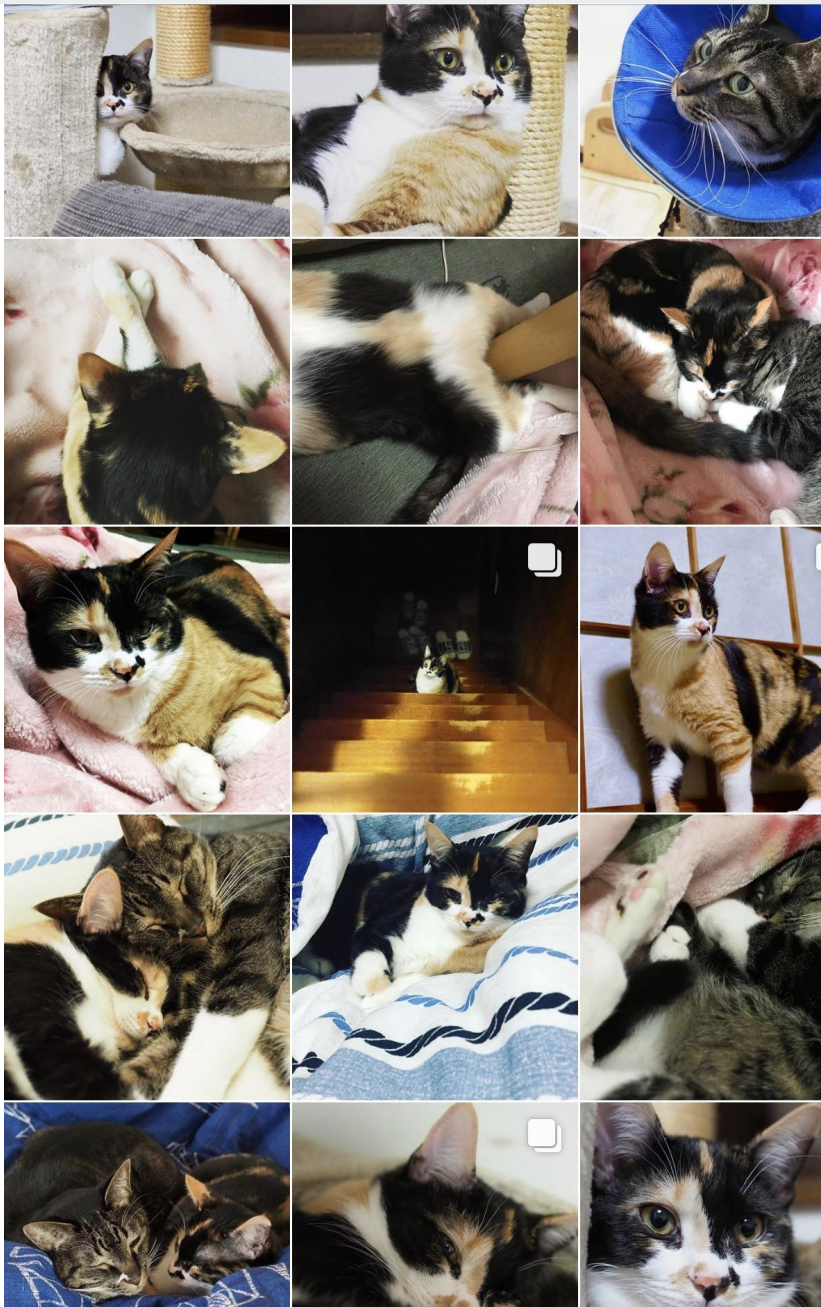
Quipper における「関心の分離」の歴史

May 24, 2018
Rails Developer Meetup 2018
Kensuke Nagae (@kyanny)

このトークについて

- 私について
  - 長永 健介（ながえ） @kyanny
  - 猫を二匹飼っています
  - Quipper で働いています
- Quipper について
  - オンライン教育サービスを開発・運営しています
    - インドネシア・フィリピン・メキシコ
  - 日本では「スタディサプリ」を開発しています
- このトークについて
  - Quipper が取り組んできた、ソフトウェア開発における「関心の分離」の事例を紹介します



アジェンダ

1. なぜ「関心の分離」なのか
    - 「関心の分離」とは何か
    - なぜ「関心の分離」が必要か
    - なぜ Quipper において「関心の分離」が重要なのか
2. 「関心の分離」事例研究
    - 名前空間を分ける
    - API エンドポイントを分ける
    - アプリケーションを分ける
    - データベースを分ける
3. 今後の展望
    - Quipper のシステムアーキテクチャの問題点
    - マイクロサービスアーキテクチャへの移行

1. なぜ「関心の分離」なのか

1. **なぜ「関心の分離」なのか**
    - 「関心の分離」とは何か
    - なぜ「関心の分離」が必要か
    - なぜ Quipper において「関心の分離」が重要なのか
2. 「関心の分離」事例研究
    - 名前空間を分ける
    - API エンドポイントを分ける
    - アプリケーションを分ける
    - データベースを分ける
3. 今後の展望
    - Quipper のシステムアーキテクチャの問題点
    - マイクロサービスアーキテクチャへの移行

「関心の分離」とは何か

Wikipedia より引用

関心の分離（かんしんのぶんり、英語 : separation of concerns、SoC）とは、ソフトウェア工学においては、プログラムを関心（何をしたいのか）毎に分離された 構成要素で構築することである。

プログラミングパラダイムは開発者が関心の分離を実践することを手助けするものもある。その為には、モジュール性とカプセル化 の実装のしやすさが重要となる。

関心の分離は複雑で依存関係が入り乱れたシステムの理解・設計・運用を容易にする ことが出来るので他の工学分野でもみられる。

https://ja.wikipedia.org/wiki/関心の分離

複雑なソフトウェアを
適切に分離することによって
扱いやすくすること

なぜ「関心の分離」が必要か

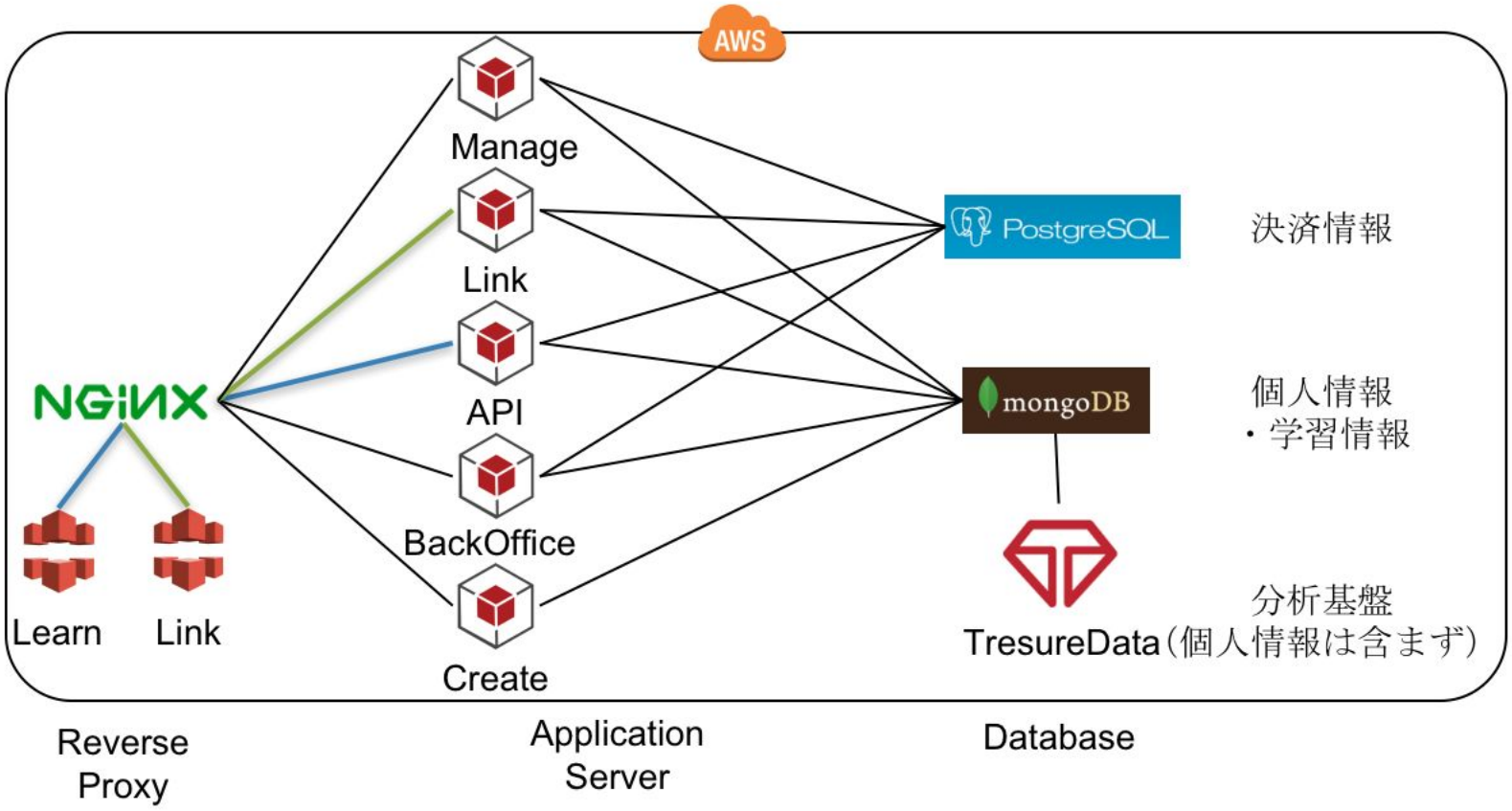
「関心の分離」を行わないと、どうなるか

- *関心（何をしたいのか）毎に分離された*
  - 分離されていない、責務の大きいメソッド・クラス・アプリケーションになってしまう
- *モジュール性とカプセル化*
  - モジュール性が低く、カプセル化されていないプログラムになってしまう
- *複雑で依存関係が入り乱れたシステムの理解・設計・運用を容易にする*
  - 複雑で依存関係が入り乱れており、理解・設計・運用が難しいシステムになってしまう

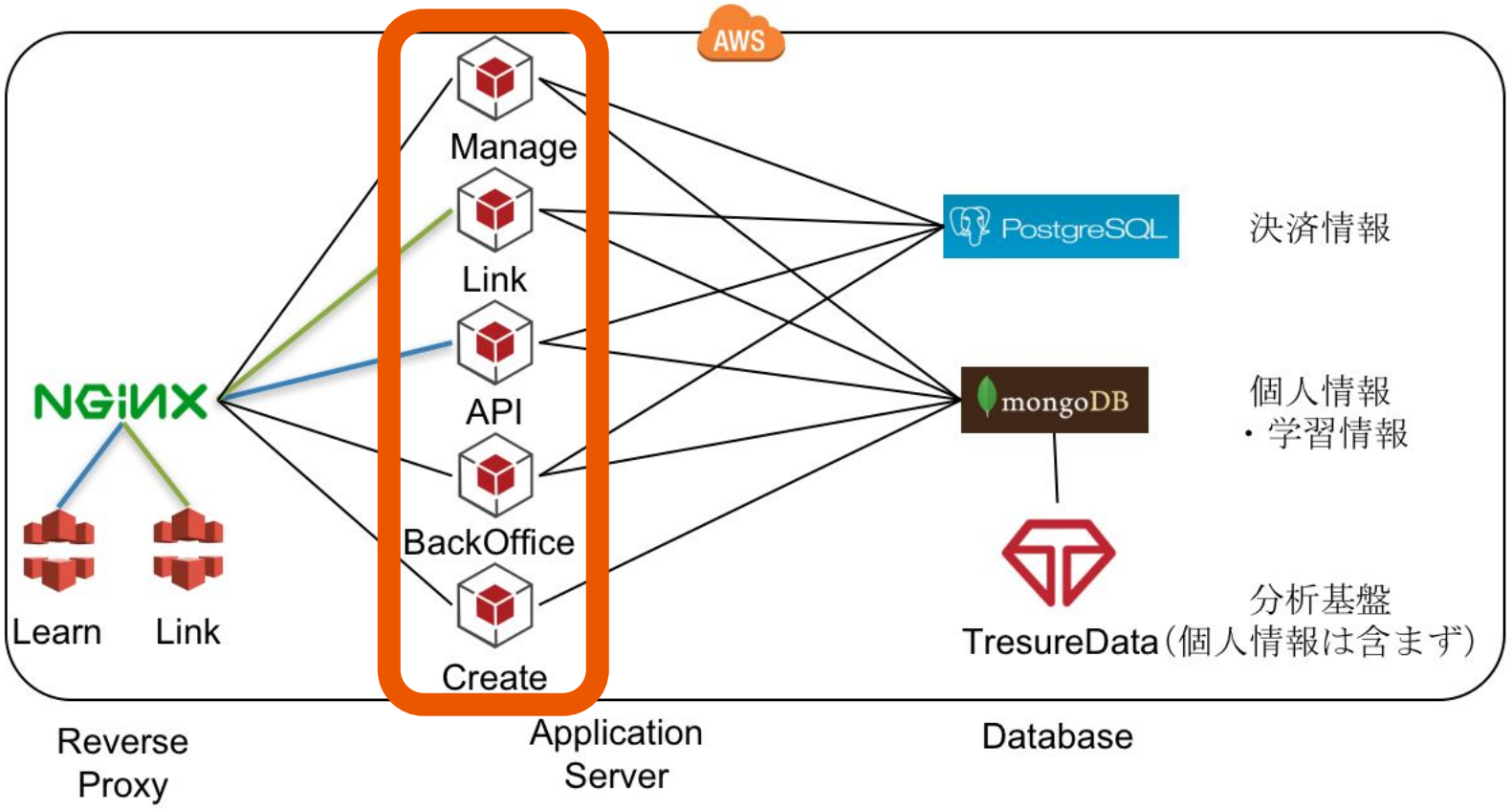
ソフトウェアは複雑化していく
複雑さを減らして
未来に備える

なぜ Quipper において「関心の分離」が重要なのか

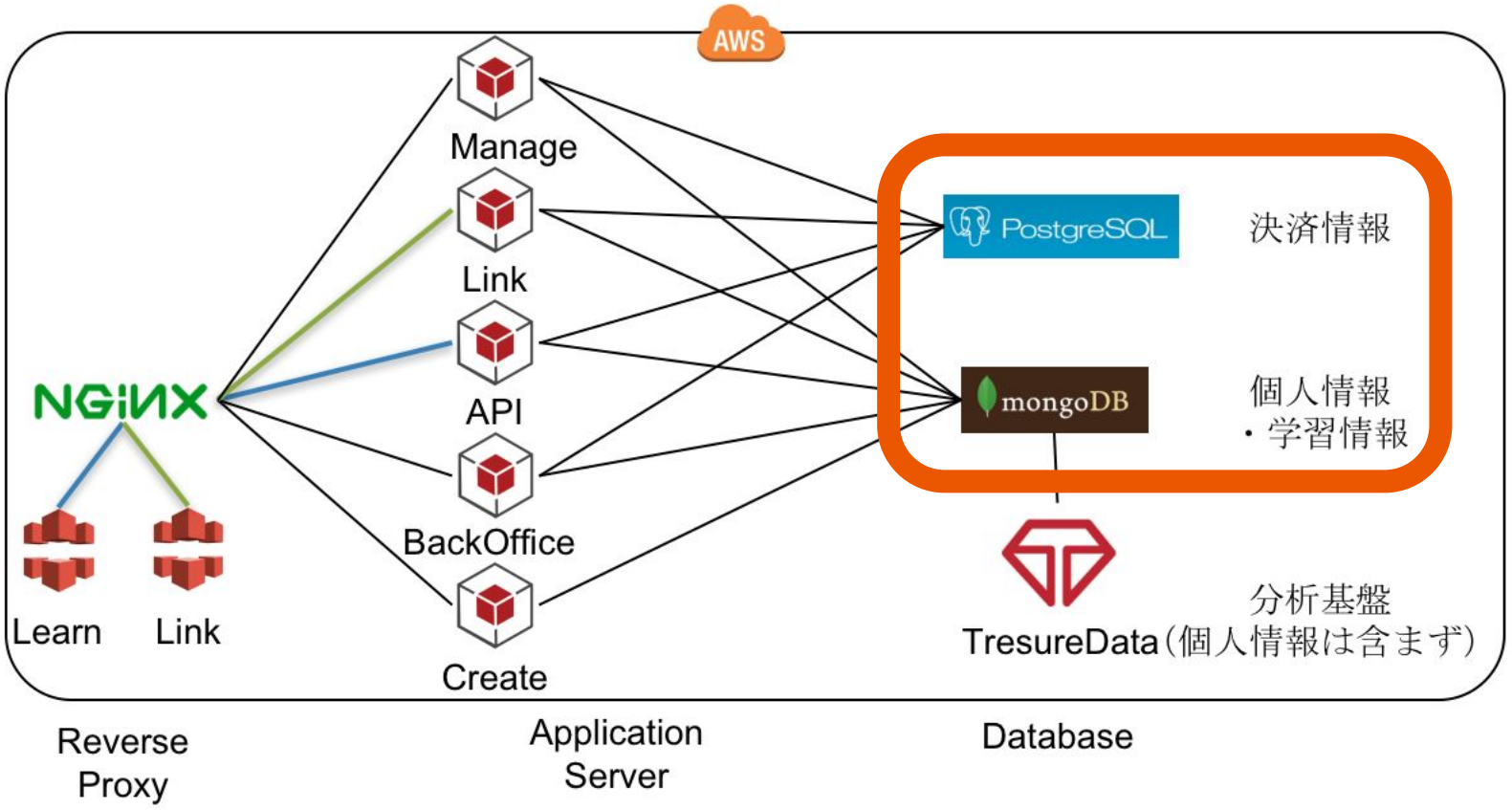
Quipper システム概観 (サプリ)



Quipper システム概観 (サプリ)

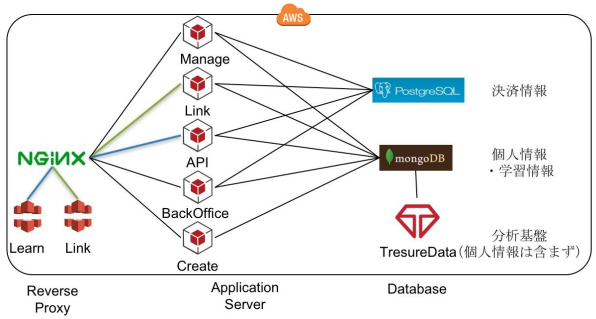


Quipper システム概観 (サプリ)

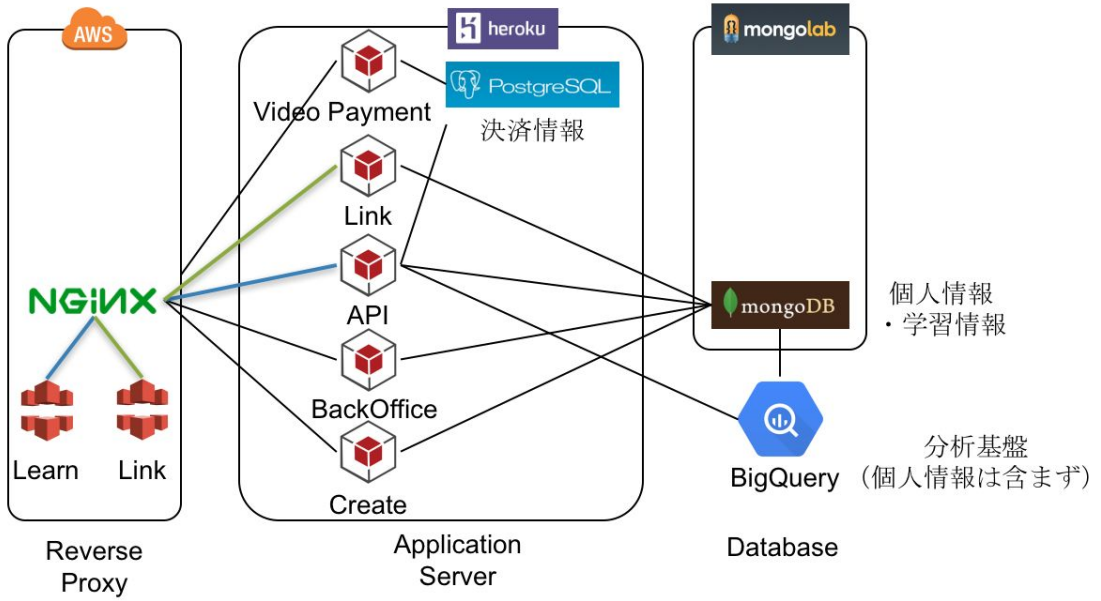


似てるけど結構違うインフラがもう一セットある

Quipper システム概観 (サプリ)



Quipper システム概観 (グローバル)



Quipper の複雑なシステムに「関心の分離」という武器で立ち向かう

2.「関心の分離」事例研究

1. なぜ「関心の分離」なのか
    - 「関心の分離」とは何か
    - なぜ「関心の分離」が必要か
    - なぜ Quipper において「関心の分離」が重要なのか
2. **「関心の分離」事例研究**
    - 名前空間を分ける
    - API エンドポイントを分ける
    - アプリケーションを分ける
    - データベースを分ける
3. 今後の展望
    - Quipper のシステムアーキテクチャの問題点
    - マイクロサービスアーキテクチャへの移行

名前空間を分ける

スタディサプリ専用のモデルを隔離する

- 「団体会員コード」というモデル
- 生徒が団体（学校）に所属している状態を表す
- スタディサプリに固有の概念（だった）
  - Quipper は別のモデルで学校への所属状態を表していた
- 将来 Quipper にも同様の概念を取り入れることを想定して、スタディサプリ専用の名前空間に隔離した
  - Aya:: ネームスペースがスタディサプリ用

```
module Aya
  class OrganizationMembership
    key :code, String
    belongs_to :user
    belongs_to :organization
  end
end

class User
  one :organization_membership,
    class_name: "Aya::OrganizationMembership"
end
```

この設計にした理由

- スタディサプリ専用と割り切った設計にできる
  - ドメインの知識が豊富なモデルになる
    - shinro 属性（スタディサプリ進路との連携）
    - 小書き対応（シュン・シユン）
  - 「早すぎる最適化」を避けられる
    - if スタディサプリ elsif Quipper. のような分岐の排除

```
module Aya
  class OrganizationMembership
    key :shinro_first_name_katakana, String
    key :shinro_last_name_katakana, String

    def set_shinro_attributes(first_name_kana:, last_name_kana:, birth_year:, birth_month:, birth_day:)
      self.shinro_first_name_katakana = first_name_kana.present? ? Schema::Utils::Aya::KogakiKanaConverter.upcase(first_name_kana).katakana : first_name_kana
    end
  end
end
```

後になって困ったこと

- 「団体会員コード」の Quipper 版ができて、かぶった
  - スタディサプリ用モデルへの association 名にプレフィックスつけるのをさぼった結果、大量のコード修正が発生
  - スタディサプリと Quipper で共用しているクライアントアプリケーション側で、似て非なるモデルを抽象化して扱わなくてはならない
- スタディサプリの仕様にフィットしすぎて、再利用できない
  - 再利用を意識していれば防げたかもしれないコードの重複・再実装が発生

```
class OrganizationMembership
  key :code, String
  belongs_to :user
  belongs_to :organization
end

class User
  one :organization_membership
# one :organization_membership,
#   class_name: "Aya::OrganizationMembership"
  one :aya_organization_membership,
    class_name: "Aya::OrganizationMembership"
end

membership = user.aya_organization_membership
organization_membership =
  user.organization_membership
```

API エンドポイントを分ける

特定のクライアント専用のエンドポイントを用意する

- ユーザー作成 API
- 新規会員登録フォームから POST する
- Quipper とスタディサプリでは新規会員登録フォームの仕様が異なる
  - 日本では姓名（ふりがな）が必須項目
  - インドネシア人は一般に姓を持たない
- クライアント側アプリケーションのプラットフォームによっても仕様が異なる
  - iOS アプリでは Apple の規約により生年月日の取得が禁止されている

```
# Quipper 用
POST        /v1/users

# スタディサプリ用
POST        /aya/v1/user

# スタディサプリ iOS アプリ用
POST        /aya/v1/ios/users
```

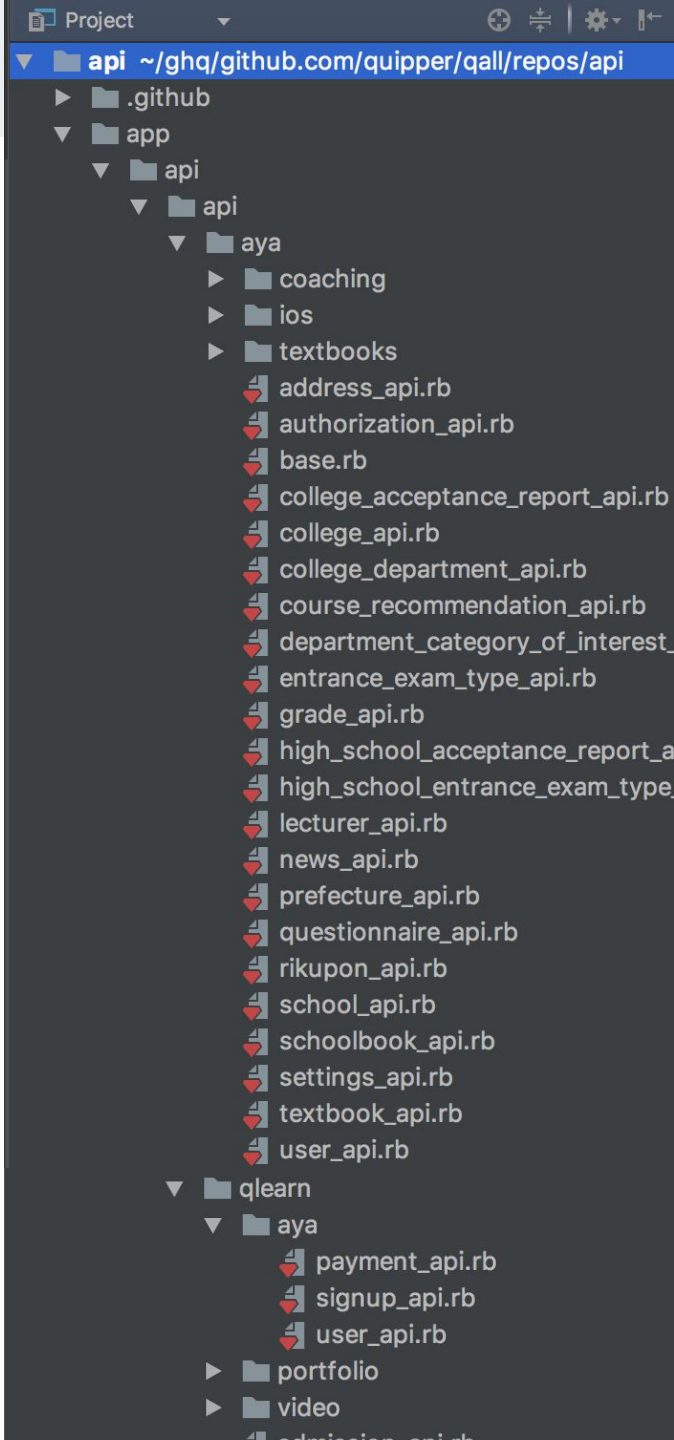
この設計にした理由

- API の実装の複雑化を防げる
  - 一つの API で全てのクライアントに対応しようとすると、分岐が増えてしまう
- クライアントの仕様変更に柔軟に追従できる
  - エンドポイントの仕様を変更しても、他のクライアントに悪影響を与えない

```
# 実在しないコードです
if is_sapuri?
elsif is_quipper?
 if is_indonesia?
   User.skip_callback(:save, :before,
:validates_first_name)
 elsif is_mexico?
 end
end
```

後になって困ったこと

- API が乱立
  - 確かに特定のクライアント専用の API たちではあるが……
  - 気軽に新しい API を作れるぶん、古い API を改訂の改訂が滞る
    - いつのまにか使われてない（かもしれない）API 多数
- 名前空間の混乱
  - Q. スタディサプリ Web クライアントが利用しているユーザー作成 API は？
  - A. /qlearn/v1/aya/signup/users
  - 直感に反する

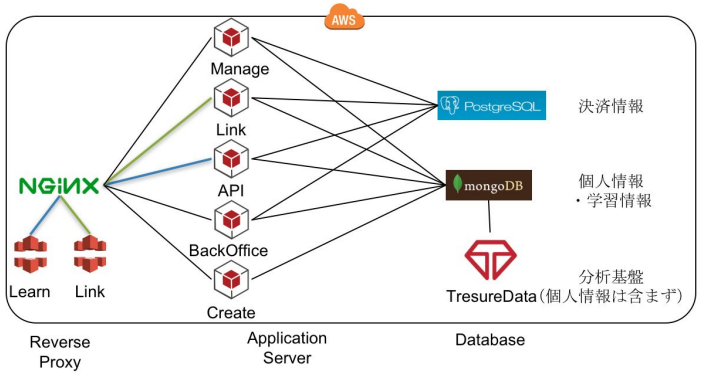


アプリケーションを分ける

用途ごとにアプリケーションを分離する

- 生徒の学習用（Learn）
- 先生の宿題管理・生徒管理用（Link）
- Quipper の運用担当者用（BackOffice）
- Quipper の教材制作者用（Create）
- 生徒・親の決済用（Manage）

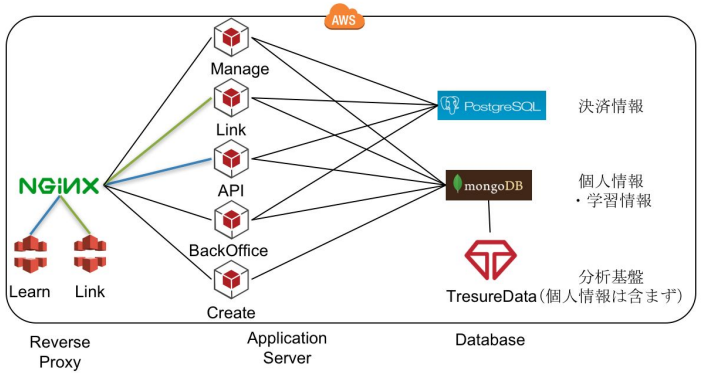
Quipper システム概観 (サプリ)



この設計にした理由

- 必要な機能に特化しやすい
- システム要件に最適化しやすい
  - 生徒用はレスポンシブデザイン
  - 先生用は大きい画面サイズ向け
  - 決済ゲートウェイ・決済用データベースとの接続は決済用アプリケーションが担う
- 不要な機能を取り除ける
  - 「一般ユーザーが管理者サイトにアクセスできる」ようなトラブルを避けやすい

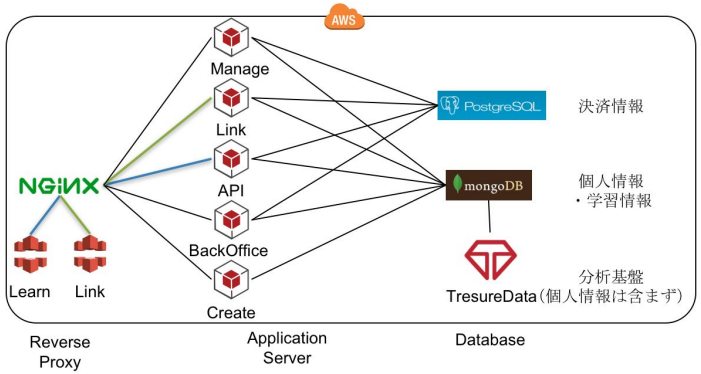
Quipper システム概観 (サプリ)



後になって困ったこと

- 見かけほど分離されていない
  - データベース・テーブルは共有
  - アプリケーションをまたぐ機能もある
    - テストしづらい
  - 決済にまつわる責務を決済用アプリケーションのみに集約できなくなったり
- システム全体として複雑になる
  - ビジネスロジックの重複を防ぐためにモデル層を共有ライブラリ化したが
    - 開発の手間が増す
    - バージョン不一致による弊害

Quipper システム概観 (サプリ)

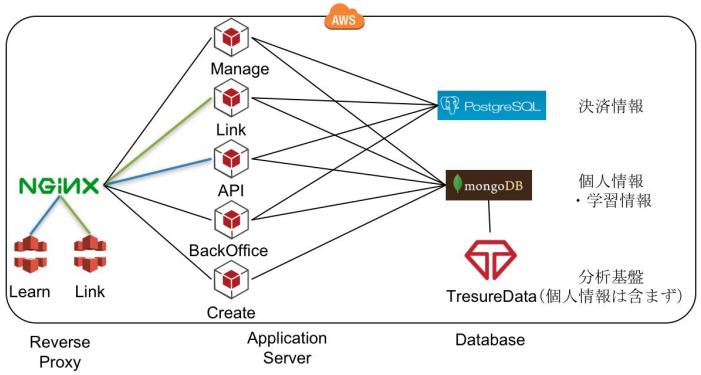


データベースを分ける

用途と特性で使い分ける

- MongoDB
  - 決済関連のトランザクションデータ以外全部
  - 元々 Quipper は MongoDB のみ
- PostgreSQL
  - 決済関連のトランザクションデータ用
  - 有料サービスを提供開始するタイミングで導入
  - 決済用アプリケーションの導入と合わせて

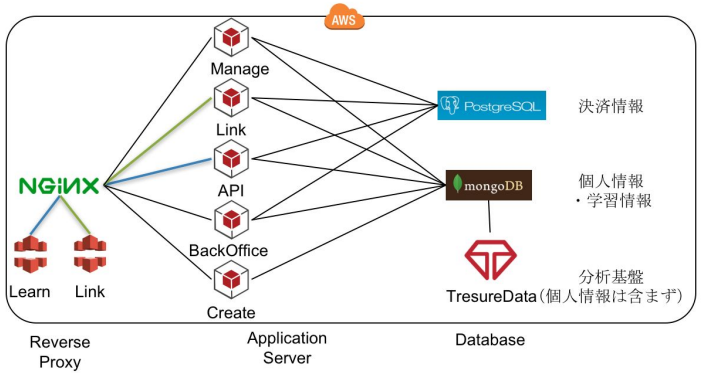
Quipper システム概観 (サプリ)



この設計にした理由

- 「MongoDB を決済データに使うのは無い」が当時の開発者全員の共通見解
  - 当然 RDBMS を使う
  - PostgreSQL を選んだのは当時 Heroku を使っていたから
  - 正直、MongoDB 以外ならなんでもよかった

Quipper システム概観 (サプリ)



後になって困ったこと

- 「決済用データベースは決済用アプリケーションから利用する」という前提が崩れた
  - システム要件の変化により、他のアプリケーションからも決済用データベースの情報が必要になった
- 「決済用データベースは決済関連のトランザクションデータ用」という前提も崩れた
  - 決済以外の機能でも RDBMS のトランザクションを利用したい
  - 別の PostgreSQL インスタンスを用意するのもアレなので、流用することに
  - 「何のデータをどのデータベースに格納するか」のポリシーが曖昧に

良かれと思って
いろいろやってきたけど
現実は厳しい

3. 今後の展望

1. なぜ「関心の分離」なのか
    - 「関心の分離」とは何か
    - なぜ「関心の分離」が必要か
    - なぜ Quipper において「関心の分離」が重要なのか
2. 「関心の分離」事例研究
    - 名前空間を分ける
    - API エンドポイントを分ける
    - アプリケーションを分ける
    - データベースを分ける
3. **今後の展望**
    - Quipper のシステムアーキテクチャの問題点
    - マイクロサービスアーキテクチャへの移行

Quipper のシステムアーキテクチャの問題点

「関心の分離」という視点から見ると

- *関心（何をしたいのか）毎に分離された*
  - 分離のされ方が適切ではなくなってきた
  - 例: 生徒の学習実績を表すデータ構造が先生向けアプリケーションからは利用しづらかったため、扱いやすく変形したデータを別で持つようになり、重複が発生した（重複によるデータ不整合も）
- *モジュール性とカプセル化*
  - モジュール性・カプセル化ともに低下してきた
  - 例:「生徒がクラスに所属する」という状態変化はひとつのデータ操作で済むが、そのイベントに付随するさまざまな状態変化も呼び出し側が意識して処理しないといけない
- *複雑で依存関係が入り乱れたシステムの理解・設計・運用を容易にする*
  - 複雑で依存関係が入り乱れたシステムになってきて、理解・設計・運用が困難に
  - 例: あるアプリケーションのためにモデル定義を変更したり、データを操作したりすると、別のアプリケーションの思いもよらない箇所に影響が出る

一言でいうと
「分断されたモノリス」
そこで……

マイクロサービスアーキテクチャへの移行

半年以上前から始めてます

- Quipper・スタディサプリ共通のシステム基盤を刷新
  - 数年ぶりの大規模なアップデート
  - 数年かけて現行システムから移行していく計画
- CTO 直属のプロジェクト
  - 会社として技術的負債の返済に本腰を入れる
- Quipper にとって新しい技術への挑戦
  - システム基盤部分に Kubernetes を採用
  - マイクロサービス間の通信プロトコルに gRPC を採用
- Quipper 開発チームの強みを活かす
  - Ruby と Ruby on Rails は引き続き活用
  - あくまで「理にかなう」選択をするのが Quipper 流

Quipper も
マイクロサービス
やっていくぞ

We are hiring!　DMください！


@mtsmfm


@kyanny