全問正解率3%: RubyKaigiで出題したやりがちな危険コード5選

Kaigi on Rails 2025, 2025.09.26 (Fri.)@JP TOWER Hall & Conference Hall Red

Hubble,Inc Backend Tech Lead

Yuta Nakashima

Partner with RubyStackNews

Independent Ruby & Rails publication for senior developers

Why RubyStackNews?

- Focused on Ruby and Ruby on Rails
- Long-form articles based on real conference talks
- Audience of senior developers and tech leads
- Readers from the US, Europe, and Asia

RubyStackNews turns conference talks and real-world experience into practical, production-focused technical articles.

Partnerships & Sponsorships

- Article sponsorships
- Inline placements inside articles
- Sidebar visibility

View partnership details

partner-with-rubystacknews

rubystacknews.com

Partnership info

自己紹介

ta Nakashima

bble,Inc Backend Tech Lead

ンジニア歴6年、Rails歴4年半

bbleにCTO以来の一人目バックエンドエンジニアとして入

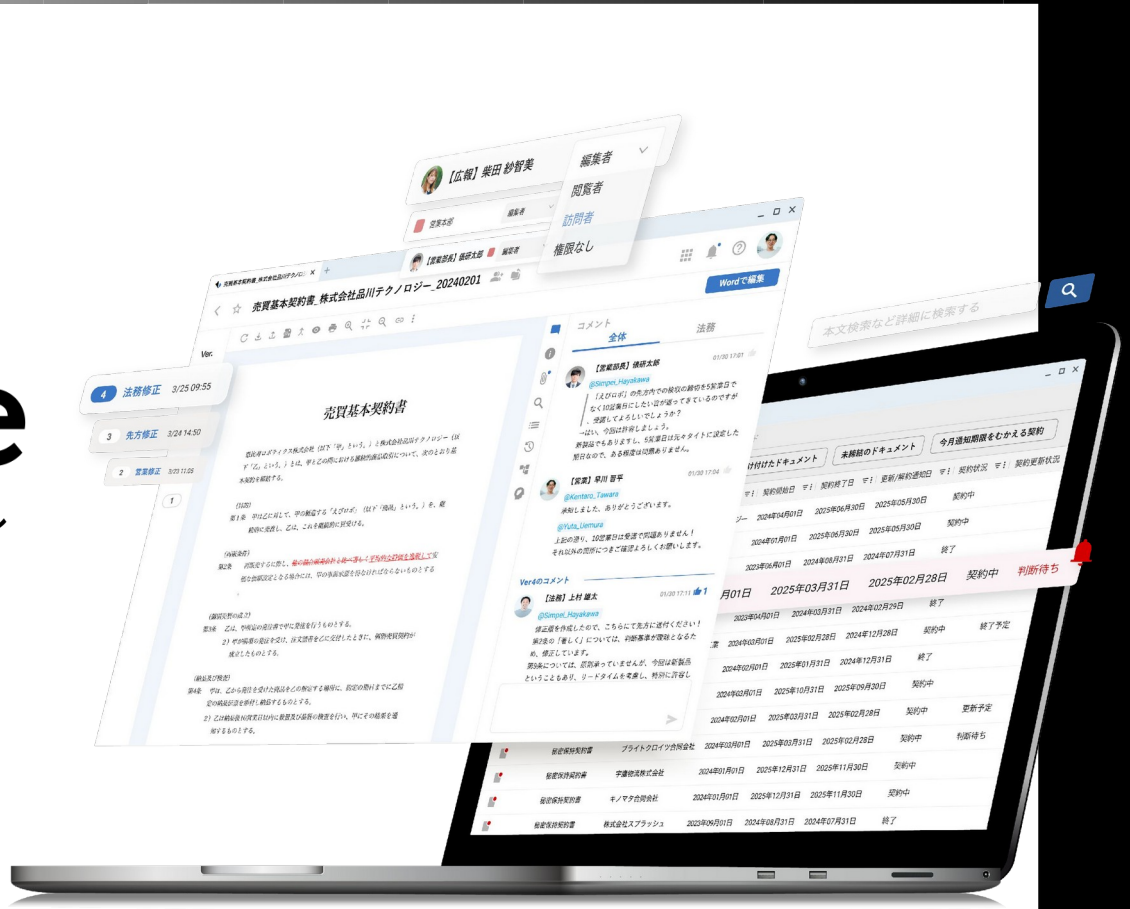
味: メタルラウド系のライブでよくモッシュしてます

AI契約業務・管理クラウド

Hubble

ハブル

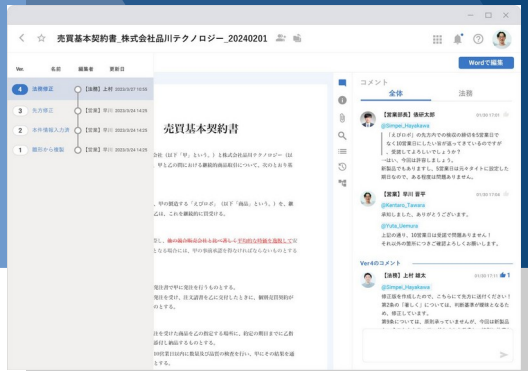
契約書の作成、社内のやりとり、
検討過程や合意文書を一元管
理。



契約業務の

オールインワンプラットフォーム

発生・起案　審査・交渉　押印申請・捺印　保管・管理　更新



前提

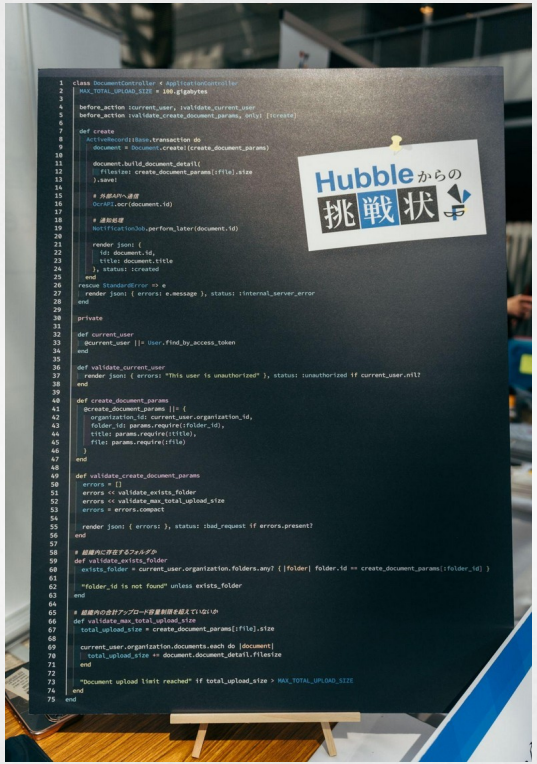
RubyKaigi 2025

初参加

初スポンサー

初ブース出展

前提



ブースにてコード問題を出題

全5問で正解数に応じて豪華景品を配布

Hubble

前提

結果、3日間で100名程度の方が参加

5問全問正解が 3名

3問以上正解が約 15名

回答しづらい環境ではありましたが、想像以上に難問だった可能性...🤔

Hubble

前提

もしかしてKaigi on Railsで内容・解答を話せば、役立つのでは...?

→ この発表になります!

Hubble

本題

今回話すことは所謂、 教科書的な話

- Transaction の範囲
- 非同期処理発火のタイミング
- エラーハンドリング
- SQL 側の責務とRuby側の責務の切り分け、テーブル設計

実際のプロダクトコードになると気づかずに書いてしまっていることや、レビューでも気づかないことが往々にしてある

出題形式

問題文 + 問題のあるコード

(問題文)

- バグ要因、パフォーマンス、セキュリティ観点で何箇所か直したほう良いところがあるので、どう修正すべき(例:logをtransaction外に出す等)を個数と行数を含めて箇条書きで回答してください。

- 問題の都合上、Modelバリデーションではなく、Controller側でバリデーションしていること、Controllerにビジネスロジックを書いていること、外部APIコールのレスポンス制御部分は対象外です。

```ruby
class DocumentsController < ApplicationController
  MAX_TOTAL_UPLOAD_SIZE = 100.gigabytes

  before_action :current_user, :validate_current_user
  before_action :validate_create_document_params, only: [:create]

  def create
    ActiveRecord::Base.transaction do
      document = Document.create!(create_document_params)

      document.build_document_detail(
        filesize: create_document_params[:file].size
      ).save!

      # 外部APIへ通信
      OcrAPI.ocr(document.id)

      # 通知処理
      NotificationJob.perform_later(document.id)

      render json: {
        id: document.id,
        title: document.title
      }, status: :created
    end
  rescue StandardError => e
    render json: { errors: e.message }, status: :internal_server_error
  end

  private

  def current_user
    @current_user ||= User.find_by_access_token
  end

  def validate_current_user
    render json: { errors: "This user is unauthorized" }, status: :unauthorized if current_user.nil?
  end

  def create_document_params
    @create_document_params ||= {
      organization_id: current_user.organization_id,
      folder_id: params.require(:folder_id),
      title: params.require(:title),
      file: params.require(:file)
    }
  end

  def validate_create_document_params
    errors = []
    errors << validate_exists_folder
    errors << validate_max_total_upload_size
    errors = errors.compact

    render json: { errors: }, status: :bad_request if errors.present?
  end

  # 組織内に存在するフォルダか
  def validate_exists_folder
    exists_folder = current_user.organization.folders.any? { |folder| folder.id == create_document_params[:folder_id] }

    "folder_id is not found" unless exists_folder
  end

  # 組織内の合計アップロード容量制限を超えていないか
  def validate_max_total_upload_size
    total_upload_size = create_document_params[:file].size

    current_user.organization.documents.each do |document|
      total_upload_size += document.document_detail.filesize
    end

    "Document upload limit reached" if total_upload_size > MAX_TOTAL_UPLOAD_SIZE
  end
end
```

Hubbleを模した

ドキュメントアップロードAPIが題材

```ruby
class DocumentsController < ApplicationController
  MAX_TOTAL_UPLOAD_SIZE = 100.gigabytes

  before_action :current_user, :validate_current_user
  before_action :validate_create_document_params, only: [:create]

  def create
    ActiveRecord::Base.transaction do
      document = Document.create!(create_document_params)

      document.build_document_detail(
        filesize: create_document_params[:file].size
      ).save!

      # 外部APIへ通信
      OcrAPI.ocr(document.id)

      # 通知処理
      NotificationJob.perform_later(document.id)

      render json: {
        id: document.id,
        title: document.title
      }, status: :created
    end
  rescue StandardError => e
    render json: { errors: e.message }, status: :internal_server_error
  end
```

```ruby
class DocumentsController < ApplicationController
  MAX_TOTAL_UPLOAD_SIZE = 100.gigabytes

  before_action :current_user, :validate_current_user
  before_action :validate_create_document_params, only: [:create]

  def create
    ActiveRecord::Base.transaction do
      document = Document.create!(create_document_params)

      document.build_document_detail(
        filesize: create_document_params[:file].size
      ).save!

      # 外部APIへ通信
      OcrAPI.ocr(document.id)

      # 通知処理
      NotificationJob.perform_later(document.id)

      render json: {
        id: document.id,
        title: document.title
      }, status: :created
    end
  rescue StandardError => e
    render json: { errors: e.message }, status: :internal_server_error
  end
```

Hubble

```
30  private
31
32  def current_user
33    @current_user ||= User.find_by_access_token
34  end
35
36  def validate_current_user
37    render json: { errors: "This user is unauthorized" }, status: :unauthorized if current_user.nil?
38  end
```

- アクセストークンからユーザを取得
- ユーザが取得できなければ、401エラー

問題なさそう！

```ruby
class DocumentsController < ApplicationController
  MAX_TOTAL_UPLOAD_SIZE = 100.gigabytes

  before_action :current_user, :validate_current_user
  before_action :validate_create_document_params, only: [:create]

  def create
    ActiveRecord::Base.transaction do
      document = Document.create!(create_document_params)

      document.build_document_detail(
        filesize: create_document_params[:file].size
      ).save!

      # 外部APIへ通信
      OcrAPI.ocr(document.id)

      # 通知処理
      NotificationJob.perform_later(document.id)

      render json: {
        id: document.id,
        title: document.title
      }, status: :created
    end
  rescue StandardError => e
    render json: { errors: e.message }, status: :internal_server_error
  end
```

```ruby
def validate_create_document_params
  errors = []
  errors << validate_exists_folder
  errors << validate_max_total_upload_size
  errors = errors.compact

  render json: { errors: }, status: :bad_request if errors.present?
end
```

- バリデーションエラーの配列を作成し、エラーを格納
- 空配列でなければ、400エラー

問題なさそう！

Hubble

```
49  def validate_create_document_params
50    errors = []
51    errors << validate_exists_folder
52    errors << validate_max_total_upload_size
53    errors = errors.compact
54
55    render json: { errors: }, status: :bad_request if errors.present?
56  end
```

Hubble

```ruby
# 組織内に存在するフォルダか
def validate_exists_folder
  exists_folder = current_user.organization.folders.any? { |folder| folder.id == create_document_params[:folder_id] }

  "folder_id is not found" unless exists_folder
end
```

- ユーザの組織からフォルダを全て取ってきてfolder_idと一致するかをany?で評価

問題点1: SQLで全件取得

Hubble

```
58  # 組織内に存在するフォルダか
59  def validate_exists_folder
60    exists_folder = current_user.organization.folders.any? { |folder| folder.id == create_document_params[:folder_id] }
61
62    "folder_id is not found" unless exists_folder
63  end
```

- **大量のオブジェクト生成**：N個のフォルダがある場合、N個のActiveRecordオブジェクトがメモリ上に生成される
- **メモリリーク**：不要なデータが大量にメモリに保持されGCの負荷が増大
- **ネットワーク帯域の無駄遣い**：DBサーバとAPIサーバ間の不要な通信量増大（**コネクションプール**を占有してAPIにも影響する可能性）
- **DBサーバのCPU負荷**：不要なデータ(id以外のカラム)をシリアライズして送信する処理コスト
- **レスポンス時間の劣化**：フォルダ数に比例してレスポンスが遅くなる(最悪計算量 O(N))
- **スケーラビリティの欠如**：データ量増加に対して線形的にパフォーマンスが悪化（エンタープライズだと、数万数十万は全然有り得る話）

解決方法1: exists?でSQLで存在確認

```ruby
58  # 組織内に存在するフォルダか
59  def validate_exists_folder
60    exists_folder = current_user.organization.folders.exists?(id: create_document_params[:folder_id])
61
62    "folder_id is not found" unless exists_folder
63  end
```

- ActiveRecordからBooleanに: exists?の返り値がTrueClass or FalseClassの1オブジェクトだけになるので大幅削減
- ネットワーク帯域を圧迫しない: DBサーバとAPIサーバ間の不要な通信量が最小
- DBサーバのCPU負荷が最小: 不要なカラムのデータを取得しない
  - 発行クエリ
  - 今回だと約8 `SELECT 1 AS one FROM folders WHERE folders.id = 1 LIMIT`
- レスポンス時間とスケーラビリティ: フォルダ数に関係なくインデックス(今回であればidでprimary key)があればO(logN)

Hubble

```ruby
def validate_create_document_params
  errors = []
  errors << validate_exists_folder
  errors << validate_max_total_upload_size
  errors = errors.compact

  render json: { errors: }, status: :bad_request if errors.present?
end
```

```ruby
# 組織内の合計アップロード容量制限を超えていないか
def validate_max_total_upload_size
  total_upload_size = create_document_params[:file].size

  current_user.organization.documents.each do |document|
    total_upload_size += document.document_detail.filesize
  end

  "Document upload limit reached" if total_upload_size > MAX_TOTAL_UPLOAD_SIZE
end
```

- ユーザの組織からドキュメントを全て取ってきてる
- ドキュメントのリレーション先のドキュメント詳細を取得してる

問題点2: SQLクエリではなくRuby側で計算

Hubble

```ruby
# 組織内の合計アップロード容量制限を超えていないか
def validate_max_total_upload_size
  total_upload_size = create_document_params[:file].size

  current_user.organization.documents.each do |document|
    total_upload_size += document.document_detail.filesize
  end

  "Document upload limit reached" if total_upload_size > MAX_TOTAL_UPLOAD_SIZE
end
```

- **1 + N回のクエリ実行** 1,000件のドキュメントがあれば1,001回のクエリが発行される（初回のdocuments取得 + 各documentごとのdocument_detail取得）
- **コネクションプール枯渇** 大量の小さなクエリでデータベース接続が長時間占有され、他のリクエストが接続待ちになる可能性 (Railsは1リクエスト1コネクション)
- **DBサーバCPU負荷:** 同じような小さなクエリを大量に処理する非効率性と **クエリパーシング** の繰り返し
  - クエリパーシング(字句解析)はSQL実行のおよそ40%を占める
  - 同一クエリでもパラメータが違えば別クエリとして扱われる
  - キャッシュが効かずパーシングが繰り返される
- **ラウンドトリップ回数増大** アプリケーションサーバとDBサーバ間の通信がN回発生し、ネットワークレイテンシが積に累積(一般に同一リージョン通信は0.5ms - 2.0ms、クエリ実行自体は0.1ms - 2.0ms)

解決方法2: sumでSQLで計算

Hubble

```ruby
# 組織内の合計アップロード容量制限を超えていないか
def validate_max_total_upload_size
  total_upload_size = create_document_params[:file].size

  total_upload_size += current_user.organization.documents
                                   .joins(:document_detail)
                                   .sum("document_details.filesize")

  "Document upload limit reached" if total_upload_size > MAX_TOTAL_UPLOAD_SIZE
end
```

- **単一クエリ実行**: 1,001回 → 1回のクエリで処理完了
- **SQL集約関数活用**：データベースエンジンの最適化されたSUM処理を利用
- **サーバCPU負荷軽減**: Ruby側での繰り返し処理が不要
- **ネットワーク通信最小化** 1回の通信で処理完了

(別解) organizationテーブルにtotal_filesizeカラムを持つ

- **ドキュメントとの分離** ドキュメント数が増えても影響なし、計算量はorganizationのO(logN)

(別解) KVSキャッシュ + 更新時検証

- **EC在庫管理等** によくあるパターン

```ruby
def create
  ActiveRecord::Base.transaction do
    document = Document.create!(create_document_params)

    document.build_document_detail(
      filesize: create_document_params[:file].size
    ).save!

    # 外部APIへ通信
    OcrAPI.ocr(document.id)

    # 通知処理
    NotificationJob.perform_later(document.id)

    render json: {
      id: document.id,
      title: document.title
    }, status: :created
  end
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

```ruby
def create
  ActiveRecord::Base.transaction do
    document = Document.create!(create_document_params)

    document.build_document_detail(
      filesize: create_document_params[:file].size
    ).save!

    # 外部APIへ通信
    OcrAPI.ocr(document.id)

    # 通知処理
    NotificationJob.perform_later(document.id)

    render json: {
      id: document.id,
      title: document.title
    }, status: :created
  end
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

問題点3: トランザクション内での外部API実行

Hubble

```ruby
def create
  ActiveRecord::Base.transaction do
    # 外部APIへ通信
    OcrAPI.ocr(document.id)
  end
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

- **参照ロック長時間保持** 外部キー制約により参照先テーブル（Organization）の行が外部APIのresponseが終わるまで共有ロック状態
- **排他ロック待ち発生** 同一Organization行の更新・削除操作が全てブロックされる
- **問題切り分けが難化:** ロジックが問題なのか、外部API障害なのかの切り分けが難しくなる

解決方法3: 外部API実行をトランザクション外に

Hubble

```ruby
def create
  document = ActiveRecord::Base.transaction do
    Document.create!(create_document_params)
  end
  # 外部APIへ通信
  OcrAPI.ocr(document.id)
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

- 通常のTransactionと同じロック: 長時間参照先テーブルをロックしない
- 問題切り分けが容易: ロジックが問題なのか、外部PI障害なのかの切り分けがわかりやすい

```ruby
def create
  ActiveRecord::Base.transaction do
    document = Document.create!(create_document_params)

    document.build_document_detail(
      filesize: create_document_params[:file].size
    ).save!

    # 外部APIへ通信
    OcrAPI.ocr(document.id)

    # 通知処理
    NotificationJob.perform_later(document.id)

    render json: {
      id: document.id,
      title: document.title
    }, status: :created
  end
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

問題点4: トランザクション内での非同期処理発火

Hubble

```ruby
def create
  ActiveRecord::Base.transaction do
    # 通知処理
    NotificationJob.perform_later(document.id)
  end
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

- **Race Condition発生:** Job実行タイミングによってはコミット前のデータにアクセス
  例外発生
- **ロールバック時の不整合** トランザクションロールバック後も非同期ジョブがキュー
  存して不要実行
- **不整合データ生成:** ロールバックしているのに他モデルのデータを作ってしまう可能
- **エラー状態の不明確性:** トランザクション失敗とジョブ失敗の区別が困難

解決方法4: 非同期処理発火をトランザクション外に

Hubble

```ruby
def create
  document = ActiveRecord::Base.transaction do
    Document.create!(create_document_params)
  end
  # 通知処理
  NotificationJob.perform_later(document.id)
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

- **DBの整合性:** 未コミットデータアクセス問題の解消
- **システム安定性の改善:** Race Conditionやデッドロックの回避
- **エラー処理の明確化:** 問題の切り分けとデバッグ効率化
- **開発・保守性向上** テスト容易性とコードの責務分離

3.8 ジョブのトランザクション整合性

Solid Queueは、デフォルトではメインアプリケーションとは別のデータベースを利用します。これにより、トランザクションの整合性に関する問題が回避され、トランザクションがコミットされた場合にのみジョブがエンキューされるようになります。

ただし、Solid Queueをアプリと同一のデータベースで利用する場合は、Active Jobの `enqueue_after_transaction_commit` オプションでトランザクションの整合性を有効にできます。このオプションは、ジョブごとに有効にすることも、以下のように `ApplicationJob` ですべてのジョブに対して有効にすることも可能です。

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

また、Solid Queueジョブ用のデータベースコネクションを別途設定することで、トランザクションの整合性の問題を回避しながら、アプリと同一のデータベースを利用するようにSolid Queueを構成することも可能です。

Hubble

14:10 ～ 14:25		Hall Red 非同期jobをtransaction内で呼ぶなよ！絶対に呼ぶなよ！ Yuto Urushima

より詳細なことは明日の14:10からの発表で!

```ruby
def create
  ActiveRecord::Base.transaction do
    document = Document.create!(create_document_params)

    document.build_document_detail(
      filesize: create_document_params[:file].size
    ).save!

    # 外部APIへ通信
    OcrAPI.ocr(document.id)

    # 通知処理
    NotificationJob.perform_later(document.id)

    render json: {
      id: document.id,
      title: document.title
    }, status: :created
  end
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

```ruby
def create
  ActiveRecord::Base.transaction do
    document = Document.create!(create_document_params)

    document.build_document_detail(
      filesize: create_document_params[:file].size
    ).save!

    # 外部APIへ通信
    OcrAPI.ocr(document.id)

    # 通知処理
    NotificationJob.perform_later(document.id)

    render json: {
      id: document.id,
      title: document.title
    }, status: :created
  end
rescue StandardError => e
  render json: { errors: e.message }, status: :internal_server_error
end
```

問題点5: StandardErrorで全例外を処理

Hubble

```
26  rescue StandardError => e
27    render json: { errors: e.message }, status: :internal_server_error
28  end
29
```

- **内部実装詳細の漏洩**：内部エラー時にDB構造、テーブル名、カラム名、ファイルパス、環境変数等が露出
- **SQLインジェクション**：DBエラーで内部クエリ構造が露出
- **HTTPステータスコードの固定化**：全てのエラーが500番になり適切でない(この場合だとバリデーションエラーも500になる)
- **ユーザビリティ低下**：技術的なエラーメッセージでユーザーが混乱(ユーザーが次に何をすべきかわからない)
- **テスト品質低下**：エラー条件のテストが不十分になる

解決方法5: エラー分岐、ステータスコード整理

Hubble

```ruby
rescue ActiveRecord::RecordInvalid => e
  render json: { errors: e.record.errors.full_messages }, status: :bad_request
rescue OcrAPI::TimeoutError => e
  Rails.logger.error("OCR timeout: #{e.message} backtrace: #{e.backtrace}")
  render json: {
    error: "外部サービスとの処理がタイムアウトしました。しばらく時間をおいて再試行してください"
  }, status: :gateway_timeout
rescue OcrAPI::InvalidResponse, OcrAPI::ServiceError => e
  Rails.logger.error("OCR service error: #{e.message} backtrace: #{e.backtrace}")
  render json: {
    error: "外部サービスとの通信でエラーが発生しました"
  }, status: :bad_gateway
rescue StandardError => e
  Rails.logger.error("Unexpected error: #{e.message} backtrace: #{e.backtrace}")
  render json: {
    error: "予期しないエラーが発生しました"
  }, status: :internal_server_error
end
```

- **内部情報漏洩防止：**技術的詳細を隠蔽し、攻撃者に有用な情報を提供しない
- **クライアント側エラー判別しやすさ**バリデーションエラー（400）と外部サービスエラー（502/504）の明確な区別
- **具体的なエラーケース検証** 各例外（TimeoutError、ServiceError等）を個別にテスト可能

まとめ

- 意外とちゃんと 意識しないといけないポイントが多い
  - Transactionの範囲
  - 非同期処理発火タイミング
  - エラーハンドリング
  - SQL側の責務とRuby側の責務の切り分け、テーブル設計
- 知識的に知ってても プロダクトコード になると意識せずに書き
- 特に最近のAIによるVibe Codingでは意識していきたいところ

最後に

Hubbleでは様々な技術的課題(特にパフォーマンスチューニング)に取り組んでいます!

- ドライブシステムの複雑な親子関係のRuby計算をSQLに置き換え
- アプリケーションの権限管理をSQLでのBit演算で計算
- AIエージェントの他機能への横展開and more...

そういったことに興味のある方はぜひ ブースまで!