TWOGATE

2025年06月27日

Railsアプリケーションとパフォーマンスチューニング

ー 秒間5万リクエストのモバイルオーダーシステムを支える事例 ー

株式会社TwoGate
取締役CTO 奥本 隼

TWOGATE

参考資料

Kaigi on Rails 2024 での発表資料では
具体的に立ち向かった課題に対してのアプローチを公開しています。

https://kaigionrails.org/2024/talks/falcon8823/

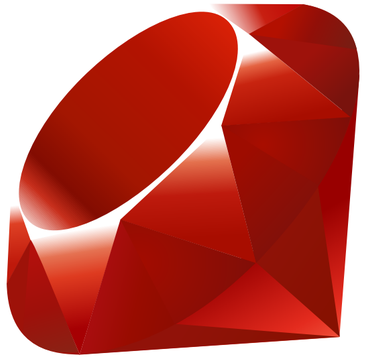
自己紹介

奥本 隼 (Hayato OKUMOTO)

- 1994年生まれ 31歳
- 株式会社TwoGate **取締役CTO**
  - チーム組成から12年 / 創業10期目
- 長野高専 出身
  - →豊橋技科大（学部）→同（修士）

TWOGATE

私とRuby

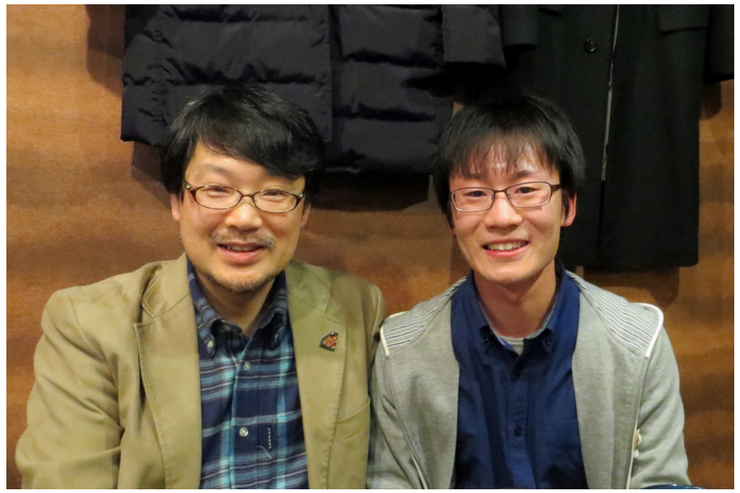


Rubyと出会うまで

17歳のとき

はじめてRubyを使う

20歳のとき



学校での講演会の
懇親会にて

30歳のとき

RubyKaigi 2024

TWOGATE

Rubyと出会うまで

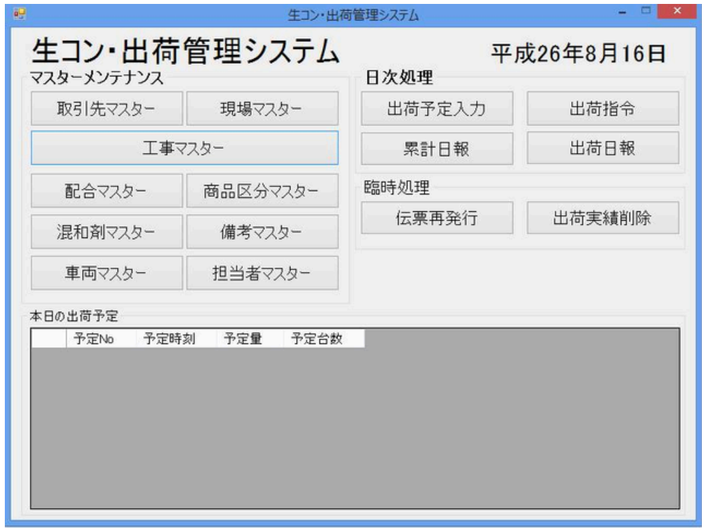
HTML → JavaScript → Perl(CGI)

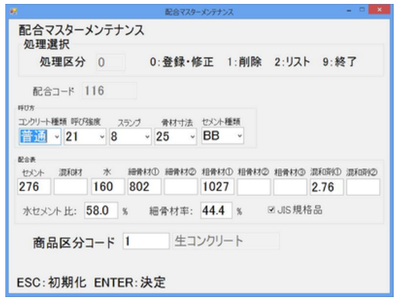
→ Linuxサーバ → VB.NET → C# → Ruby

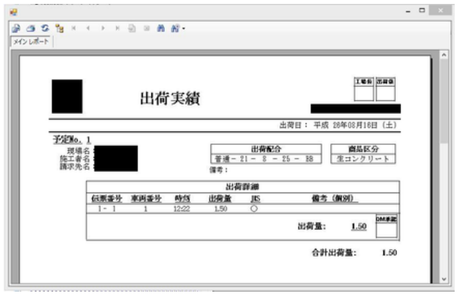
Rubyと出会うまで

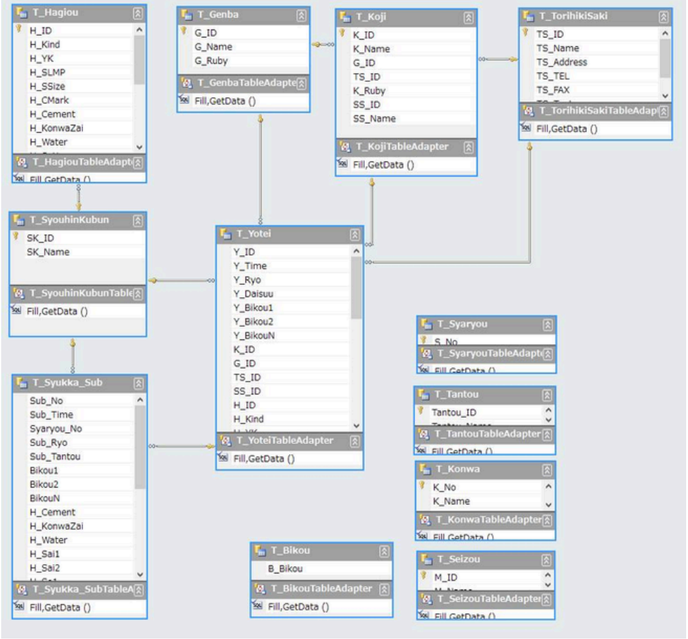
高専への入学前（2009年）

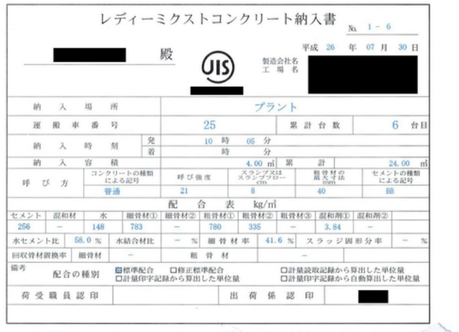
父親の会社にソフトウェアを納品して入学金を稼ぐ











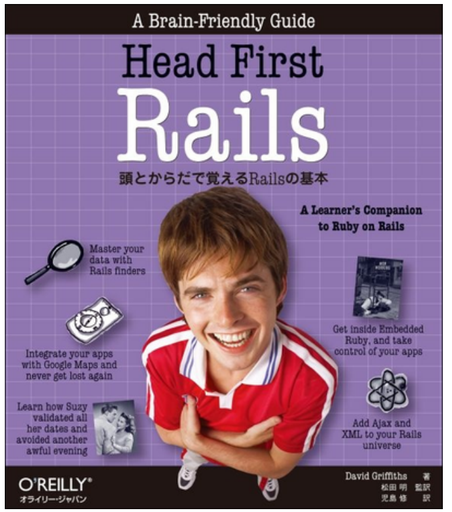
Rubyとの出会い - 2011年

4月

- 高専OB 「長野で高専カンファやるから、受付システムつくって」
- 当時の先輩「Rails流行ってるしRailsでいいんじゃね？」

5月

- GWにRailsを覚えながら合宿して一気にプロトタイプを実装
- Ruby / Rails / Gitを覚えた
- Rails 2系で開発



初めてRailsを最初に学んだ書籍

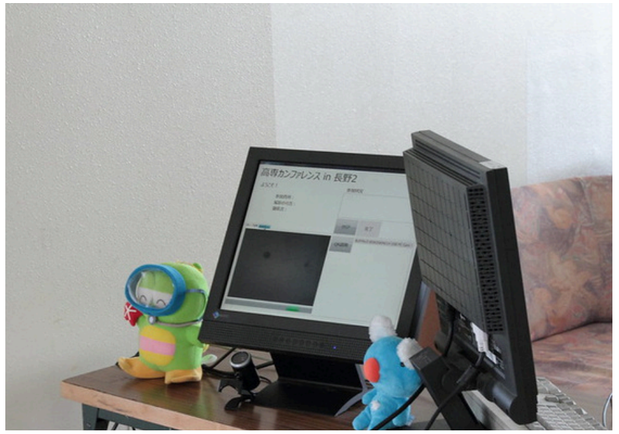
初めてのRubyアプリケーション

TWOGATE

「カンファイン」

高専カンファレンス向けのイベント受付システム

QRコードでチェックイン

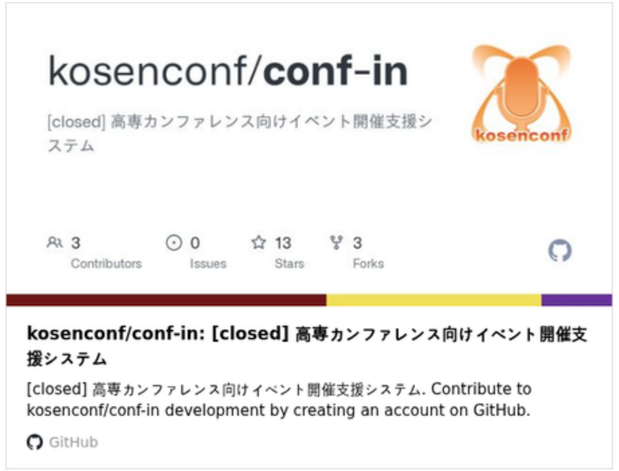


5月 開発開始

6/24 リリース/受付開始

7/16 イベント当日

120人の参加者を無事に受付



TWOGATE

Rubyでレッドブル1年分

そのまま高専プログラミングコンテストに応募

→本選で「さくらインターネット賞」を受賞

- Realforceキーボード
- レッドブル1年分

学生時代の仲間たちと - TwoGate設立

学生時代のプログラミング仲間たちを

率いてTwoGateを共同創業

Rubyに支えられています

TWOGATE

TwoGateの全てのプロダクトはRuby製

- 創業時から採用
- 国内トップクラスのトラフィックを捌いている

TWOGATE

Rubyコミュニティへの恩返し

Kaigi on Rails 2024 / RubyKaigi 2025

Rubyスポンサー

Kaigi on Rails 2024では
TwoGateから2名登壇

TWOGATE

TwoGate について

About TWOGATE

- 2016年創業
  - 受託開発から自社プロダクト開発に変遷
- 従業員42名
  - うち16名が高専出身の高専チーム
  - インターン＋リファラル中心での採用
- エンジニア中心の会社経営

TwoGateの主要なソリューション

音楽アーティストやYouTuber, タレントの
収益装置を提供している会社です。

Caravan
ライブイベント向け
モバイルオーダーアプリ

TRIPLE
ライブチケット販売サイト

CRAYON
ファンクラブアプリ

SlashGift
オンラインくじサイト

TWOGATE

Caravan - イベント物販に特化したアプリ

original design

アーティストごとのオリジナルデザイン

アーティスト・IPの世界観に
合わせたデザインに

シンプルでスムーズな会場受取

アプリで事前に商品を購入し、会場に行って受け取るだけ。時間枠も管理可能なので売り場の混雑状況もコントロール可

欲しい商品を選択

受け取りたい時間を選択

購入完了後、引取用QR表示

GPSで来場の認証後、QR表示

TWOGATE

技術スタック

サーバサイド

Built with

RAILS

フロントエンド

Angular

ionic

インフラ

aws

fastly

Firebase

Auth0 by Okta


TWOGATE
推し活
×
ハイトラフィック

ピークトラフィック

CDN

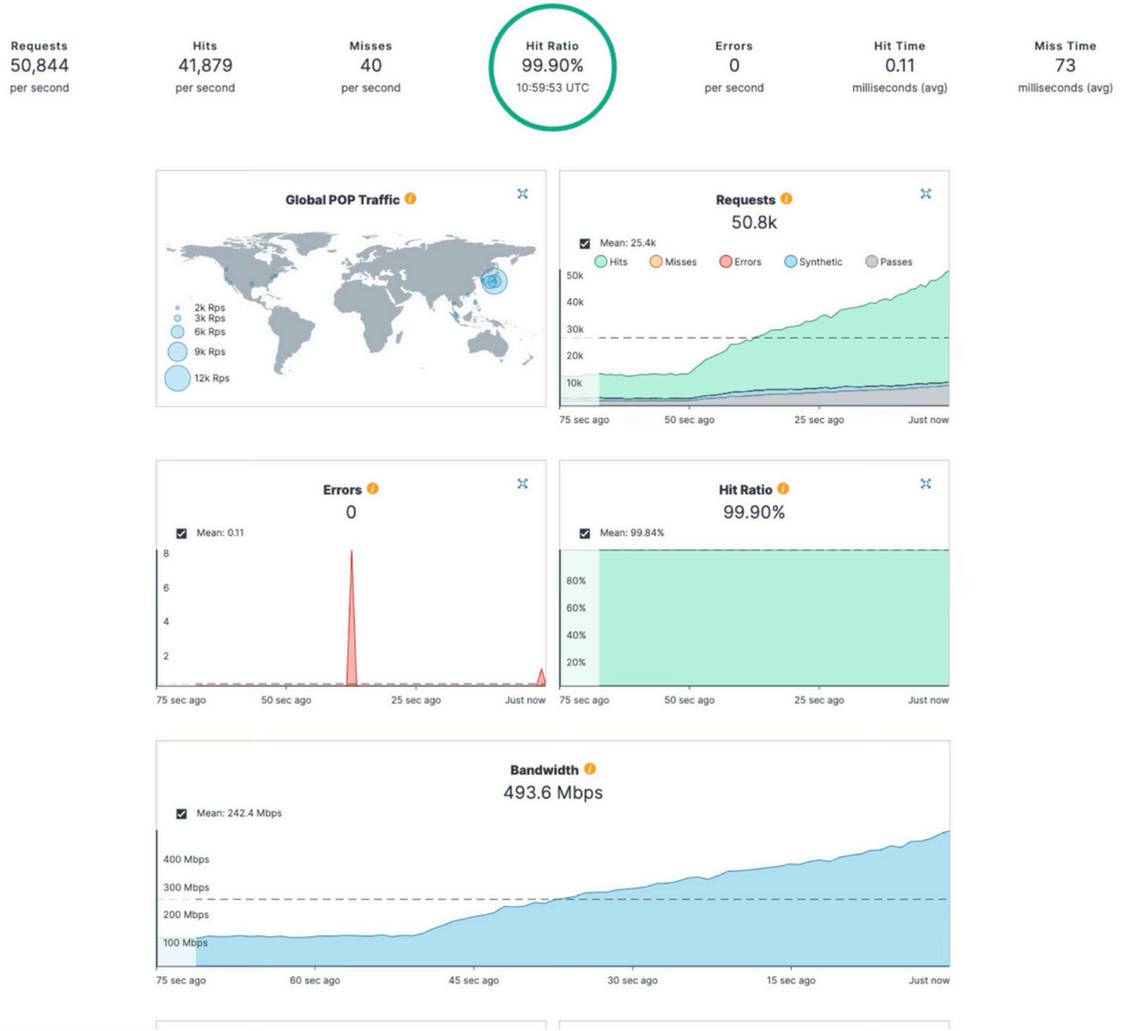
- 50,000 RPS

ALB / Rails

- 8,000 RPS

決済エンドポイント

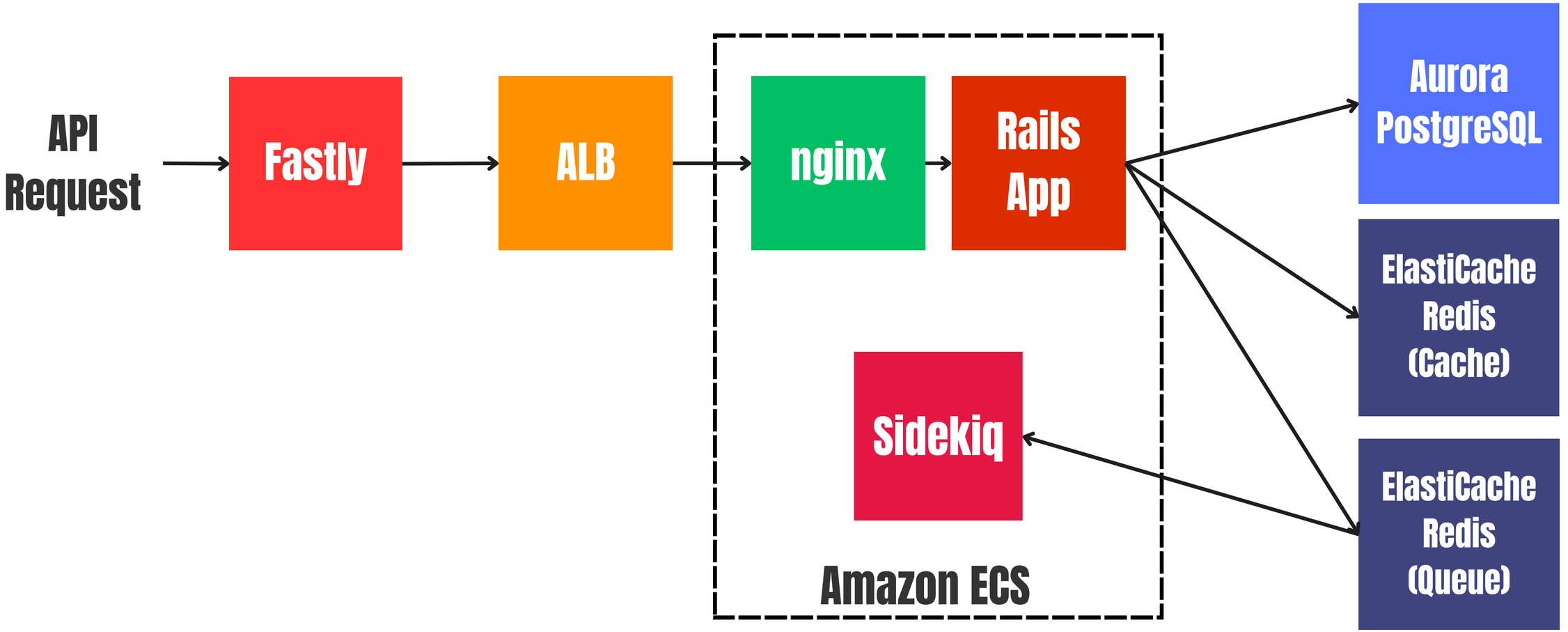
- 660 RPS



TWOGATE

インフラアーキテクチャ

実はシンプル＋モノリシックなRailsアプリ



TWOGATE

本題

Railsアプリケーションと
パフォーマンスチューニング

TWOGATE

本発表で扱うテーマ

パフォーマンスを意識したRailsアプリ開発の基礎

- 設計レベルで気をつけること
- 負荷試験
- パフォーマンスチューニング

TWOGATE

話すこと・話さないこと

- **話すこと**
  - パフォーマンスを意識した設計・心構え
  - APIベースのRailsアプリケーション
  - ちょっとしたテクニック
- **話さないこと**
  - Rails Viewベースのアプリケーション
  - Ruby自体のチューニングや高速化

TWOGATE

パフォーマンスを
意識した設計

パフォーマンスを意識した設計

- **基礎を積み上げていくことが基本**
  - N+1クエリ / スロークエリを避ける
  - 計算量・速いアルゴリズムの感覚
- **全てアプリケーションサーバでは解決しない**
  - Webはアーキテクチャの総合格闘技

TWOGATE

パフォーマンスを意識した設計

- **アプリケーション設計**
  - N+1が起きないクエリ設計・テーブル設計
  - パス・コントローラの設計
- **キャッシュ戦略**
  - CDN / アプリケーションキャッシュ

N+1クエリを防げ

- **N+1問題**
  - テーブルの関連先の行を読み出すときに、アプリケーションからN回のクエリを実行してしまう実装のこと
  - DBとの通信コストによるレイテンシーの増加
- **対策**：Bullet gemを入れてCIでも実行する

TWOGATE

CDN導入を前提とした設計

- **CDNによるキャッシュは特効薬**
  - 個人に寄らないAPIはキャッシュすることを検討する
  - Railsサーバへの負荷を効果的に減らせる
- **キャッシュによる情報漏えいに注意**
  - Cache-Control / Vary / ETagヘッダをよく理解する
  - プロバイダごとの挙動を理解する

Railsでの実装上の工夫

ありがちな認証処理を共通化するコード

CDNでキャッシュするエンドポイントで

current_userを呼び出していたら…

```ruby
class ApplicationController < ActionController::API
  before_action :authenticate_user!
  helper_method :current_user

  def authenticate_user!
    unless current_user
      render json: { error: "Unauthorized" }, status: :unauthorized
    end
  end

  def current_user
    @current_user ||= User.find_by(id: @user_id)
  end
end
```

Railsでの実装上の工夫

CDNキャッシュするパスとコントローラを分けておく

- 仮に間違えてcurrent_userを参照してもエラーになる
- CDN設定で/api/private/*はキャッシュしない設定にしておく

```ruby
# /api/public/...
module API::Public
  class ApplicationController < API::ApplicationController
  end
end

# /api/private/...
module API::Private
  class ApplicationController < API::ApplicationController
    before_action :authenticate_user!
    helper_method :current_user

    def authenticate_user!
      unless current_user
        render json: { error: "Unauthorized" }, status: :unauthorized
      end
    end

    def current_user
      @current_user ||= User.find_by(id: @user_id)
    end
  end
end
```

ガードレールを敷きやすい＋シンプルなルールを運用すること

ゆるいCDNキャッシュ

- 本気でCDNキャッシュを考え始めると大変
  - max-ageの設定 / 更新時のパージAPIを呼び出す…
- **stale-while-revalidate での逃げはオススメ**
  - キャッシュが切れても再検証されるまで古いキャッシュを返す
  - →短めに設定してもキャッシュミスが頻発しない

TWOGATE

CORSをCDNで応答する

- ブラウザ＋クロスドメインのケースで**Preflightリクエスト**が起きるケース
  - 細かいリクエストまでRailsで応答するのは無駄
  - CDNでのキャッシュや応答を検討する
  - →**劇的にオリジンリクエストを減らせた**

ユーザを効果的に待たせる

- どうしてもチューニングによって最適化できない領域がある
  - 例）外部の決済プロバイダのレートリミット
- 最終的に**ユーザ体験を損なわないようにユーザを待たせる**設計
  - **「サーバエラー」**→お問い合わせに走ってしまう
  - **「現在混み合っております。しばらく時間を空けて再度お試しください。」**→ユーザが何をすればよいか伝わる
- ユーザやクライアントの教育も重要

TWOGATE

負荷試験

負荷試験の基本

- 負荷を疑似的に再現してボトルネックを見つける作業
- 負荷試験ツールを使って負荷を掛ける
  - 単一型 / シナリオ型
- 負荷試験ツールの使い方については踏み込みません

シナリオ試験

- **実際のユーザの利用シナリオからリクエストをシミュレーションする**
  - ブラウザで実際の操作を記録してシミュレーションコードを生成する
  - HARファイルで記録したものからコード化できる
    - 例）Chromeのネットワークタブで保存できる
  - ここから認証回避など負荷試験固有な書き換えをする

計測せよ

- **パス単位での応答時間は最低限集計**できるように
  - ログのテキストからでもなんでもできればOK
- **APMはあった方がよい**
  - 高いので普段は入れなくても負荷試験の時だけでも
- **クラウドプロバイダの標準機能も活用**
  - AWSのRDS/AuroraならPerformance Insightsあれば勝てる

ちょっと待って

- クラウドプロバイダの**負荷試験のルールを確認**すること
  - AWS: 事前申請必要（一定の負荷以内であれば不要）
  - AWSアカウントを別にする必要がある
- 外部APIに負荷を掛けないように**モック化する**こと
- **HTTPSを切って**試験すること
  - 特に負荷を掛ける側のパフォーマンスが出ない
- **ロードバランサの暖気**に注意

試験結果の眺め方

- **全体のエラー率を確認**する
  - エラーが高ければ低い負荷でエラーが無くなるラインを探す
  - 目標スループットに対して**何倍くらい耐久が必要か**考える
- **エンドポイントごとのエラー率、応答時間を見る**
  - 経験則：特定のエンドポイントが**8-9割のボトルネック**

TWOGATE

パフォーマンス
チューニング

チューニングの悪魔

みなさん、速いの好きですよね？

なにをチューニングしたいのか

投下したチューニングコストと事業インパクトに照らし合わせよう

- **インフラの過負荷の回避**
  - サービスダウンによる機会損失・信用失墜の回避
- **ユーザ体験の最適化**
  - ユーザがより快適にサービスを利用できること
- **インフラコストの低減**
  - 事業規模に対して適性な原価となるように調整する
- **エンジニアの稼働を減らす**
  - 深夜対応、休日対応、障害対応によるコストを減らす

TWOGATE

どこからチューニングするか

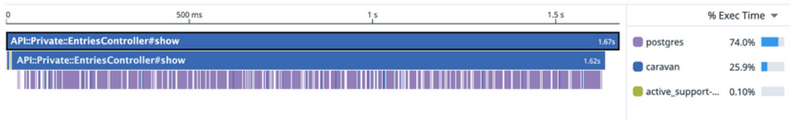
- **App / DB / Web / LB / CDNの順にチューニングする**
  - App以外は偉人たちの長年の経験が詰まっている
    - 遅いのはたいてい自分のせい
- **平均よりもレスポンスタイムが遅いエンドポイントから**
  - 100倍速くできそうなエンドポイント >1000ms
  - リクエスト数×レイテンシが高いエンドポイント
  - 2-10倍速くできそうなエンドポイント >100ms

原因を探る

- **読み込みのボトルネック**
  - クエリ・インデックスの最適化
  - リードレプリカによる負荷分散
- **書き込みのボトルネック**
  - トランザクション単位の見直し / Bulk Insertなど
  - 非同期化
- **外部APIのボトルネック**
  - 非同期化 / アプリケーションキャッシュ

クエリの最適化

- **N+1クエリがないか**
  - 対処は比較的容易
  - Railsならinclude, preload, eager_loadを掛ける
- **1つのクエリによるスロークエリか**
  - 問題を分解して確認する必要がある
  - インデックス / 実行計画を詳しく確認



GPTで実行計画を解析

このPostgreSQLの実行計画におけるボトルネックを見てみると、いくつかの重要なポイントがあります。特に以下の操作が注目に値します。

1. 高コストのネステッドループ操作

- 実行計画において、`Nested Loop`および`Index Scan`が頻繁に使用されており、これが特に高コストの操作です（コストが`0.42..235.76`で2072行を処理）。`Index Scan`の条件やフィルタが十分に効率的でない可能性があるため、これに関連するインデックスの構造やカーディナリティ（データの分布）を確認し、最適化を検討する価値があります。

2. ソート操作

- `Sort`操作（コスト`58.97..58.98`）は、行数が少ないにもかかわらず比較的高コストです。ソートが必要なフィールドにインデックスが存在しているか、またはそのインデックスが適切に利用されているかを確認してください。適切なインデックスがあれば、ソート操作のコストを削減できる可能性があります。

3. ハッシュアグリゲートおよびグループ集計

- `HashAggregate`（コスト`286.66..293.26`）および`GroupAggregate`（コスト`58.97..59.21`）もパフォーマンスに影響を与える可能性があります。これらの集計処理で扱うデータ量を減らすために、前段のフィルタリングを強化することが有効です。

改善提案

1. **インデックスの再検討**:
   - `gacha_check_outs`テーブルの`machine_id`および`paymentable_id`、`paymentable_type`などに対するインデックスが適切に機能しているか確認し、必要に応じてインデックスを追加または最適化してください。
2. **フィルタリングの強化**:

パラメータチューニング

- アプリケーションの性質によるので一概に言えない
- 目的は**計算資源を最大限に使い切る**こと

パラメータチューニング

- ロードバランサ
  - 負荷分散アルゴリズムによる偏り防止
- DB
  - 大抵いじらなくてよいがワークロードによって調整すると改善するパラメータもある
- Linux
  - ファイルディスクリプタ数 / TCP関連のパラメータ調整

TWOGATE

まとめ

まとめ

- **なにをチューニングする？どうしてチューニングする？**
  - ゴール設定を見失わないように
- **荒く効果的にできるところからチューニング**
  - 初めて負荷試験すると大抵はイージーな課題が多い
- **ユーザやクライアントを巻き込んだ体験設計**
  - パーツのチューニングが目的ではなく、事業価値、ユーザ体験、運用負荷の全方位を最適化することが大切