινέ τα με το key. Μετά φαντάσου cache miss για κάθε request επί ένα λεπτό. Αν η database δεν αντέχει το fallback, βάλε capacity guard πριν αυξήσεις το TTL.

ΚΕΦΑΛΑΙΟ 11

Concurrency, locks και conflicts

Αν η απόφαση βασίζεται σε τιμή που μόλις διάβασες, πρέπει να εξηγήσεις τι συμβαίνει όταν άλλος writer την αλλάξει πριν γράψεις.



Σχήμα 11.1 · Δύο requests μπορούν να περάσουν το ίδιο check. Ένα row lock σειριοποιεί την κρίσιμη απόφαση.

To check-then-act race

Δύο requests διαβάζουν `available_seats = 1`. Και τα δύο περνούν το check. Και τα δύο δημιουργούν reservation. Το τελευταίο update γράφει πάλι μηδέν και η βάση καταλήγει με δύο κρατήσεις για μία θέση. Κανένα request δεν έκανε «λάθος» μόνο του. Η σύνθεση ήταν λάθος.

```ruby
def unsafe_confirm!(event:, reservation:)
  raise SoldOut if event.available_seats < reservation.seats

  event.update!(available_seats: event.available_seats - reservation.seats)
  reservation.update!(status: :confirmed)
end
```

Transaction χωρίς lock δεν λύνει από μόνη της αυτό το race στο συνηθισμένο isolation level. Χρειάζεται atomic predicate, constraint ή serialization του read-decision-write.

Pessimistic locking

Το `with_lock` ανοίγει transaction, κάνει reload το record με row lock και μετά δίνει το block. Στον PostgreSQL το default αντιστοιχεί σε `SELECT ... FOR UPDATE`. Άλλοι adapters έχουν δικές τους λεπτομέρειες.

```ruby
def confirm!(event:, reservation:)
  event.with_lock do
    raise SoldOut if event.available_seats < reservation.seats

    event.update!(available_seats: event.available_seats - reservation.seats)
    reservation.update!(status: :confirmed, confirmed_at: Time.current)
  end
end
```

Τώρα μόνο ένας writer αποφασίζει πάνω στην locked έκδοση του event κάθε στιγμή. Ο δεύτερος περιμένει, κάνει reload και βλέπει μηδέν. Η κρίσιμη ενότητα πρέπει να μένει μικρή. External HTTP call μέσα στο `with_lock` μετατρέπει provider latency σε lock wait για όλους.

Όταν κλειδώνεις πολλά rows, κράτα σταθερή σειρά. Δύο transactions που κλειδώνουν A μετά B και B μετά A μπορούν να κάνουν deadlock. Η database θα ακυρώσει έναν writer. Ο application κώδικας χρειάζεται bounded retry ολόκληρης της transaction, όχι retry από τη μέση.

```ruby
def lock_events(ids)
  Event.where(id: ids).order(:id).lock.load
end
```

Database-specific clauses όπως `FOR UPDATE NOWAIT` και `SKIP LOCKED` είναι χρήσιμες για interactive failure ή work claiming. Δεν είναι portable Rails semantics. Κράτα τις σε query object και δοκίμασέ τις με τον production adapter.

Atomic conditional update

Μερικές invariants μπορούν να εκφραστούν σε ένα SQL update. Δεν υπάρχει κενό ανάμεσα σε read και write.

```ruby
updated = Event.where(id: event.id)
  .where("available_seats >= ?", reservation.seats)
  .update_all(["available_seats = available_seats - ?", reservation.seats])

raise SoldOut unless updated == 1
```

Το `update_all` παρακάμπτει callbacks, validations και συνήθη timestamp updates. Αυτό είναι γνωστό tradeoff, όχι κρυφή λεπτομέρεια. Αν χρειάζεσαι και reservation write, βάλε το conditional update και τη δημιουργία reservation στην ίδια transaction. Αν το δεύτερο αποτύχει, το seat update κάνει rollback.

Unique constraint ως arbiter

Για invariant μοναδικότητας, unique index είναι συνήθως απλούστερος από lock. Δύο inserts μπορούν να αγωνιστούν και η database θα δεχτεί έναν. Το model uniqueness validation υπάρχει για feedback αλλά δεν είναι arbiter.

```ruby
def create_idempotently!(attributes)
  Reservation.create!(attributes)
rescue ActiveRecord::RecordNotUnique
  Reservation.find_by!(
    account_id: attributes.fetch(:account_id),
    idempotency_key: attributes.fetch(:idempotency_key)
  )
end
```

Το lookup πρέπει να χρησιμοποιεί ακριβώς τα columns του logical key. Αν η unique violation μπορεί να προέρχεται από άλλο index, έλεγξε constraint name ή κράτα το rescue σε μικρότερο boundary.

Optimistic locking

Με integer column `lock_version`, το Active Record βάζει την αναμενόμενη version στο update. Αν άλλος writer έχει ήδη αλλάξει το row, σηκώνεται `ActiveRecord::StaleObjectError` αντί να γίνει silent overwrite.

```ruby
def update_profile!(user:, attributes:)
  user.update!(attributes)
rescue ActiveRecord::StaleObjectError
  raise ProfileConflict, "profile changed while you were editing"
end
```

Το optimistic locking ταιριάζει όταν conflicts είναι σπάνια και θέλεις να τα δείξεις ή να τα συγχωνεύσεις. Για web form χρειάζεται να επιστρέψει το `lock_version` μαζί με τα fields και να το στείλει πίσω. Διαφορετικά το action φορτώνει πάντα φρέσκο object και δεν συγκρίνει την έκδοση που είδε ο χρήστης.

Δεν αντικαθιστά unique constraint και δεν προστατεύει invariant που απλώνεται σε πολλά rows χωρίς πρόσθετο σχεδιασμό.

Isolation και retry

Υψηλότερο isolation μπορεί να μετατρέψει ανωμαλίες σε serialization failures. Αυτό δεν σημαίνει ότι το exception είναι bug. Είναι απαίτηση να ξανατρέξει η ολόκληρη logical transaction πάνω σε νέα snapshot. Το retry πρέπει να είναι bounded, με μικρό jitter και μόνο για αναγνωρισμένες adapter exceptions.

Το block πρέπει επίσης να είναι safe να ξανατρέξει. Αν έχει ήδη καλέσει provider, δεν μπορείς απλώς να επαναλάβεις. Για αυτό τα external effects μένουν μετά το commit ή περνούν από outbox.

Δοκιμή με πραγματική concurrency

Ένα sequential model test δεν αποδεικνύει race safety. Χρειάζεσαι δύο ξεχωριστές connections, συντονισμένο barrier και production database adapter. SQLite έχει διαφορετικό locking και δεν αποδεικνύει PostgreSQL row-level behavior.

Το test πρέπει να ελέγχει τελική invariant, όχι ποιο thread κέρδισε. Για μία θέση, ένα reservation επιβεβαιώνεται, ένα αποτυγχάνει και το counter δεν γίνεται αρνητικό. Τρέξε το test πολλές φορές αλλά μην θεωρήσεις ότι το repeat μόνο του αποδεικνύει απουσία race. Η βάση της απόδειξης είναι το constraint ή το atomic mechanism.

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Διάλεξε το μικρότερο primitive που εκφράζει την invariant. Unique index για μοναδικότητα, conditional update για atomic counter, row lock για σύνθετη απόφαση πάνω σε συγκεκριμένα rows και optimistic lock για visible edit conflict.

Οι Rails APIs τεκμηριώνονται στα Pessimistic Locking και Optimistic Locking.

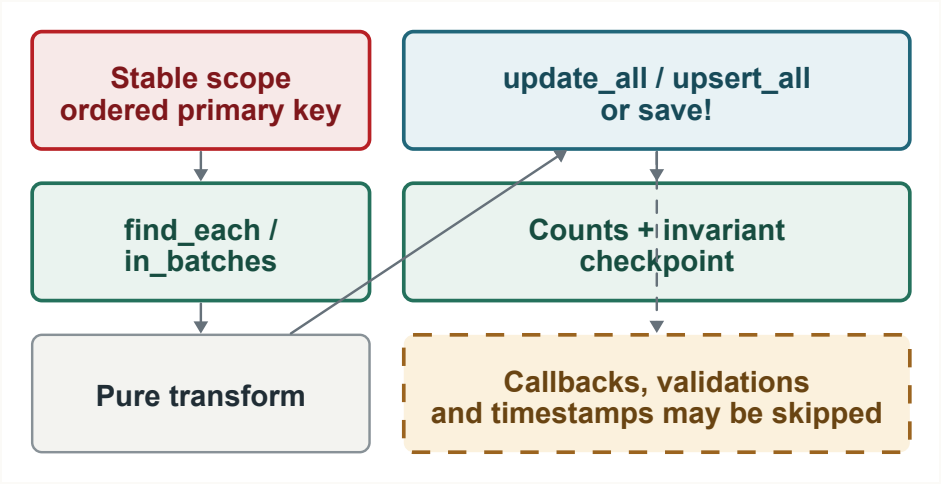
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Βρες ένα check-then-write flow. Σχεδίασε δύο requests που σταματούν αμέσως μετά το check. Γράψε την ανεπιθύμητη τελική κατάσταση. Μετά επίλεξε constraint, conditional update ή lock και εξήγησε με μία πρόταση γιατί κλείνει ακριβώς το κενό.

ΚΕΦΑΛΑΙΟ 12

Bulk writes και αλλαγές δεδομένων

To bulk API κερδίζει επειδή παρακάμπτει object lifecycle. Πρέπει να ξέρεις ακριβώς ποια συμπεριφορά άφησες έξω.



Σχήμα 12.1 · Ένα ασφαλές bulk run έχει stable scope, batches, idempotent transform και checkpoints.

Τρία διαφορετικά μονοπάτια

Το `record.update!` φορτώνει model, κάνει type casting, validations, callbacks, dirty tracking και timestamps. Το `update_all` στέλνει ένα SQL update στη relation και παρακάμπτει validations και callbacks. Τα `insert_all` και `upsert_all` γράφουν πολλά rows χωρίς να δημιουργήσουν model instances.

Δεν υπάρχει γενικά καλύτερο API. Υπάρχει semantic κόστος και operational κόστος.

API	Model instances	Validations και callbacks	Καλή χρήση
`update!`	Ναι	Ναι	Domain transition ανά record
`update_all`	Όχι	Όχι	Ενιαίο deterministic update
`insert_all`	Όχι	Όχι	Νέα rows με ήδη ελεγμένο payload
`upsert_all`	Όχι	Όχι	Idempotent sync με unique key

Αν callback στέλνει email, το bulk write δεν θα το καλέσει. Αυτό μπορεί να είναι ακριβώς το σωστό επειδή τα external effects δεν πρέπει να κρύβονται σε data migration. Μπορεί επίσης να είναι production bug αν η derived state δεν ξαναχτιστεί. Κάνε inventory πριν τρέξεις.

Bounded batches

Μην φορτώσεις ολόκληρο table. Το `in_batches` δίνει relation ανά batch και το `find_each` δίνει objects. Το cursor πρέπει να έχει σταθερή σειρά και μοναδικό tie-breaker. Primary key order είναι η απλούστερη βάση όταν τα rows δεν αλλάζουν id.

```ruby
Reservation.where(normalized_email: nil).in_batches(of: 1_000) do |batch|
  batch.each do |reservation|
    normalized = reservation.email.to_s.strip.downcase
    reservation.update_columns(
      normalized_email: normalized,
      updated_at: Time.current
    )
  end
end
```

Το `update_columns` παρακάμπτει validations και callbacks αλλά κάνει type cast μέσω Active Record. Εδώ το timestamp γράφεται ρητά. Το loop εξακολουθεί να κάνει ένα update ανά row. Αν η database μπορεί να εκφράσει σωστά το transform, ένα `update_all` είναι πολύ φθηνότερο. Αν η normalization πρέπει να ταιριάζει ακριβώς με Ruby semantics, κράτα το per-row transform και μέτρα.

Idempotency και restart

Ένα backfill που κρατά ώρες θα διακοπεί κάποια στιγμή. Σχεδίασέ το ώστε να ξανατρέχει. Scope μόνο rows που λείπουν, deterministic transform και checkpoint με processed count. Μην βασίζεσαι σε offset πάνω σε dataset που αλλάζει. Μετά από κάθε update, το row πρέπει να βγαίνει από το pending scope.

```ruby
scope = Reservation.where(normalized_email: nil).where.not(email: nil)

scope.find_each(batch_size: 500) do |reservation|
  reservation.update_columns(
    normalized_email: reservation.email.strip.downcase,
    updated_at: Time.current
  )
end
```

Αν το process πεθάνει, τα completed rows δεν επιλέγονται ξανά. Αν ένα row αλλάξει email ανάμεσα σε read και write, χρειάζεται version predicate ή maintenance rule. Το «τρέχει ξανά» δεν λύνει concurrent edits από μόνο του.

`upsert_all` και conflict target

Το upsert χρειάζεται unique index που ορίζει identity. Το `unique_by` είναι adapter-dependent ως προς δυνατότητες και schema cache. Η ενημέρωση δεν τρέχει model callbacks. Fields όπως `updated_at` και lock versions θέλουν συνειδητή πολιτική.

```ruby
rows = external_events.map do |item|
  {
    source: "partner",
    external_id: item.fetch(:id),
    name: item.fetch(:name),
    updated_at: Time.current,
    created_at: Time.current
  }
end

Event.upsert_all(rows, unique_by: %i[source external_id])
```

Αν το provider στείλει παλιότερο event μετά από νεότερο, ένα απλό upsert μπορεί να γυρίσει τα δεδομένα πίσω. Βάλε source version ή timestamp predicate στη sync λογική. Idempotent key δεν σημαίνει monotonic freshness.

Data change έξω από schema migration

Μικρό, deterministic update μπορεί να ζήσει σε migration αλλά γίνεται δύσκολο να παρατηρηθεί, να σταματήσει και να ξανατρέξει. Για μεγάλο dataset προτίμησε ξεχωριστό script ή job με versioned code, metrics και bounded batches. Το schema migration κάνει expand. Το backfill εκτελείται ανεξάρτητα. Μετά έρχεται verification και contract.

Μην φορτώνεις current application model σε παλιά migration. Το model αλλάζει με τον χρόνο και μπορεί να έχει callbacks ή columns που δεν υπήρχαν όταν γράφτηκε η migration. Αν χρειάζεται model wrapper, όρισε μικρή local class με explicit table και μόνο τα fields που χρειάζεται, ή χρησιμοποίησε SQL.

Locking και πολλοί workers

Για parallel backfill μπορείς να κάνεις claim rows με status ή με database-specific `SKIP LOCKED`. Το claim πρέπει να είναι durable ώστε crash να μην αφήνει row για πάντα κλειδωμένο λογικά. Lease timestamp και retry policy είναι συχνά καθαρότερα από boolean `processing`.

Μην αυξήσεις workers μέχρι να δεις database saturation, replication lag και lock waits. Ένα backfill είναι background μόνο ως προς τον χρήστη. Για τη βάση είναι κανονικό write workload που παράγει WAL, vacuum work και index updates.

Verification πριν από constraint

Μην χρησιμοποιείς μόνο processed counter. Μέτρα pending rows από source-of-truth predicate, invalid values, duplicates και referential gaps. Πάρε samples από αρχή, μέση και τέλος του key range. Για critical transform, υπολόγισε independent checksum ή ξαναϋπολόγισε expected value σε sample.

RUBY

```ruby
missing = Reservation.where(normalized_email: nil).where.not(email: nil).count
invalid = Reservation.where("normalized_email <> lower(trim(email))").count

raise "backfill incomplete" unless missing.zero? && invalid.zero?
```

Το SQL function behavior μπορεί να διαφέρει από Ruby για Unicode. Το παράδειγμα είναι verification μόνο αν αυτή είναι η συμφωνημένη normalization semantics.

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Ένα production backfill είναι μικρό σύστημα. Έχει scope, cursor, idempotency, throttle, progress, stop condition, verification και rollback plan.

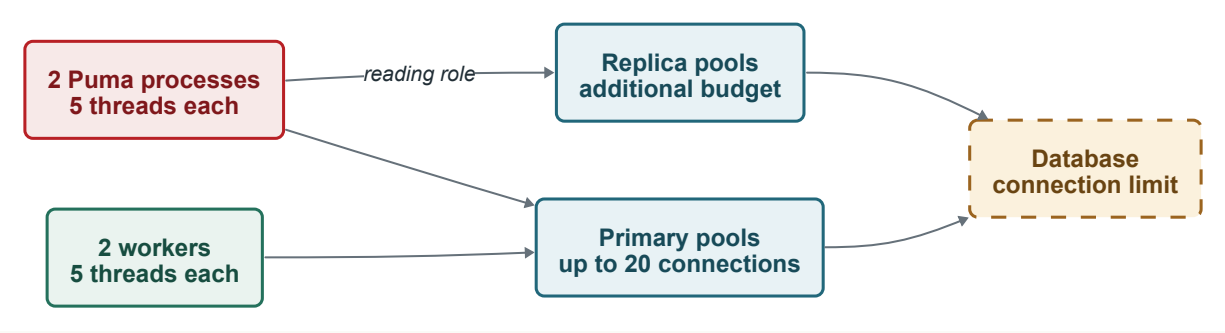
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Βρες το μεγαλύτερο table που χρειάζεται derived column. Γράψε dry-run query με pending count και sample output. Υπολόγισε writes ανά δευτερόλεπτο που αντέχει η βάση από πραγματικό monitoring. Σχεδίασε batch και pause χωρίς να τρέξεις το backfill.

ΚΕΦΑΛΑΙΟ 13

Connection pools, replicas και shards

To connection pool δεν είναι ένα global νούμερο. Πολλαπλασιάζεται ανά process, role, shard και database configuration.



Σχήμα 13.1 · Web και job concurrency δημιουργούν ξεχωριστά pools προς primary και replica.

Κάνε τον πολλαπλασιασμό

Δύο Puma processes με πέντε threads μπορούν να ζητήσουν μέχρι δέκα web connections προς το primary. Δύο job processes με πέντε threads προσθέτουν άλλα δέκα. Αν κάθε process έχει και reading pool προς replica, δημιουργούνται επιπλέον connections. Console, migration process, scheduler και health tooling μπαίνουν στο ίδιο συνολικό budget.

Το pool size πρέπει να καλύπτει τη μέγιστη πραγματική concurrency που χρειάζεται database access μέσα σε κάθε process. Πολύ μικρό pool δημιουργεί checkout wait. Πολύ μεγάλο pool δεν δημιουργεί capacity στη database. Απλώς επιτρέπει περισσότερο ταυτόχρονο work και μπορεί να την κορέσει.

YAML

```yaml
production:
  primary:
    url: <%= ENV.fetch("DATABASE_URL") %>
    pool: <%= ENV.fetch("RAILS_MAX_THREADS", 5) %>
  primary_replica:
    url: <%= ENV.fetch("REPLICA_DATABASE_URL") %>
    pool: <%= ENV.fetch("RAILS_MAX_THREADS", 5) %>
    replica: true
```

Το configuration δείχνει σχήμα, όχι προτεινόμενο production νούμερο. Το σωστό pool προκύπτει από process topology, thread count και database limit. Μην εμφανίζεις URLs ή credentials σε diagnostics.

Roles ως explicit choice

Το Rails υποστηρίζει writing και reading roles. Ένα abstract record μπορεί να συνδέσει το primary με replica.

```ruby
class ApplicationRecord < ActiveRecord::Base
  primary_abstract_class

  connects_to database: {
    writing: :primary,
    reading: :primary_replica
  }
end
```

Με `connected_to(role: :reading)` ο κώδικας δηλώνει ότι δέχεται read semantics της replica. Το Rails μπορεί να αποτρέψει προφανή writes σε readonly role, αλλά δεν μετατρέπει τη replica σε φρέσκια snapshot του primary.

```ruby
report = ApplicationRecord.connected_to(role: :reading) do
  Event.published.group(:city).count
end
```

Το report μπορεί να ανεχτεί λίγα δευτερόλεπτα lag. Η σελίδα που εμφανίζεται αμέσως μετά από επιβεβαίωση κράτησης ίσως όχι.

Read-your-writes

Μετά από write στο primary, η replica μπορεί να μην έχει εφαρμόσει ακόμα την αλλαγή. Redirect σε GET που διαβάζει replica μπορεί να δείξει «η κράτηση δεν βρέθηκε». Αυτό δεν είναι Rails caching bug. Είναι consistency property της topology.

Το Database Selector middleware μπορεί να κρατά πρόσφατο writer session στο primary για configurable delay. Είναι pragmatic heuristic, όχι απόδειξη ότι η replica έχει προλάβει μετά το delay. Για κρίσιμο read χρησιμοποίησε writer role ή version/LSN-aware μηχανισμό της υποδομής.

```ruby
reservation = ApplicationRecord.connected_to(role: :writing) do
  Reservation.find(params.require(:id))
end
```

Η explicit writer read είναι σωστή όταν η freshness requirement το απαιτεί. Αν τη βάλεις παντού, η replica δεν αποφορτίζει τίποτα. Κατάταξε endpoints σε fresh, session-consistent και stale-tolerant.

Transactions δεν περνούν connections

Μια transaction ανήκει σε μία database connection. Write στο primary και write σε ξεχωριστή queue database δεν είναι atomic. Nested Ruby blocks δεν δημιουργούν distributed transaction. Το ίδιο ισχύει για δύο shards.

Αυτό επηρεάζει Solid Queue, outbox και audit stores. Αν το intent πρέπει να δεθεί atomically με business row, γράψ' το στην ίδια database transaction και μετέφερέ το αργότερα. Αν επιλέξεις ξεχωριστή database, αποδέξου explicit reconciliation.

Horizontal sharding

Το Rails μπορεί να συνδέσει shards και να αλλάζει `shard:` μέσα σε block. Το shard key πρέπει να είναι διαθέσιμο πριν από το query και να ταξιδεύει σε jobs, URLs ή current context με ασφαλή τρόπο.

```ruby
class AccountRecord < ApplicationRecord
  self.abstract_class = true

  connects_to shards: {
    europe: { writing: :europe_primary, reading: :europe_replica },
    america: { writing: :america_primary, reading: :america_replica }
  }
end
```

Cross-shard joins δεν υπάρχουν ως απλό SQL operation. Aggregations, uniqueness και transactions γίνονται πιο δύσκολα. Το sharding είναι λύση όταν έχεις μετρημένο database bottleneck ή isolation requirement, όχι επειδή το app έχει accounts.

Το `disable_joins: true` σε through associations μπορεί να μετατρέψει cross-database join σε πολλαπλά queries. Αυτό μπορεί να μεταφέρει μεγάλο ID set και να κάνει ordering ή limit στη μνήμη. Η δυνατότητα δεν εξαφανίζει το κόστος της κατανομής.

Pool pressure και long work

Ένα request κρατά connection μόνο όταν το Active Record work το χρειάζεται, αλλά μεγάλη transaction την κρατά συνεχώς. Manual threads, async queries και job concurrency μπορούν να ζητήσουν παραπάνω. Παρακολούθησε checkout wait, timeouts, busy/idle connections και query duration.

Αν τα threads είναι περισσότερα από το pool, αυτό δεν είναι πάντα λάθος. Μερικά requests κάνουν CPU ή network work χωρίς connection. Πρέπει όμως να ξέρεις τι συμβαίνει στο peak. Ένα cache outage μπορεί να κάνει ξαφνικά κάθε thread να ζητήσει database connection.

> **ΠΡΟΣΟΧΗ**
>
> Μην αυξάνεις pool επειδή βλέπεις checkout timeout πριν ελέγξεις αργές transactions και database saturation. Μπορεί να μεταφέρεις την ουρά από το app στη database και να κάνεις το failure μεγαλύτερο.

Η επίσημη αναφορά είναι το Multiple Databases with Active Record.

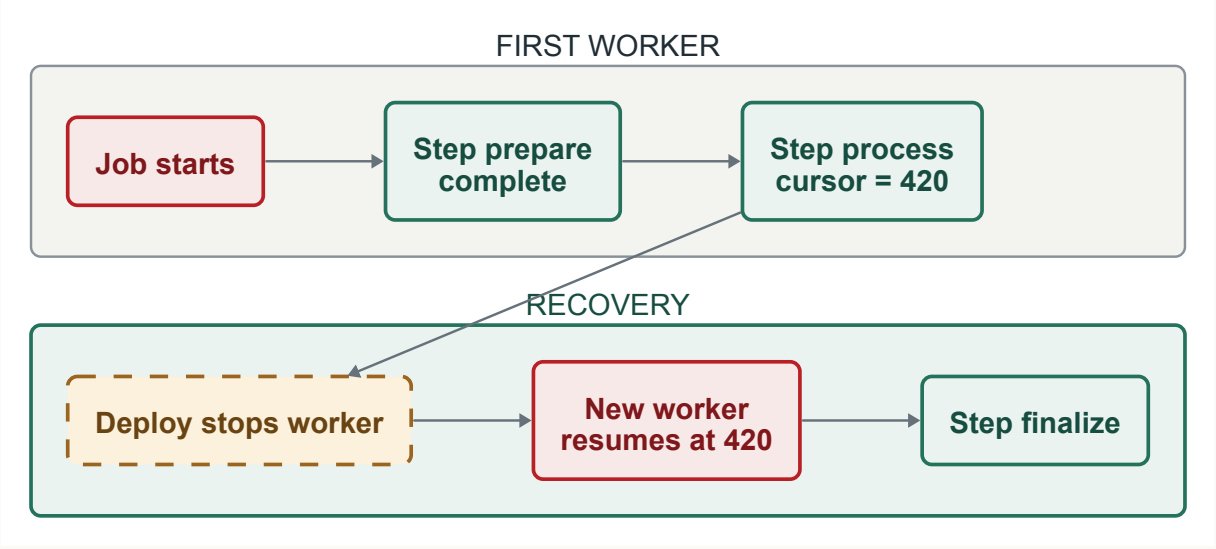
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Σχεδίασε όλους τους process types της production και γράψε processes επί threads επί pools ανά role. Σύγκρινε το άνω όριο με το database connection limit. Μετά βρες ποια endpoints απαιτούν read-your-writes και ποια μπορούν να διαβάσουν replica.

ΚΕΦΑΛΑΙΟ 14

Solid Queue και job continuations

Η queue topology είναι μέρος της συνέπειας του συστήματος. Τα continuations λύνουν interruption, όχι διπλά external effects.



Σχήμα 14.1 · Ένα continuation αποθηκεύει ολοκληρωμένα steps και cursor ώστε νέο worker να συνεχίσει.

Τι προσθέτει το Solid Queue

Το Solid Queue είναι το default Active Job backend για νέα Rails apps από το Rails 8. Είναι database-backed και υποστηρίζει delayed jobs, recurring tasks, numeric priority, queue order, process και thread workers, μαζί με concurrency controls. Δεν χρειάζεται Redis μόνο και μόνο για να υπάρξει durable queue.

Αυτό δεν σημαίνει ότι η queue είναι δωρεάν. Polling, inserts, ready-job indexes, heartbeats και cleanup είναι database workload. Το default production setup τη βάζει σε ξεχωριστή database ώστε το queue traffic να μην μοιράζεται την ίδια transaction και τα ίδια tables με το application data.

YAML

```
production:
  primary:
    url: <%= ENV.fetch("DATABASE_URL") %>
  queue:
    url: <%= ENV.fetch("QUEUE_DATABASE_URL") %>
    migrations_paths: db/queue_migrate
```

Με ξεχωριστή database, business write και job enqueue δεν είναι atomic. Το `enqueue_after_transaction_commit` αποφεύγει job πριν από commit αλλά δεν εξαλείφει process crash μετά το commit και πριν από το queue insert. Για critical intent χρειάζεται outbox στην application database.

Queue order πριν από numeric priority

Στο Solid Queue οι workers εξετάζουν queues με τη διαμορφωμένη σειρά. Το numeric priority ταξινομεί jobs μέσα στην ίδια queue. Αν βάλεις `critical` πριν από `default`, ένα default job με «καλύτερο» αριθμό δεν περνά μπροστά από critical queue work.

Χώρισε queues για capacity isolation και διαφορετικό service objective. Μην φτιάχνεις queue ανά job class. Ένα σύστημα με δεκάδες queues γίνεται δύσκολο να παρατηρηθεί και η σειρά μπορεί να προκαλέσει starvation στις τελευταίες.

YAML

```yaml
production:
  workers:
    - queues: "critical,mailers"
      threads: 3
      processes: 2
    - queues: "default,maintenance"
      threads: 5
      processes: 1
```

Τα νούμερα είναι παράδειγμα topology, όχι έτοιμο capacity plan. Κάθε thread που τρέχει Active Record work χρειάζεται πιθανή connection προς application και queue databases.

Concurrency controls

Το `limits_concurrency` περιορίζει πόσα jobs συγκεκριμένου logical key τρέχουν μαζί. Είναι χρήσιμο όταν ο provider επιτρέπει ένα request ανά account ή όταν δύο jobs πάνω στην ίδια reservation θα συγκρούονταν άσκοπα.

RUBY

```ruby
class SyncReservationJob < ApplicationJob
  limits_concurrency to: 1,
    key: ->(reservation_id) { reservation_id },
    duration: 5.minutes,
    group: "reservation-sync"

  def perform(reservation_id)
    Reservations::Sync.call(reservation: Reservation.find(reservation_id))
  end
end
```

Το limit είναι scheduling control. Δεν είναι database invariant. Αν άλλος code path γράφει το ίδιο row ή αν αλλάξεις adapter, η ασφάλεια πρέπει να παραμένει σε constraint, lock ή idempotent operation. Το `duration` πρέπει να καλύπτει αναμενόμενο work χωρίς να μπλοκάρει για πάντα μετά από lost worker.

Continuations στο Rails 8.1

Το Rails 8.1 πρόσθεσε `ActiveJob::Continuable`. Ένα μεγάλο job χωρίζεται σε named steps. Όταν worker σταματήσει ελεγχόμενα, νέο execution συνεχίζει από το τελευταίο ολοκληρωμένο step. Step με cursor μπορεί να συνεχίσει μέσα σε μεγάλη collection.

```ruby
class ProcessImportJob < ApplicationJob
  include ActiveJob::Continuable

  def perform(import_id)
    @import = Import.find(import_id)

    step :prepare do
      @import.prepare!
    end

    step :process do |step|
      @import.rows.order(:id).find_each(start: step.cursor) do |row|
        Imports::ApplyRow.call(row: row)
        step.advance! from: row.id
      end
    end

    step :finalize
  end

  private

  def finalize
    @import.finalize!
  end
end
```

Το setup πριν από τα `step` calls τρέχει ξανά όταν γίνεται resume, άρα πρέπει να είναι φθηνό και safe. Το cursor χρειάζεται monotonic, stable order. Αν records προστίθενται ή αλλάζουν ids δεν έχεις πρόβλημα με integer primary key, αλλά cursor σε mutable timestamp θέλει tie-breaker και σαφή snapshot semantics.

Τι ακριβώς έχει αποθηκευτεί

Η completion ενός step σημαίνει ότι το continuation state καταγράφηκε. Δεν σημαίνει ότι κάθε external side effect είναι exactly-once. Process μπορεί να διακοπεί μετά το effect και πριν το `advance!`. Στο resume το ίδιο row θα ξαναεκτελεστεί. Το `ApplyRow` πρέπει να είναι idempotent ή να χρησιμοποιεί stable key.

Ακόμα και τα steps χωρίς cursor πρέπει να αντέχουν επανάληψη γύρω από crash boundaries. Το continuation μειώνει χαμένο work. Δεν αντικαθιστά retry design.

```ruby
module Imports
  class ApplyRow
    def self.call(row:)
      ImportedRecord.upsert(
        { source_id: row.source_id, payload: row.payload },
        unique_by: :source_id
      )
    end
  end
end
```

Το upsert είναι idempotent μόνο ως προς το unique key και το chosen update. External calls μέσα στο operation χρειάζονται δικό τους deduplication.

Deploy και shutdown

Ένας worker πρέπει να πάρει TERM, να σταματήσει να ζητά νέα jobs και να δώσει χρόνο στα in-flight jobs. Μετά το grace period μπορεί να τερματιστεί. Τα μικρά jobs ολοκληρώνονται. Τα continuable jobs αφήνουν progress. Jobs που αγνοούν signals ή μπλοκάρουν σε unbounded I/O παραμένουν πρόβλημα.

Μέτρα execution duration percentiles και σύγκρινέ τα με shutdown timeout. Αν τα μισά jobs ξεπερνούν το grace period, δεν αρκεί να αυξήσεις timeout για πάντα. Χώρισέ τα σε resumable units ή βάλε provider timeouts.

Recurring tasks και overlap

Recurring schedule δεν εγγυάται ότι η προηγούμενη εκτέλεση τελείωσε. Ένα job κάθε λεπτό που κρατά δύο λεπτά μπορεί να συσσωρεύεται. Συνδύασε stable logical key, concurrency limit και observable skipped or coalesced behavior. Για maintenance task, συχνά «μία ενεργή εκτέλεση, η επόμενη θα ξαναδεί όλο το pending scope» είναι καθαρό contract.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Το continuation απαντά «από πού συνεχίζω;». Η idempotency απαντά «τι γίνεται αν το ίδιο unit ξανατρέξει;». Χρειάζεσαι και τις δύο απαντήσεις.

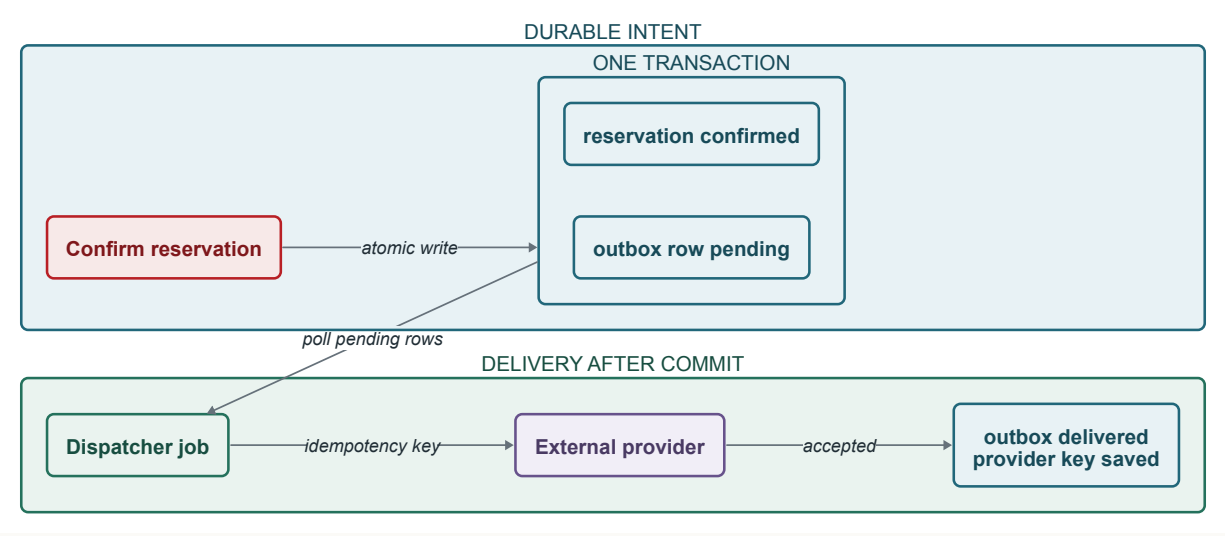
Η επίσημη αναφορά βρίσκεται στο Active Job Basics και στα Rails 8.1 release notes.

> **ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ**
>
> Βρες το πιο αργό job. Χώρισέ το σε setup, επαναλαμβανόμενο unit και finalize. Διάλεξε stable cursor. Μετά βάλε νοητό crash πριν και μετά από κάθε effect. Αν το ίδιο unit δεν είναι safe να ξανατρέξει, λύσε αυτό πριν προσθέσεις continuation.

ΚΕΦΑΛΑΙΟ 15

Εξωτερικά effects και transactional outbox

Η database μπορεί να αποθηκεύσει atomically την πρόθεση για ένα external effect. Δεν μπορεί να κάνει rollback τον provider.



Σχήμα 15.1 · Η reservation και το outbox intent γράφονται μαζί. Dispatcher παραδίδει αργότερα με stable key.

Το κενό ανάμεσα σε δύο συστήματα

Θέλεις να επιβεβαιώσεις reservation και να ενημερώσεις partner API. Αν καλέσεις τον partner πριν το database commit, μπορεί να δεχτεί κάτι που μετά κάνει rollback. Αν τον καλέσεις μετά το commit, το process μπορεί να πεθάνει πριν στείλει. Αν κάνεις enqueue σε ξεχωριστή queue database, υπάρχει το ίδιο κενό.

Το transactional outbox γράφει business change και durable intent στην ίδια database transaction. Ένας dispatcher διαβάζει pending intents και καλεί το external system. Το pattern δεν είναι αυτόματη Rails δυνατότητα. Είναι schema, workflow και operational ownership που χτίζεις πάνω σε Active Record.

```ruby
class CreateOutboxEvents < ActiveRecord::Migration[8.1]
  def change
    create_table :outbox_events do |table|
      table.string :topic, null: false
      table.string :deduplication_key, null: false
      table.json :payload, null: false
      table.string :status, null: false, default: "pending"
      table.datetime :available_at, null: false
      table.datetime :delivered_at
      table.datetime :lease_expires_at
      table.integer :attempts, null: false, default: 0
      table.timestamps
    end

    add_index :outbox_events, :deduplication_key, unique: true
    add_index :outbox_events, %i[status available_at]
  end
end
```

Το JSON type και index strategy διαφέρουν ανά database. Το payload πρέπει να είναι μικρό και versioned. Μην αντιγράψεις ολόκληρο model ή ευαίσθητα δεδομένα που ο consumer δεν χρειάζεται.

Atomic intent

Η operation δημιουργεί outbox row μέσα στην ίδια transaction με την state transition.

```ruby
module Reservations
  class Confirm
    def call(reservation:)
      Reservation.transaction do
        reservation.confirm!
        OutboxEvent.create!(
          topic: "reservation.confirmed.v1",
          deduplication_key: "reservation-confirmed-#{reservation.id}",
          payload: { reservation_id: reservation.id },
          available_at: Time.current
        )
      end
    end
  end
end
```

Αν το commit πετύχει, υπάρχουν και τα δύο rows. Αν αποτύχει, δεν υπάρχει ούτε confirmed reservation ούτε intent. Το unique key εμποδίζει δύο logical intents για την ίδια transition. Αν το domain επιτρέπει confirm, cancel και ξανά confirm, το key χρειάζεται transition id ή version, όχι μόνο reservation id.

Claim χωρίς μεγάλο lock

Ο dispatcher δεν πρέπει να κρατά database transaction ανοιχτή όσο περιμένει HTTP provider. Μπορεί να κάνει μικρό claim transaction, να καλέσει έξω από το lock και μετά να σημειώσει αποτέλεσμα. Lease επιτρέπει reclaim μετά από crash.

```ruby
class OutboxEvent < ApplicationRecord
  scope :ready, -> {
    where(status: :pending, available_at: ..Time.current)
      .or(where(status: :processing, lease_expires_at: ..Time.current))
  }

  def self.claim_batch(limit: 100)
    transaction do
      rows = ready.order(:id).limit(limit).lock("FOR UPDATE SKIP LOCKED").to_a
      rows.each do |row|
        row.update!(
          status: :processing,
          lease_expires_at: 2.minutes.from_now,
          attempts: row.attempts + 1
        )
      end
      rows
    end
  end
end
```

Το `SKIP LOCKED` είναι PostgreSQL-specific σε αυτό το παράδειγμα. Το returned Ruby object έχει claimed state αλλά άλλος worker μπορεί να το reclaim μετά τη λήξη lease. Ο provider call χρειάζεται το ίδιο stable deduplication key.

```ruby
class DispatchOutboxJob < ApplicationJob
  def perform
    OutboxEvent.claim_batch.each { |event| deliver(event) }
  end

  private

  def deliver(event)
    PartnerClient.publish(
      topic: event.topic,
      payload: event.payload,
      idempotency_key: event.deduplication_key
    )
    event.update!(status: :delivered, delivered_at: Time.current)
  rescue PartnerClient::TemporaryError
    event.update!(status: :pending, available_at: 1.minute.from_now)
    raise
  end
end
```

Αν το process πεθάνει μετά το publish και πριν το update, το lease λήγει και ο dispatcher ξαναστέλνει με ίδιο key. Αν ο partner δεν κάνει dedupe, ο consumer πρέπει να κρατά inbox ledger ή το duplicate παραμένει αποδεκτό business risk.

Failure classes και poison messages

Μην επιστρέφεις κάθε failure σε άπειρο retry. Invalid payload, revoked account και unknown topic είναι permanent μέχρι να αλλάξει data ή code. Μετά από bounded attempts, βάλε failed status με sanitized error class, report και operator path. Κράτα raw provider response μόνο αν επιτρέπεται από privacy και retention policy.

Το reconciliation query είναι απλό και πολύτιμο. Confirmed reservations χωρίς delivered outbox effect, pending events παλιότερα από threshold και processing leases που έληξαν. Dashboard μόνο με queue length δεν βλέπει logical intent που δεν μπήκε ποτέ στην queue.

Inbox για webhooks

Το αντίστροφο boundary έχει το ίδιο πρόβλημα. Provider μπορεί να στείλει webhook πολλές φορές ή εκτός σειράς. Επαλήθευσε signature πάνω στο raw body, περιόρισε μέγεθος, αποθήκευσε provider event id με unique index και απάντησε γρήγορα. Μετά επεξεργάσου το inbox row idempotently.

RUBY

```ruby
class InboxEvent < ApplicationRecord
  validates :provider_event_id, presence: true
end

def store_webhook!(provider_event_id:, payload:)
  InboxEvent.create!(provider_event_id: provider_event_id, payload: payload)
rescue ActiveRecord::RecordNotUnique
  InboxEvent.find_by!(provider_event_id: provider_event_id)
end
```

Το unique index πρέπει να υπάρχει παρότι δεν φαίνεται στο snippet. Η model validation μόνη της δεν σταματά concurrent duplicate deliveries.

Compensation αντί για ψεύτικο rollback

Σε workflow πολλών external βημάτων μπορεί να μην υπάρχει atomic commit. Μια saga καταγράφει state και ορίζει compensating action. Refund δεν είναι rollback του charge. Είναι νέο effect που μπορεί και αυτό να αποτύχει. Χρειάζεται δικό του idempotency key, retries και visibility.

Μη χτίζεις γενικό event bus πριν υπάρξει πραγματικό multi-step workflow. Για ένα κρίσιμο email, outbox table και ένας dispatcher είναι αρκετά. Η αξία είναι στο durable intent και στην επανάληψη, όχι στο πόσα abstractions περιβάλλουν το pattern.

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Η outbox κλείνει το lost-intent window. Το stable provider key κλείνει το duplicate-effect window. Η reconciliation βρίσκει ό,τι ξέφυγε και από τα δύο.

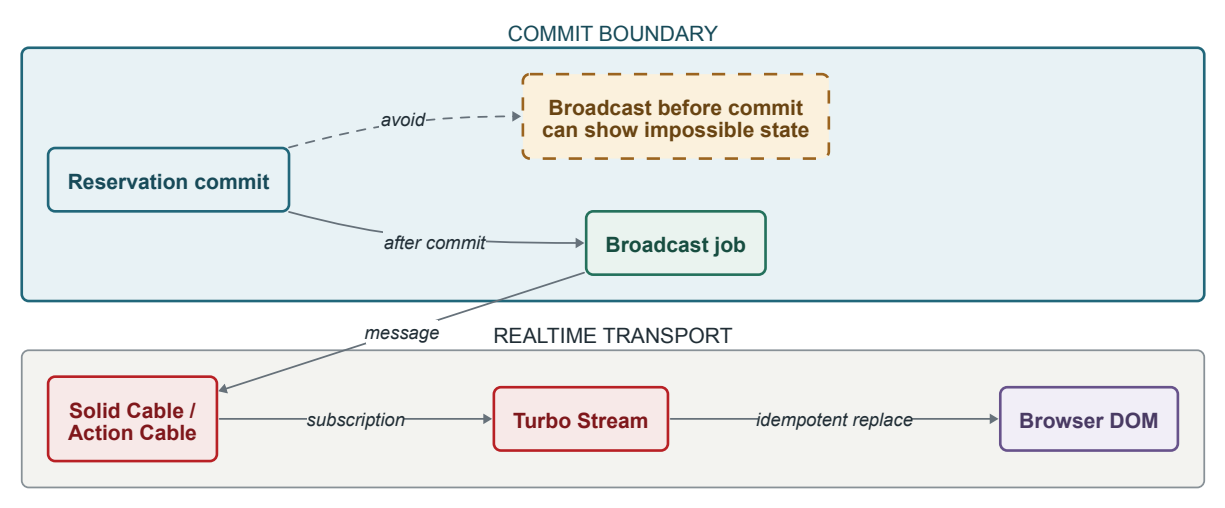
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Διάλεξε flow που γράφει database και καλεί provider. Σημείωσε τα δύο crash windows. Σχεδίασε outbox columns, logical deduplication key, lease και permanent failure status. Γράψε το reconciliation query πριν γράψεις dispatcher.

ΚΕΦΑΛΑΙΟ 16

Hotwire, Cable, Storage και async UI

To UI event είναι ειδοποίηση ότι κάτι άλλαξε. Η durable κατάσταση παραμένει η πηγή αλήθειας.



Σχήμα 16.1 · Μετά το commit, broadcast job μεταφέρει Turbo Stream στον browser.

Broadcast μετά το commit

Ένα Turbo Stream μπορεί να αλλάξει DOM χωρίς full page reload. Αν όμως γίνει broadcast πριν το database commit, ο browser μπορεί να δείξει state που τελικά έκανε rollback. Προτίμησε after-commit enqueue ή explicit outbox όταν η παράδοση είναι κρίσιμη.

RUBY

```ruby
class Reservation < ApplicationRecord
  after_update_commit :broadcast_status, if: :saved_change_to_status?

  private

  def broadcast_status
    broadcast_replace_later_to(
      event,
      target: self,
      partial: "reservations/reservation"
    )
  end
end
```

Το callback ταιριάζει αν κάθε status change πρέπει να ενημερώνει την ίδια representation. Αν μόνο ένα use case κάνει broadcast, βάλε την πρόθεση στην operation. Το `_later_` μεταφέρει rendering σε job αλλά φέρνει τα γνωστά queue semantics. Καθυστέρηση, retry και duplicate delivery.

Idempotent DOM operations

Το `replace` με σταθερό DOM id είναι ευκολότερο να επαναληφθεί από `append`. Δύο ίδια append events δημιουργούν διπλά στοιχεία. Για collection insert χρειάζεται stable element id ή έλεγχος existence στον client.

Out-of-order updates είναι επίσης πιθανά. Job για `confirmed` μπορεί να καθυστερήσει και να φτάσει μετά από job για `cancelled`. Το partial που κάνει render από current database state είναι ασφαλέστερο από payload με παλιό HTML. Για υψηλό event rate, βάλε version στο message και αγνόησε μικρότερη version στον client ή coalesce updates.

```erb
<%= turbo_stream.replace dom_id(@reservation) do %>
  <%= render @reservation.reload %>
<% end %>
```

Το reload πρέπει να διαβάζει από κατάλληλο role. Replica lag μπορεί να ξαναφέρει παλιό state σε broadcast job. Για read-after-write representation, χρησιμοποίησε writer database.

Action Cable είναι transport

Μια Cable connection ζει περισσότερο από ένα HTTP request. Κάνει authentication στο connection και authorization όταν γίνεται subscription. Το ότι κάποιος γνωρίζει stream name δεν πρέπει να αρκεί για πρόσβαση. Turbo χρησιμοποιεί signed stream names για τα δικά του helpers, αλλά custom channels χρειάζονται explicit policy.

```ruby
class ReservationsChannel < ApplicationCable::Channel
  def subscribed
    reservation = Reservation.find(params.fetch("reservation_id"))
    reject unless reservation.user_id == current_user.id

    stream_for reservation
  end
end
```

Το `current_user` της Cable connection πρέπει να προκύπτει από verified session ή token. Μην εμπιστεύεσαι user id από subscription params.

Solid Cable μπορεί να αποθηκεύει messages σε database-backed backend. Redis ή άλλος adapter μπορεί να είναι καλύτερος για διαφορετικό latency και scale. Σε κάθε περίπτωση, η Cable παράδοση δεν είναι ledger business events. Client που έχασε σύνδεση πρέπει να μπορεί να ξαναφορτώσει current state με κανονικό HTTP.

Reconnect και recovery

Σχεδίασε τη σελίδα ώστε refresh να διορθώνει κάθε χαμένο stream. Για κρίσιμο workflow, ο client μπορεί μετά το reconnect να ζητά resources changed since γνωστή version. Το live message βελτιώνει latency. Το read API αποκαθιστά συνέπεια.

Μην βάζεις μη αναστρέψιμη business action μόνο σε Stimulus callback που μπορεί να μη τρέξει. Η ενέργεια πρέπει να έχει server endpoint με idempotency και σαφές result. Το UI απλώς την καλεί και παρουσιάζει state.

Active Storage έχει δύο κόσμους

Το Active Storage κρατά blob και attachment metadata στη database αλλά τα bytes ζουν σε storage service. Database rollback δεν διαγράφει upload που έχει ήδη ολοκληρωθεί. Direct upload δημιουργεί blob πριν συνδεθεί απαραίτητα με record. Abandoned forms αφήνουν unattached blobs και χρειάζονται retention cleanup.

```ruby
class ReservationDocument < ApplicationRecord
  has_one_attached :file

  validate :acceptable_content_type

  private

  def acceptable_content_type
    return unless file.attached?
    return if file.content_type.in?(%w[application/pdf image/jpeg image/png])

    errors.add(:file, "has unsupported type")
  end
end
```

Το client-provided content type δεν είναι επαρκής security έλεγχος. Για untrusted uploads χρειάζεσαι size limits πριν και μετά το upload, content inspection, malware policy, safe download headers και authorization σε κάθε access path. Public bucket URL μπορεί να παρακάμψει controller authorization.

Variant processing και analysis τρέχουν συχνά σε jobs. Η πρώτη προβολή μπορεί να ζητήσει variant πριν ολοκληρωθεί analysis. Κράτα UI state που αντέχει `processing`, retryable failure και deleted original. Μην υποθέτεις ότι επειδή υπάρχει attachment row υπάρχουν και προσβάσιμα bytes για πάντα.

Purge, retention και references

Το `purge_later` αφαιρεί metadata και storage object ασύγχρονα. Αν το ίδιο blob είναι συνδεδεμένο αλλού ή υπάρχει legal retention, η διαγραφή θέλει policy. Επίσης CDN copies και provider versioning μπορεί να κρατούν bytes μετά την application deletion. Περιέγραψε το πραγματικό retention contract.

Mail ως async UI

Email είναι επίσης external representation της κατάστασης. `deliver_later` κληρονομεί job retries και duplication. Link μέσα στο email πρέπει να οδηγεί σε current authorized state, όχι να θεωρεί ότι το email έφτασε μία φορά και στη σωστή σειρά. Signed ids έχουν expiry και purpose ώστε ένα token για confirmation να μη χρησιμοποιείται ως γενικό record access.

ΠΡΟΣΟΧΗ

Realtime transport μειώνει τον χρόνο μέχρι να δει ο χρήστης την αλλαγή. Δεν αυξάνει από μόνο του durability, ordering ή authorization.

Οι επίσημες αναφορές είναι τα Action Cable Overview, Active Storage Overview και Working with JavaScript in Rails.

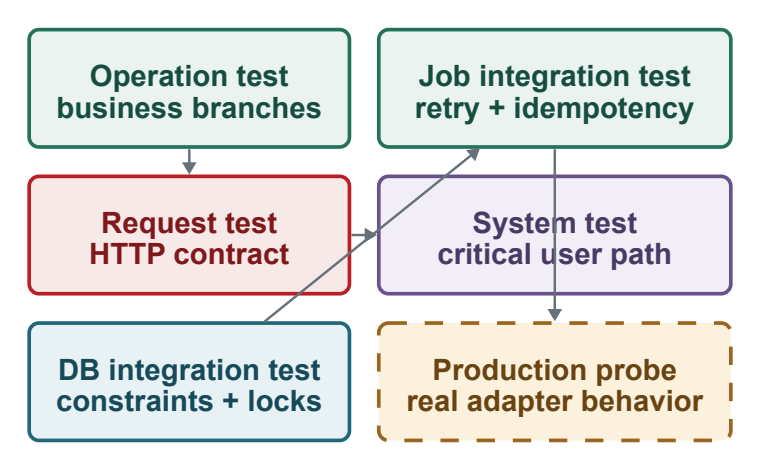
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Κόψε προσωρινά τη live σύνδεση σε μία πραγματική οθόνη και άλλαξε το record από δεύτερο session. Μετά κάνε reconnect και refresh. Αν η οθόνη δεν επανέρχεται σε σωστό current state, διόρθωσε το read path πριν προσθέσεις περισσότερα broadcasts.

ΚΕΦΑΛΑΙΟ 17

Tests για production failure modes

To test level δεν επιλέγεται από τον φάκελο του κώδικα. Επιλέγεται από την εγγύηση που θέλεις να αποδείξεις.



Σχήμα 17.1 · Κάθε failure mode χρειάζεται το test boundary που περιλαμβάνει τον μηχανισμό του.

Behavior, όχι implementation choreography

Test που περιγράφει «καλεί το service A, μετά το service B» σπάει όταν αναδιατάξεις κώδικα χωρίς να αλλάξεις αποτέλεσμα. Test που περιγράφει «μία θέση δίνει μία confirmed reservation και ένα durable notification intent» προστατεύει το contract.

Για κάθε use case γράψε success, expected rejection και μία κρίσιμη failure διαδρομή. Μετά διάλεξε το χαμηλότερο boundary που περιλαμβάνει όλους τους μηχανισμούς της εγγύησης.

Εγγύηση	Κατάλληλο boundary	Τι δεν αποδεικνύει
Transition ενός object	Model ή plain object test	HTTP και database race
HTTP status και payload	Request test	Browser behavior
Unique constraint	Database integration test	Friendly validation message
Retry και idempotency	Job integration test	Provider production semantics
Critical user path	System test	Κάθε business branch

Application operation test

Η operation είναι καλό σημείο για business branches χωρίς browser. Χρησιμοποίησε πραγματικά models όταν η transaction και τα constraints είναι μέρος του contract. Fake μόνο τα external boundaries.

```ruby
class ReservationsConfirmTest < ActiveSupport::TestCase
  test "confirms once and records notification intent" do
    reservation = reservations(:pending)

    assert_difference -> { OutboxEvent.count }, 1 do
      Reservations::Confirm.call(
        reservation: reservation,
        idempotency_key: "request-123"
      )
    end

    assert_predicate reservation.reload, :confirmed?
    assert_equal "reservation.confirmed.v1", OutboxEvent.last.topic
  end
end
```

Αν χρησιμοποιείς RSpec, η ίδια αρχή ισχύει. Η επιλογή testing DSL δεν αλλάζει το boundary. Μην mock-άρεις `Reservation.transaction` και `update!` αν ακριβώς αυτά θέλεις να ελέγξεις.

Database guarantees ξεχωριστά

Η model validation μπορεί να αποτύχει πριν φτάσει στον unique index. Για να ελέγξεις τη βάση, χρησιμοποίησε API που παρακάμπτει το validation ή raw insert σε απομονωμένο integration test.

```ruby
class ReservationConstraintTest < ActiveSupport::TestCase
  test "database rejects duplicate idempotency key" do
    original = reservations(:confirmed)

    assert_raises ActiveRecord::RecordNotUnique do
      Reservation.insert_all!([
        {
          account_id: original.account_id,
          event_id: original.event_id,
          user_id: original.user_id,
          idempotency_key: original.idempotency_key,
          seats: 1,
          status: "confirmed",
          created_at: Time.current,
          updated_at: Time.current
        }
      ])
    end
  end
end
```

Το enum storage στο πραγματικό schema μπορεί να είναι integer και το sample τότε χρειάζεται το αντίστοιχο value. Το σημαντικό είναι ότι το test χτυπά το ίδιο constraint με production adapter. SQLite test δεν αποδεικνύει PostgreSQL partial index ή lock clause.

Request contract

Το request test καλεί router, controller stack, params handling, authentication και rendering. Έλεγξε explicit status, response schema και durable state. Μην κοιτάς controller instance variables.

```ruby
class ReservationsRequestTest < ActionDispatch::IntegrationTest
  test "duplicate confirmation returns the existing result" do
    reservation = reservations(:confirmed)

    post confirm_reservation_path(reservation),
      params: { idempotency_key: reservation.idempotency_key },
      as: :json

    assert_response :success
    assert_equal reservation.id, response.parsed_body.fetch("reservation_id")
  end
end
```

Για authorization γράψε τουλάχιστον allowed και forbidden actor πάνω στο ίδιο object. Ένα 404 αντί 403 μπορεί να είναι σωστό αν δεν θέλεις να αποκαλύψεις existence. Κράτα το contract σταθερό.

Query count ως regression guard

Το Rails 8.1 test API περιλαμβάνει assertions για query count και patterns. Χρησιμοποίησέ τα σε render ή serialization boundary όπου ένα νέο association call δημιουργεί N+1.

```ruby
class EventSerializerTest < ActiveSupport::TestCase
  test "serializes a preloaded page with bounded queries" do
    events = Event.order(:id).limit(20).preload(:venue, :reservations)

    assert_queries_count(3) do
      EventSerializer.new(events).as_json
    end
  end
end
```

Το ακριβές count μπορεί να γίνει brittle αν schema queries ή unrelated instrumentation αλλάζουν. Το `include_schema: false` είναι default στα σχετικά assertions. Για ορισμένα boundaries είναι καλύτερο να απαγορεύσεις συγκεκριμένο pattern ή να ελέγξεις upper bound με δικό σου helper. Το test πρέπει να προστατεύει cost shape, όχι τυχαία αρίθμηση.

Jobs σε δύο επίπεδα

Στο unit επίπεδο κάλεσε `perform_now` και έλεγξε idempotent behavior. Στο context επίπεδο χρησιμοποίησε `perform_enqueued_jobs` ώστε να αποδείξεις ότι το use case κάνει enqueue μετά από commit και το job βρίσκει το row. External provider μένει fake με contract που καταγράφει idempotency key.

Δοκίμασε duplicate delivery καλώντας το ίδιο job δύο φορές. Δοκίμασε timeout μετά το fake provider effect και πριν το local mark. Έλεγξε την τελική business invariant, όχι απλώς ότι έγινε retry.

Transactional tests και το κρυφό όριο

Τα Rails tests τυλίγονται συνήθως σε transactions. Αυτό κρατά τη βάση καθαρή αλλά μπορεί να αλλάξει το πότε τρέχουν after-commit callbacks ή τι βλέπει δεύτερη connection. Concurrency test με threads χρειάζεται πραγματικές ξεχωριστές connections και cleanup strategy που δεν κρύβει uncommitted fixtures.

Μην απενεργοποιείς transactional tests για όλο το suite επειδή ένα test το χρειάζεται. Φτιάξε μικρή integration κατηγορία με σαφή cleanup. Τρέξ' την με τον ίδιο database engine και isolation που χρησιμοποιεί production.

Time και nondeterminism

Χρησιμοποίησε `travel_to` για expiry logic αλλά έλεγξε και timezone boundary. Μη στηρίζεσαι σε `sleep`. Για job scheduling έλεγξε scheduled timestamp ή advance controlled time. Randomized test order αποκαλύπτει shared global state. Όταν εμφανιστεί failure, κράτα το seed και διόρθωσε την απομόνωση.

Parallel tests αποκαλύπτουν hard-coded unique values, shared files και global singletons. Δεν πρέπει να απενεργοποιείς parallelism μόνο για να κρύψεις race του test setup. Ξεχώρισε όμως test-only collision από πραγματικό production concurrency claim.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Το test πρέπει να αποτυγχάνει όταν σπάσει η εγγύηση και να μένει πράσινο όταν αλλάξει μόνο η εσωτερική διαδρομή. Αυτό είναι πιο χρήσιμο από υψηλό line coverage.

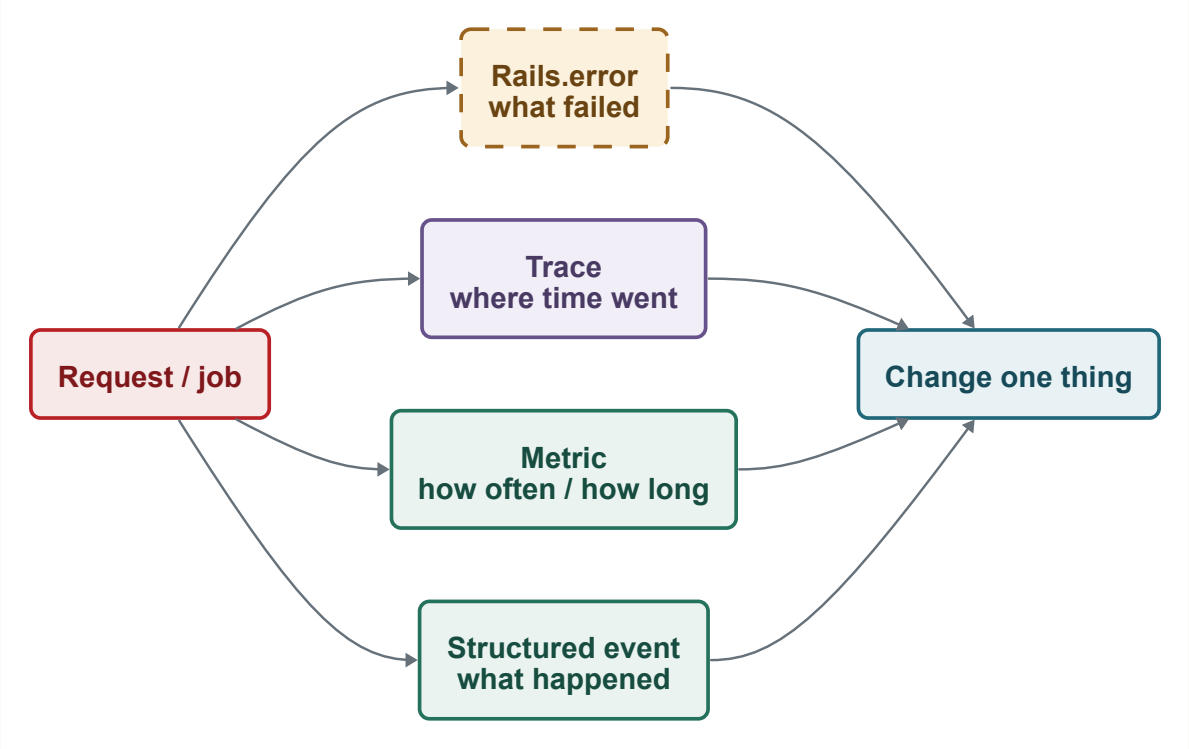
Η πλήρης Rails αναφορά είναι το Testing Rails Applications.

> **ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ**
>
> Πάρε ένα production incident ή σοβαρό bug. Γράψε πρώτα την ανεπιθύμητη τελική κατάσταση. Βρες το χαμηλότερο test boundary που μπορεί να την αναπαράγει χωρίς να mock-άρει τον μηχανισμό που έσπασε. Πρόσθεσε το test πριν κάνεις refactor.

ΚΕΦΑΛΑΙΟ 18

Instrumentation, events και errors

Logs λένε τι συνέβη. Metrics λένε πόσο συχνά. Traces λένε πού πήγε ο χρόνος. Error reports κρατούν το failure μαζί με context.



Σχήμα 18.1 · Requests και jobs παράγουν διαφορετικά feedback signals που οδηγούν σε μία συγκεκριμένη αλλαγή.

Ξεκίνα από ερώτηση

«Βάλε observability» δεν είναι task. «Γιατί το confirm endpoint έχει p95 πάνω από 800 ms;» είναι ερώτηση. Χρειάζεσαι συνολικό latency, queue wait, SQL time, lock wait, external call time και allocation signal. Κάθε signal πρέπει να οδηγεί σε απόφαση.

Βάλε request id σε logs, events και error context. Βάλε job id και logical operation id στα background flows. Μη χρησιμοποιείς email, token ή ολόκληρο payload ως correlation key.

Framework notifications

Το `ActiveSupport::Notifications` εκθέτει events για controllers, SQL, jobs, cache, rendering και storage. Subscriber παίρνει duration, allocations και payload ανάλογα με το event.

```ruby
ActiveSupport::Notifications.subscribe("process_action.action_controller") do |
event|
  Rails.logger.info(
    event: "rails.request",
    controller: event.payload[:controller],
    action: event.payload[:action],
    duration_ms: event.duration,
    allocations: event.allocations
  )
end
```

Subscriber τρέχει μέσα στο observed execution path. Βαρύ serialization ή network call εκεί προσθέτει latency σε κάθε request. Προτίμησε γρήγορο adapter, sampling και bounded payload.

Το `sql.active_record` δίνει SQL, binds, async flag, cached flag, row count και current transaction όπου υποστηρίζονται. Μην στέλνεις raw binds σε logs αν μπορεί να περιέχουν προσωπικά δεδομένα. Για query fingerprint αφαίρεσε literal values και κράτα operation name.

Structured events στο Rails 8.1

Το Rails 8.1 πρόσθεσε Event Reporter με `Rails.event`. Η εφαρμογή εκπέμπει named event και structured payload. Subscribers αποφασίζουν serialization και προορισμό.

```ruby
Rails.event.notify(
  "reservation.confirmed",
  reservation_id: reservation.id,
  account_id: reservation.account_id,
  seats: reservation.seats
)
```

Το event name πρέπει να περιγράφει γεγονός, όχι log sentence. Κράτα schema μικρό, documented και low-cardinality όπου προορίζεται για metrics. Το `reservation_id` είναι χρήσιμο για trace ή audit search αλλά καταστροφικό ως metric label σε time-series backend.

Context μπορεί να εφαρμοστεί μία φορά στο request boundary.

```ruby
class ApplicationController < ActionController::Base
  before_action :set_event_context

  private

  def set_event_context
    Rails.event.set_context(
      request_id: request.request_id,
      account_id: Current.account&.id
    )
  end
end
```

Μην βάζεις secrets, authorization headers ή ανεπεξέργαστο request body στο context. Το structured format κάνει τα δεδομένα ευκολότερα στην αναζήτηση και άρα αυξάνει την ανάγκη για privacy discipline.

`Rails.error` για failures

Το Rails τυλίγει requests, jobs και runner executions με error reporter. Ένας subscriber μπορεί να στείλει unhandled errors σε monitoring service χωρίς custom middleware. Για handled exception χρησιμοποίησε το κατάλληλο API και δήλωσε context.

```ruby
def fetch_partner_status(reservation)
  Rails.error.record(context: { reservation_id: reservation.id }) do
    PartnerClient.status(reservation.external_id)
  end
end
```

Το `record` αναφέρει και ξανασηκώνει. Το `handle` αναφέρει και καταπίνει. Μην διαλέγεις `handle` επειδή θέλεις λιγότερα errors. Χρησιμοποίησέ το μόνο όταν υπάρχει πραγματικό fallback και η συνέχεια είναι ασφαλής.

```ruby
def optional_recommendations(user)
  Rails.error.handle(RecommendationService::Unavailable, fallback: -> { [] })
do
    RecommendationService.for(user)
  end
end
```

Το error class, handled flag και source είναι πιο χρήσιμα από free-form message aggregation. Κράτα stack trace για debugging αλλά ομαδοποίησε με σταθερό fingerprint ώστε ένα incident να μην γίνει χιλιάδες διαφορετικές ειδοποιήσεις.

Metrics με σωστό denominator

Μετράς reservation conflicts. Ο αριθμός 200 μόνος του δεν λέει τίποτα. Θέλεις conflicts ανά confirm attempts, χωρισμένα ίσως ανά endpoint version ή shard. Δεν θέλεις label ανά user ή reservation.

Latency θέλει distribution, όχι average. p50 δείχνει typical, p95 και p99 δείχνουν tail. Queue latency μετριέται από enqueue μέχρι start. Execution latency από start μέχρι finish. End-to-end business latency από durable intent μέχρι delivered effect.

Traces και boundaries

Ένα trace πρέπει να δείχνει request, SQL, cache, external HTTP και enqueued job ως ξεχωριστά spans με propagation id. Δεν χρειάζεται να trace-άρεις κάθε Ruby method. Βάλε spans γύρω από boundaries όπου περιμένεις I/O ή όπου αλλάζει owner.

Sampling 1% μπορεί να χάσει σπάνιο error. Κράτα head sampling για συνηθισμένα requests και tail ή forced sampling για failures αν το tooling το επιτρέπει. Μην αυξήσεις sampling χωρίς να μετρήσεις κόστος και data retention.

Από dashboard σε runbook

Κάθε alert χρειάζεται symptom, threshold, χρονικό παράθυρο, owner και πρώτο diagnostic query. CPU 90% για δέκα δευτερόλεπτα μπορεί να είναι φυσιολογικό. Oldest critical job πάνω από πέντε λεπτά μπορεί να σημαίνει user-visible failure ακόμα κι αν CPU είναι χαμηλή.

Σύνδεσε deploy marker με metrics και errors. Αν το p95 ανέβηκε αμέσως μετά το release, το diff είναι evidence. Δεν είναι απόδειξη αιτίας αλλά μικραίνει πολύ το search space.

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Το καλύτερο event έχει όνομα, σταθερό schema, correlation id και σαφή καταναλωτή. Αν κανείς δεν μπορεί να πει ποια απόφαση θα πάρει από αυτό, μην το εκπέμπεις ακόμα.

Οι επίσημες αναφορές είναι το Active Support Instrumentation, το Error Reporting και τα Rails 8.1 release notes.

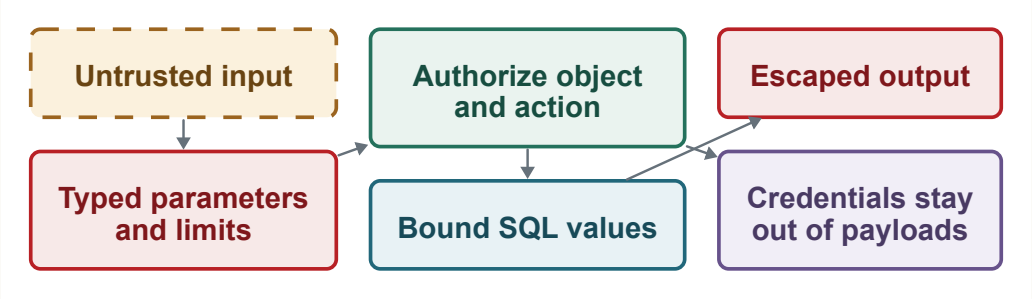
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Διάλεξε ένα critical endpoint. Γράψε τρεις ερωτήσεις που σήμερα δεν μπορείς να απαντήσεις. Για καθεμία επίλεξε ένα event, metric ή trace span με bounded cardinality. Πρόσθεσε πρώτα μόνο το signal που οδηγεί σε συγκεκριμένη απόφαση.

ΚΕΦΑΛΑΙΟ 19

Security boundaries

Το Rails δίνει ασφαλείς defaults και helpers. Η εφαρμογή εξακολουθεί να αποφασίζει ποιον εμπιστεύεται, για ποιο object και για ποια ενέργεια.



Σχήμα 19.1 · Κάθε untrusted input περνά shape, authorization, bound query και escaped output.

Authentication δεν είναι authorization

Το `Current.user` απαντά ποιος είναι ο caller. Δεν απαντά αν μπορεί να δει αυτή τη reservation. Το ασφαλέστερο lookup ξεκινά από authorized scope αντί να φορτώνει global record και να ελέγχει αργότερα.

```ruby
class ReservationsController < ApplicationController
  def show
    reservation = Current.user.reservations.find(params.require(:id))
    render json: ReservationSerializer.new(reservation).as_json
  end
end
```

Για admin ή account-level access, το scope πρέπει να εκφράζει την πραγματική policy. Strong parameters δεν κάνουν authorization. Απλώς περιορίζουν fields που μπορούν να περάσουν σε mass assignment.

```ruby
def reservation_params
  params.expect(reservation: %i[event_id seats])
end
```

Μην επιτρέπεις `account_id`, `user_id`, `role` ή `status` από client αν πρέπει να προκύπτουν από authenticated context ή server transition.

Injection σε κάθε γλώσσα

Hash conditions και bind placeholders προστατεύουν SQL values. Dynamic column, order direction και function name είναι SQL syntax και χρειάζονται allowlist.

```ruby
SORTS = {
  "soonest" => { starts_at: :asc },
  "latest" => { created_at: :desc }
}.freeze

order = SORTS.fetch(params.fetch(:sort, "soonest"), SORTS.fetch("soonest"))
events = Event.published.order(order)
```

Το ίδιο boundary υπάρχει στο shell. Μην χτίζεις command string από filename ή URL. Χρησιμοποίησε argument array με `system`, `Open3` ή library API. Ακόμα και τότε χρειάζεσαι size, path και resource limits. Safe escaping δεν σταματά zip bomb ή command που επιτρέπεται αλλά κοστίζει υπερβολικά.

XSS και safe HTML

Το ERB κάνει escape κανονικά το output. Το `raw`, `html_safe` και custom sanitization αφαιρούν αυτή την προστασία. Μην σημειώνεις user content safe επειδή «έρχεται από Markdown renderer». Renderer configuration, URL schemes και embedded HTML είναι μέρος του threat model.

Χρησιμοποίησε permitted tags και attributes με Rails sanitizer όταν χρειάζεται rich content. Content Security Policy μειώνει impact αλλά δεν διορθώνει unsafe rendering. Test με payloads σε text, attribute, URL και script contexts επειδή το σωστό escaping εξαρτάται από το context.

CSRF και session semantics

Σε browser app με cookie authentication, ο browser στέλνει cookie και σε cross-site request όταν οι cookie rules το επιτρέπουν. Το Rails authenticity token προστατεύει state-changing requests. Μην απενεργοποιείς CSRF global για να δουλέψει ένα API endpoint. Χώρισε cookie-auth browser controllers από token-auth API boundary και όρισε σωστό CORS policy.

Μετά από login κάνε session rotation ώστε attacker-provided session id να μην επιβιώσει. Μετά από password reset ή privilege change, σκέψου server-side invalidation όλων των παλιών sessions. Το default encrypted CookieStore κρατά session data στον client και έχει μικρό size limit. Encryption προστατεύει το περιεχόμενο όσο το secret μένει ασφαλές. Δεν κάνει το cookie άμεσα revocable χωρίς δικό σου server-side version ή session record.

Signed και encrypted μηνύματα

Signed id αποδεικνύει ότι το token δημιουργήθηκε από την εφαρμογή και δεν άλλαξε. Δεν κρύβει αναγκαστικά το underlying id. Encrypted message κρύβει και προστατεύει το content. Βάλε purpose και expiry ώστε password-reset token να μην χρησιμοποιείται σε άλλο endpoint ή για πάντα.

```ruby
token = user.signed_id(purpose: :email_confirmation, expires_in: 30.minutes)
user = User.find_signed!(token, purpose: :email_confirmation)
```

Authorization ξανατρέχει όταν χρησιμοποιείται το token. Έγκυρη υπογραφή δεν σημαίνει ότι ο account είναι ακόμα ενεργός ή ότι η ενέργεια δεν ολοκληρώθηκε.

Secrets και logs

Rails credentials βοηθούν να αποθηκεύεις encrypted configuration στο repo με key έξω από αυτό. Environment variables και external secret stores είναι άλλες επιλογές. Σε όλες, περιόρισε ποιο process μπορεί να διαβάσει τι. Μην εμφανίζεις secret σε console transcript, exception context, structured event ή job payload.

Το `filter_parameters` πρέπει να καλύπτει password, token, authorization, secret, card-like fields και domain-specific προσωπικά δεδομένα. Έλεγξε πραγματικό log sample με synthetic values. Μην υποθέτεις ότι gem-specific field name θα φιλτραριστεί μόνο του.

URLs, redirects και server-side fetch

Open redirect επιτρέπει phishing μέσω έμπιστου domain. Μη δίνεις `allow_other_host: true` σε URL από params χωρίς permitted hosts και scheme. Για return path, προτίμησε signed internal path ή named route.

Server-side URL fetch έχει κίνδυνο SSRF. DNS μπορεί να δείξει internal IP, redirect μπορεί να αλλάξει host και IPv6 notation μπορεί να παρακάμψει απλό string filter. Προτίμησε provider-specific clients και allowlisted hosts. Κάνε validation μετά από κάθε redirect και βάλε connect, read και total limits.

Uploads και background boundaries

Filename, content type και metadata είναι untrusted. Αποθήκευσε bytes έξω από public executable path, δημιούργησε safe server filename και σέρβιρε με σωστό Content-Disposition. Image processing και document parsing είναι attack surface. Τρέξε τα με resource limits και ενημερωμένα libraries.

Job payload μπορεί να έχει δημιουργηθεί από παλιό release ή compromised producer. Κάνε authorization και precondition check ξανά όταν εκτελείται critical job. Μην θεωρείς ότι επειδή το job είναι «εσωτερικό» το record ανήκει ακόμα στον ίδιο account.

Admin και blast radius

Admin interface χρειάζεται ισχυρό authentication, μικρό session lifetime, explicit authorization και audit events για επικίνδυνες ενέργειες. Hide button δεν είναι permission. Bulk export πρέπει να έχει row scope, rate limit και retention. Support impersonation πρέπει να φαίνεται καθαρά, να λήγει και να καταγράφει πραγματικό actor μαζί με impersonated subject.

ΚΕΝΤΡΙΚΗ ΙΔΕΑ

Σε κάθε boundary γράψε actor, object, action, trusted fields και output context. Μετά έλεγξε ότι η απόφαση γίνεται server-side στη στιγμή της ενέργειας.

Η βασική επίσημη αναφορά είναι το Securing Rails Applications. Οι security fixes έρχονται και ως Rails patch releases, άρα η συντήρηση dependencies είναι μέρος της εφαρμογής και όχι ξεχωριστή δουλειά.

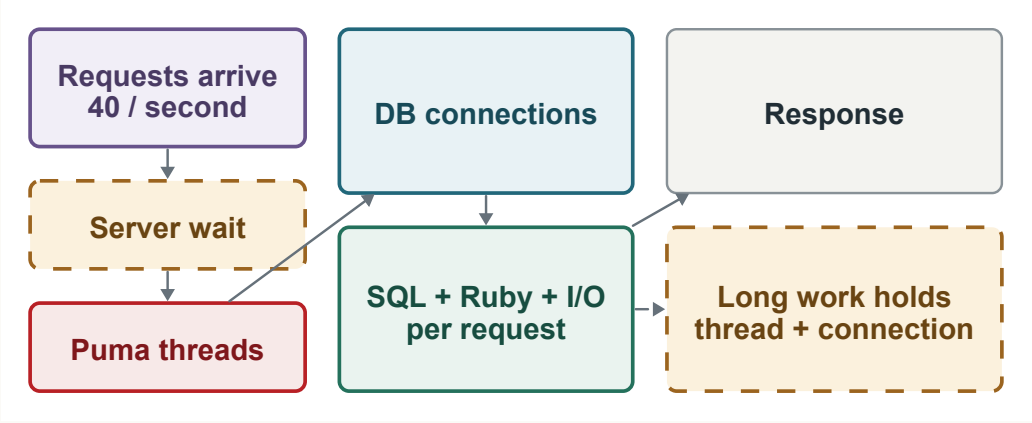
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Πάρε ένα admin action και ακολούθησε actor, params, lookup, authorization, query, render και log. Δοκίμασε object άλλου account, extra role param, external return URL και HTML payload. Κατέγραψε κάθε σημείο όπου ο κώδικας βασίζεται μόνο στο UI.

ΚΕΦΑΛΑΙΟ 20

Performance και capacity

Latency είναι χρόνος ενός request. Throughput είναι requests ανά χρόνο. Capacity είναι πόσο work αντέχεις πριν οι ουρές αρχίσουν να μεγαλώνουν.



Σχήμα 20.1 · Ένα αργό request κρατά server slot και συχνά database connection, μειώνοντας συνολική capacity.

Μέτρα όλη την αναμονή

Ο χρήστης βλέπει χρόνο από την άφιξη μέχρι την απόκριση. Το APM μπορεί να μετρά μόνο τον χρόνο αφού ένα Puma thread πάρει το request. Αν όλα τα threads είναι busy, υπάρχει queue time μπροστά από την εφαρμογή. Η διάκριση εξηγεί γιατί το application duration φαίνεται σταθερό ενώ το end-to-end latency ανεβαίνει.

Κατέγραψε τουλάχιστον request rate, queue time, application time, SQL time, external I/O time, allocations, error rate και saturation. Χρησιμοποίησε p50, p95 και p99. Το average κρύβει μικρό ποσοστό πολύ αργών requests που κρατούν disproportionate capacity.

Πρώτα το work ανά request

Πριν αλλάξεις Puma settings, αφαίρεσε περιττό work. N+1 queries, μεγάλα result sets, repeated serialization, unbounded regex και synchronous provider calls δεν θεραπεύονται με περισσότερα threads. Περισσότερη concurrency μπορεί να στείλει το ίδιο ακριβό work γρηγορότερα στη database και να αυξήσει contention.

Ένα πρακτικό breakdown για αργό endpoint είναι το εξής.

1. Πόσο περίμενε πριν πάρει server slot.
2. Πόσα SQL queries έκανε και πόσα rows επέστρεψαν.
3. Πόσα model objects και strings δημιουργήθηκαν.

4. Πόσο κράτησε database connection ή transaction.
5. Ποιο external I/O έγινε inline.
6. Πόσο χρόνο πήρε render και response transfer.

Άλλαξε πρώτα το μεγαλύτερο μετρημένο component που μπορείς να μειώσεις χωρίς να αλλάξεις correctness.

Puma processes και threads

Στο CRuby το Global VM Lock επιτρέπει ένα thread ανά process να εκτελεί Ruby code κάθε στιγμή. Threads μπορούν να επικαλύψουν waits σε database, network ή άλλο I/O. Processes χρησιμοποιούν διαφορετικούς CPU cores αλλά κοστίζουν περισσότερη μνήμη.

Δεν υπάρχει universal «5 threads, 2 workers». CPU-heavy rendering μπορεί να θέλει λιγότερα threads και περισσότερα processes. I/O-heavy app μπορεί να κερδίσει από περισσότερα threads μέχρι να αυξηθεί tail latency ή να κορεστεί κάποιο dependency.

```ruby
workers Integer(ENV.fetch("WEB_CONCURRENCY", 2))
threads_count = Integer(ENV.fetch("RAILS_MAX_THREADS", 3))
threads threads_count, threads_count

preload_app!
```

Το snippet είναι baseline για μέτρηση, όχι prescription. Το database pool κάθε process πρέπει να ταιριάζει με τη μέγιστη ταυτόχρονη database χρήση. Πρόσθεσε web, job, Cable και maintenance processes πριν συγκρίνεις με database limit.

Copy-on-write και preload

Με process workers, preload μπορεί να επιτρέψει να μοιραστούν read-only memory pages από το parent. Κάθε μεταβολή μετά το fork κάνει copy τη σχετική page. Μεγάλα mutable registries και eager caches μειώνουν το όφελος. Μέτρα resident memory ανά worker μετά από warmup, όχι μόνο αμέσως στο boot.

Hooks μετά το fork πρέπει να ξανανοίγουν resources που δεν είναι ασφαλές να μοιραστούν. Σύγχρονο Rails και Puma διαχειρίζονται πολλά defaults αλλά custom sockets, clients και threads χρειάζονται lifecycle review.

YJIT με πραγματικό workload

Το YJIT μπορεί να βελτιώσει Rails throughput και latency σε κατάλληλο Ruby version, με επιπλέον memory cost και warmup. Ενεργοποίησέ το σε αντιπροσωπευτικό canary ή load test. Μέτρα throughput, tail latency και RSS μετά από σταθερό warmup. Μην κρατήσεις αλλαγή μόνο επειδή microbenchmark μιας helper method έγινε γρηγορότερο.

Ruby και Rails upgrades μπορεί να αλλάξουν JIT behavior. Κατέγραψε ακριβείς versions στο benchmark ώστε το αποτέλεσμα να επαναλαμβάνεται.

Allocations και garbage collection

Πολλά μικρά objects αυξάνουν GC work και memory. Active Record instantiation, JSON serializers και template helpers είναι συχνές πηγές. Το `process_action.action_controller` event δίνει allocations. Συνδύασέ το με profiler όταν ένα endpoint ξεχωρίζει.

Μείωσε rows και fields πριν κάνεις manual string tricks. `pluck` ή aggregation στη βάση μπορεί να αφαιρέσει χιλιάδες model instances. Cached primitive result μπορεί να βοηθήσει repeated expensive calculation. Πρόσεξε να μη μεταφέρεις τεράστια objects σε global cache που αυξάνει process RSS.

Memory leak δεν είναι κάθε αυξανόμενο RSS. Allocator fragmentation, JIT code και warm caches μπορούν να σταθεροποιηθούν σε υψηλότερο plateau. Κοίτα growth μετά από επαναλαμβανόμενο ίδιο workload και heap profiles. Worker recycling περιορίζει blast radius αλλά δεν αντικαθιστά root-cause investigation.

Database ως shared capacity

Query latency ανεβαίνει όταν αυξηθούν active connections, lock waits, disk I/O ή cache misses. Application server tuning που διπλασιάζει concurrent queries μπορεί να μειώσει συνολικό throughput. Μέτρα database CPU, active sessions, wait events και slow-query distribution μαζί με Rails metrics.

Connection checkout time είναι ουρά μέσα στο process. Database execution time είναι work μετά το checkout. Άλλες λύσεις διορθώνουν τα δύο. Μικρότερες transactions και queries απελευθερώνουν capacity. Μεγαλύτερο pool απλώς μειώνει την πρώτη ουρά μέχρι να εμφανιστεί αλλού.

Load test που μοιάζει με production

Ζέστανε app, JIT, database caches και connection pools πριν μετρήσεις steady state. Μετά δοκίμασε και cold start χωριστά. Χρησιμοποίησε mix endpoints και data cardinality που μοιάζουν με production. Ένα benchmark μόνο στο `/health` μετρά routing και server overhead, όχι την εφαρμογή.

Αύξησε load σταδιακά μέχρι να δεις saturation. Σημείωσε το σημείο όπου queue time και tail latency αρχίζουν να ανεβαίνουν μη γραμμικά. Μετά κράτα headroom για deploy overlap, cache outage, retry burst και ένα αργό dependency.

Μη load-test-άρεις production χωρίς σαφή έγκριση και safeguards. Ένα test που γράφει reservations ή στέλνει mail είναι πραγματική external action.

Performance budget ανά boundary

Για critical endpoint βάλε budget σε query count, returned rows, external calls και response size. Δεν χρειάζεται να γίνει κάθε test timing benchmark, επειδή CI noise είναι μεγάλο. Κράτα structural assertions στο suite και calibrated load test χωριστά.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Βελτιστοποίησε work πριν concurrency. Μετά ρύθμισε processes, threads και pools ως ενιαίο capacity model. Κάθε αύξηση σε parallelism πρέπει να έχει μετρημένο downstream headroom.

Η επίσημη βάση είναι το Tuning Performance for Deployment.

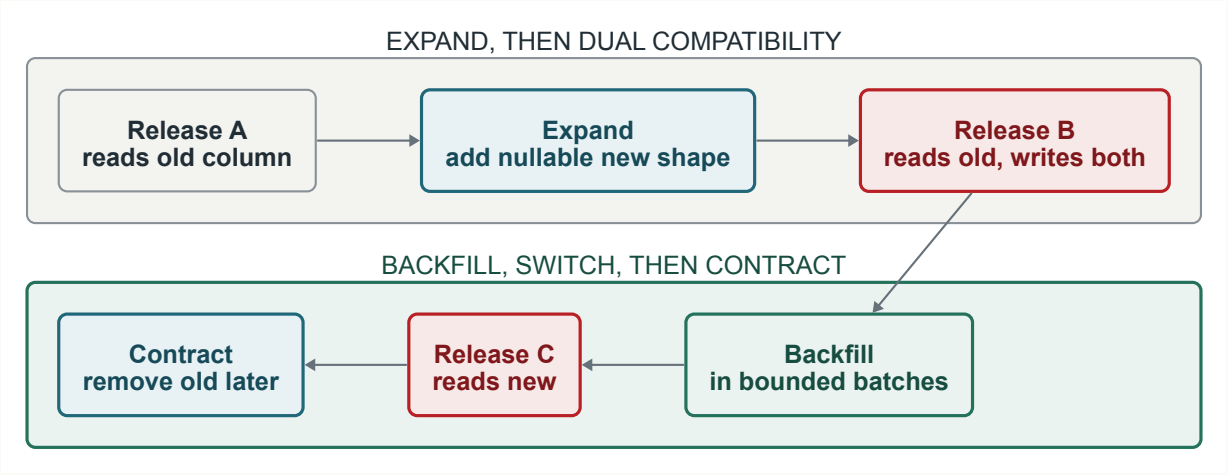
ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ

Διάλεξε ένα endpoint με αρκετό traffic. Φτιάξε breakdown queue, Ruby, SQL, external I/O και render. Βρες το saturation point σε ασφαλές test περιβάλλον με τρέχον process/pool setup. Άλλαξε μία μόνο παράμετρο και ξαναμέτρα p95, p99, throughput, database load και RSS.

ΚΕΦΑΛΑΙΟ 21

Upgrades, migrations και deploys χωρίς downtime

Σε rolling deploy το παλιό και το νέο release τρέχουν μαζί. Κάθε αλλαγή πρέπει να είναι συμβατή και με τα δύο μέχρι να τελειώσει η μετάβαση.



Σχήμα 21.1 · Η ασφαλής αλλαγή schema περνά από expand, dual compatibility, backfill και αργότερα contract.

Δύο versions ταυτόχρονα

Το release B ξεκινά ενώ instances του A εξυπηρετούν ακόμα requests και jobs. Job που μπήκε από A μπορεί να εκτελεστεί από B. Web response από A μπορεί να στείλει Turbo event μετά το boot του B. Rollback σημαίνει ότι A μπορεί να επιστρέψει αφού B έχει ήδη γράψει νέα data shape.

Αυτό είναι το βασικό compatibility test. Όχι μόνο «τρέχουν τα tests στο νέο branch;» αλλά «μπορεί κάθε ενεργό version να διαβάσει ό,τι γράφει το άλλο;».

Expand και contract

Για rename column, μην κάνεις rename και code switch στο ίδιο deploy. Πρόσθεσε νέο nullable column. Κάνε το νέο release να γράφει και τα δύο ή να γράφει σε μορφή που καταλαβαίνουν και οι δύο versions. Backfill σε batches. Άλλαξε reads στο νέο column. Σταμάτα dual write και αφαίρεσε το παλιό column σε μεταγενέστερο release αφού δεν υπάρχει rollback που το χρειάζεται.

```ruby
class AddNormalizedEmailToReservations < ActiveRecord::Migration[8.1]
  def change
    add_column :reservations, :normalized_email, :string
  end
end
```

Η πρώτη migration είναι additive και γρήγορη στις περισσότερες σύγχρονες databases, αλλά κάθε DDL θέλει adapter και table-size review. Το backfill δεν ανήκει υποχρεωτικά στη migration. Μεγάλο update χρειάζεται throttle, progress και stop control.

Αφού όλα τα rows έχουν τιμή και όλα τα writers τη γράφουν, πρόσθεσε constraint με ασφαλή adapter-specific σειρά. Στον PostgreSQL μπορείς να χρησιμοποιήσεις validated check strategy ή staged validation για μεγάλο table. Μην αντιγράψεις DDL recipe χωρίς να ελέγξεις ακριβή database version.

Defaults χωρίς table rewrite surprises

Database defaults προστατεύουν writers που δεν περνούν value. Model defaults βελτιώνουν new object behavior. Χρειάζεσαι και τα δύο όταν η invariant ισχύει για κάθε writer. Η προσθήκη default και `NOT NULL` σε υπάρχον μεγάλο table έχει διαφορετικό locking/rewrite behavior ανά database version. Μέτρα σε production-like copy και βάλε lock timeout.

Migration process δεν πρέπει να περιμένει για πάντα πίσω από long transaction. Αν DDL αποτύχει γρήγορα, το deploy μπορεί να σταματήσει καθαρά αντί να παγώσει writes. Το ακριβές `lock_timeout` είναι operational επιλογή και όχι γενικό Rails default.

Indexes σε ζωντανό table

Στον PostgreSQL χρησιμοποίησε concurrent index build όταν πρέπει να διατηρήσεις writes, με `disable_ddl_transaction!`. Μετά έλεγξε valid state και query plan. Ένα migration file που ολοκληρώθηκε δεν αποδεικνύει ότι ο planner χρησιμοποιεί τον index για τις πραγματικές bind values.

Unique constraint σε παλιά data χρειάζεται duplicate cleanup πριν δημιουργηθεί. Αν το cleanup τρέχει ενώ writers συνεχίζουν, βάλε πρώτα mechanism που εμποδίζει νέα duplicates ή κάνε retry μέχρι η unique build να κερδίσει καθαρό dataset.

Rails upgrade ως σειρά μικρών diffs

Αν η εφαρμογή απέχει πολλές releases, αναβάθμισε μία minor σειρά κάθε φορά. Το Rails upgrade guide προτείνει πρώτα καλή test coverage και μετά διαδοχικά βήματα. Τρέξε `bin/rails app:update` σε καθαρό branch και κάνε review κάθε config diff. Μην δεχτείς όλα τα generated defaults μηχανικά.

```
TERMINAL

bundle update rails
bin/rails app:update
bin/rails zeitwerk:check
bin/rails test
```

Το `config.load_defaults` κρατά παλιά defaults μέχρι να τα ενεργοποιήσεις. Το generated `new_framework_defaults_X_Y.rb` επιτρέπει να αλλάξεις behavior ένα setting κάθε φορά, με test και deploy ανά ομάδα. Όταν όλα έχουν υιοθετηθεί, ενημερώνεις `config.load_defaults` και αφαιρείς το προσωρινό initializer.

Διάβασε deprecations σε production-like traffic πριν την επόμενη upgrade. Ένα method που δεν εμφανίζεται στα tests μπορεί να καλείται μόνο από σπάνιο job. Στο Rails 8.1 υπάρχουν αλλαγές όπως deprecation για order-dependent finder χωρίς explicit order. Το fix πρέπει να εκφράζει πραγματικό order, όχι να προσθέσει τυχαίο `order(:id)` για να σιωπήσει warning.

Job compatibility

Queue μπορεί να κρατά serialized job από προηγούμενο release. Αν αλλάξεις class name ή arguments, το νέο worker πρέπει να διαβάζει τα παλιά jobs μέχρι να αδειάσουν ή να γίνει migration. Προτίμησε μικρά scalar arguments και tolerant keyword defaults.

```ruby
RUBY

class DeliverReservationJob < ApplicationJob
  def perform(reservation_id, template: "confirmation_v1")
    reservation = Reservation.find(reservation_id)
    ReservationDelivery.call(reservation: reservation, template: template)
  end
end
```

Αν μετονομάσεις job class, κράτα compatibility wrapper για το retention window της queue. Μην αφαιρέσεις model constant που βρίσκεται σε GlobalID payload πριν χειριστείς τα pending jobs.

Assets, cache και realtime compatibility

Fingerprint assets επιτρέπουν παλιά HTML να ζητήσει παλιό JS ή CSS όσο το asset παραμένει διαθέσιμο. Μην καθαρίζεις προηγούμενα assets πριν λήξουν cached pages. Turbo και Cable message schema πρέπει να είναι ανεκτό και από τους δύο client versions που υπάρχουν στα ανοιχτά tabs.

Cache keys με version prefix επιτρέπουν νέο release να μη διαβάζει incompatible serialized value. Προτίμησε primitive cache payloads. Marshal object από παλιά class definition είναι εύθραυστο σε rolling deploy και reload.

Health, readiness και graceful shutdown

Liveness απαντά αν το process ζει. Readiness απαντά αν μπορεί να δεχτεί traffic. Readiness δεν πρέπει να κάνει ακριβό query σε κάθε probe ούτε να δημιουργεί cascading restart επειδή ένα optional dependency έπεσε.

Στο shutdown, σταμάτα νέο traffic, άφησε in-flight requests και jobs να ολοκληρωθούν μέσα σε bounded grace period και μετά τερμάτισε. Σύγκρινε timeout του load balancer, container runtime, Puma και job supervisor. Το μικρότερο κρυφό timeout κερδίζει.

Canary, observation και rollback

Canary μειώνει blast radius μόνο αν συγκρίνεις σωστά signals. Error rate, p95, database load, queue latency και business invariant failures πριν και μετά. Traffic mix του canary πρέπει να περιλαμβάνει critical paths.

Rollback code δεν αναιρεί migration ή external effects. Αν B έγραψε value που A δεν διαβάζει, η επιστροφή στο A δεν είναι ασφαλής. Για αυτό η backward compatibility προηγείται του deploy mechanism.

Γράψε abort criteria πριν ξεκινήσει rollout. «Θα δούμε» δημιουργεί διαπραγμάτευση την ώρα του incident. Ένα release μπορεί να σταματήσει χωρίς rollback αν το schema είναι additive και τα παλιά instances παραμένουν healthy.

To release packet

Κάθε high-risk αλλαγή χρειάζεται σύντομο packet. Exact migration, compatibility matrix, expected locks, backfill command, dashboards, abort threshold, reconciliation query και cleanup release. Αυτό δεν είναι γραφειοκρατία. Είναι ο τρόπος να μην ανακαλύπτεις το plan ενώ η βάση περιμένει lock.

> **ΚΕΝΤΡΙΚΗ ΙΔΕΑ**
>
> Expand σε ένα release. Μετέφερε και επαλήθευσε data ανεξάρτητα. Contract μόνο όταν κανένα ενεργό ή rollback release δεν χρειάζεται το παλιό shape.

Οι επίσημες αναφορές είναι το Upgrading Ruby on Rails, το Rails 8.1 release guide και το Configuring Rails Applications.

> **ΕΛΕΓΧΟΣ ΣΤΗ ΔΙΚΗ ΣΟΥ ΕΦΑΡΜΟΓΗ**
>
> Πάρε την επόμενη schema αλλαγή. Γράψε matrix με παλιό code/new schema, νέο code/old schema, pending jobs και rollback. Αν κάποιο κελί δεν είναι συμβατό, σπάσε την αλλαγή σε expand, code switch, backfill, verification και contract πριν γίνει deploy.

ΠΑΡΑΡΤΗΜΑ

Πηγές και χάρτες απόφασης

Έκδοση αναφοράς

Το χειρόγραφο διασταυρώθηκε τον Σεπτέμβριο του 2026 με τα επίσημα Rails Guides για Rails 8.1.3.1. Η έκδοση 8.1.3.1 δημοσιεύτηκε στις 29 Ιουλίου 2026 σύμφωνα με το επίσημο Rails releases. Patch releases ασφαλείας μπορεί να αλλάξουν μετά την έκδοση αυτού του PDF. Έλεγξε ξανά τη maintenance policy πριν πάρεις version decision.

Το κείμενο είναι πρωτότυπη σύνθεση. Δεν αντιγράφει τα Rails Guides. Οι σύνδεσμοι είναι η authoritative συνέχεια για API details και αλλαγές έκδοσης.

Framework και execution

- Rails on Rack
- Action Controller Overview
- Autoloading and Reloading Constants
- Threading and Code Execution
- Configuring Rails Applications

Active Record

- Active Record Query Interface
- Active Record Associations
- Active Record Validations
- Active Record Callbacks
- Active Record Migrations
- Transactions API
- Pessimistic Locking API
- Optimistic Locking API
- Multiple Databases

Async work και delivery

- Active Job Basics
- Action Cable Overview
- Active Storage Overview
- Caching with Rails
- Working with JavaScript in Rails

Feedback, ασφάλεια και production

- Testing Rails Applications

- Active Support Instrumentation
- Error Reporting
- Securing Rails Applications
- Tuning Performance for Deployment
- Upgrading Ruby on Rails
- Rails 8.1 Release Notes

Ποιο εργαλείο για ποιο πρόβλημα

Πρόβλημα	Πρώτο εργαλείο	Τι παραμένει να ελέγξεις
Διπλό logical request	Unique idempotency key	External effect dedupe
Lost update	Optimistic lock ή atomic update	Conflict UX και retry
Oversell	Row lock ή conditional update	Database invariant
N+1	Preload ή query redesign	Rows και allocations
Χαμένο job intent	Transactional outbox	Dispatcher reconciliation
Διπλό job effect	Stable provider key	Crash window μετά το effect
Stale page μετά write	Writer read ή version	Replica lag budget
Αργό endpoint	Breakdown και plan	Queue time και saturation
Risky schema change	Expand-contract	Release overlap και rollback

Το production review σε δώδεκα ερωτήσεις

1. Ποια invariant πρέπει να ισχύει ακόμα και για δεύτερο writer;
2. Ποιο database constraint την κρατά;
3. Πού αρχίζει και πού τελειώνει η transaction;
4. Ποιο external effect δεν μπορεί να κάνει rollback;
5. Τι γίνεται αν το ίδιο job τρέξει ξανά;
6. Ποιο read μπορεί να ανεχτεί replica lag;
7. Πόσα rows και objects δημιουργεί το πιο συχνό query;
8. Ποιο cache key περιλαμβάνει όλες τις dimensions του output;
9. Ποιο signal δείχνει queue wait, lock wait και provider wait ξεχωριστά;
10. Ποιο test χρησιμοποιεί τον ίδιο database adapter με production;
11. Μπορεί το προηγούμενο release να διαβάσει το νέο data shape;
12. Ποιο query αποδεικνύει ότι το backfill ή το delivery ολοκληρώθηκε;

Αν μια ομάδα μπορεί να απαντήσει αυτές τις ερωτήσεις με links σε κώδικα, schema, tests και dashboards, τα advanced Rails concepts έχουν ήδη γίνει καθημερινή πρακτική.