SHOWCASE GiG

Goで実践するBFP　−backend for product−

2025/01/18 Gopher’s Gathering

自己紹介

- 照井寛也
- 株式会社Showcase Gig
  - エンジニアリングオフィス
- Sendai.go オーガナイザー
- X : @10_ru_1

©Showcase Gig

セッションの概要

- BFP導入のきっかけと背景
- BFPとは
- BFP導入に向けて
- BFPをGoで実装する
- BFPの効果

©Showcase Gig

BFP導入のきっかけと背景

©Showcase Gig

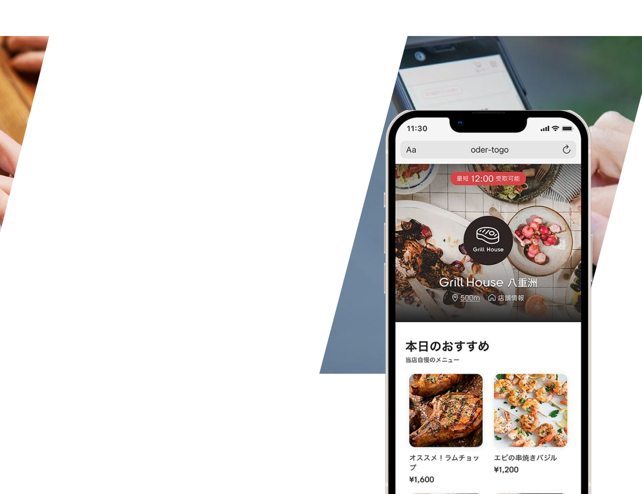
提供プロダクト

店内モバイルオーダー

O:der Table

テイクアウトモバイルオーダー

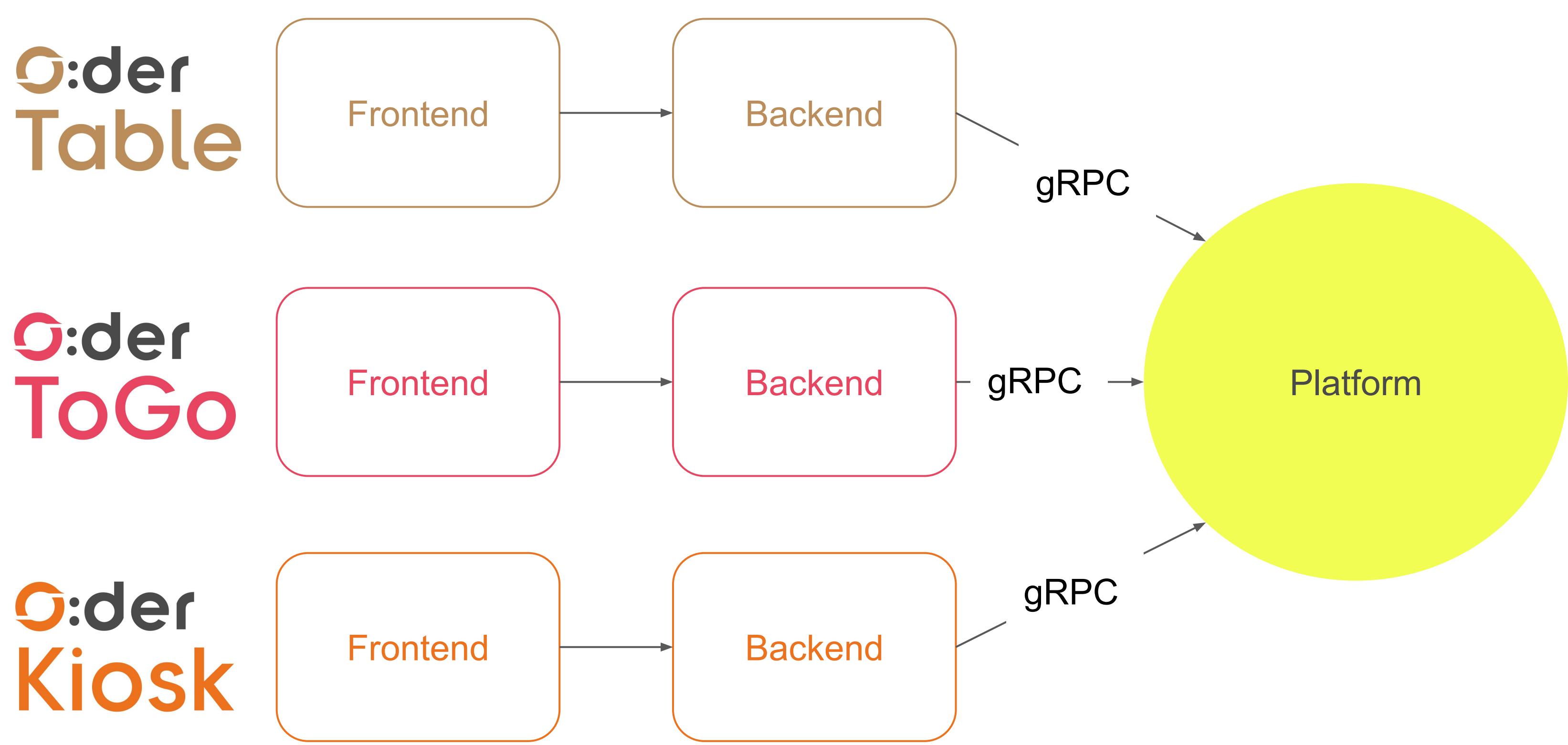
O:der ToGo



次世代タッチパネル型
注文決済端末

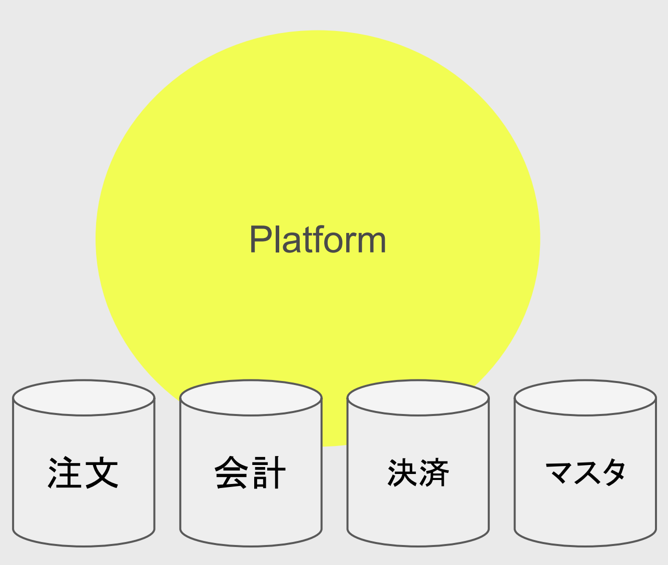
O:der Kiosk

プロダクト構成



©Showcase Gig

Platformの目指す姿



- 注文・会計・決済・マスタデータを持つ
- 共通マスタから様々な飲食形態に対応する
- 各プロダクトからの注文・会計データを集約し、データを利活用できる状態にする

©Showcase Gig

Platformの目指す姿

飲食形態（プロダクト）に依存しないAPI設計

©Showcase Gig

プロダクトに依存しないAPI = primitiveなAPI

```
// どのプロダクトでも共通して使われるデータモデル
message Menu {
        uint64 menu_id = 1;
        uint64 restaurant_id = 2;
        string name = 3;
        repeated Item items = 4;
}

message Item {
        uint64 item_id = 1;
        uint64 restaurant_id = 2;
        string name = 3;
        string description = 4;
        uint32 price = 5;
        repeated Allergen Allergens = 6;
}
```

©Showcase Gig

プロダクトに依存しないAPI設計

- APIの抽象度が高く以下の問題が生じてきた
  - primitiveなAPIの特性上、複数回呼び出す必要がある
    - レイテンシーの悪化
    - インフラコストの増加
  - primitiveなAPI故、プロダクトの処理には必要のないデータも含まれる
    - 導入企業のマスタによっては、gRPCのデータサイズ上限の4MBを突破

©Showcase Gig

実際に生じた問題 💥

- シチュエーション
  - 店員さんが商品を品切れにしたい
- 設計の課題
  - Platform側が提供しているAPIは、あくまで「メニューの取得」のみ
  - メニュー取得のAPIは、商品情報全てを返している
- 問題
  - プロダクト側では「注文できる商品の一覧」が欲しいにもかかわらず、まず全メニューを取得しなくてはならない
    - メニュー内の商品が重複している場合は、プロダクト側で重複を弾く実装が必要
    - 品切れ画面に不要な情報も含まれる
      - メニューのデータ量が多い店舗ではデータが 4MBを超える

©Showcase Gig

プロダクトに依存しないAPI = primitiveなAPI

```
// どのプロダクトでも共通して使われるデータモデル
message Menu {
        uint64 menu_id = 1;
        uint64 restaurant_id = 2;
        string name = 3;
        repeated Item items = 4; ←この情報欲しい
}

message Item {
        uint64 item_id = 1; ←この情報欲しい
        uint64 restaurant_id = 2; ←この情報欲しい
        string name = 3; ←この情報欲しい
        string description = 4;
        uint32 price = 5;
        repeated Allergen Allergens = 6;
}
```

©Showcase Gig

実際に生じた問題 💥

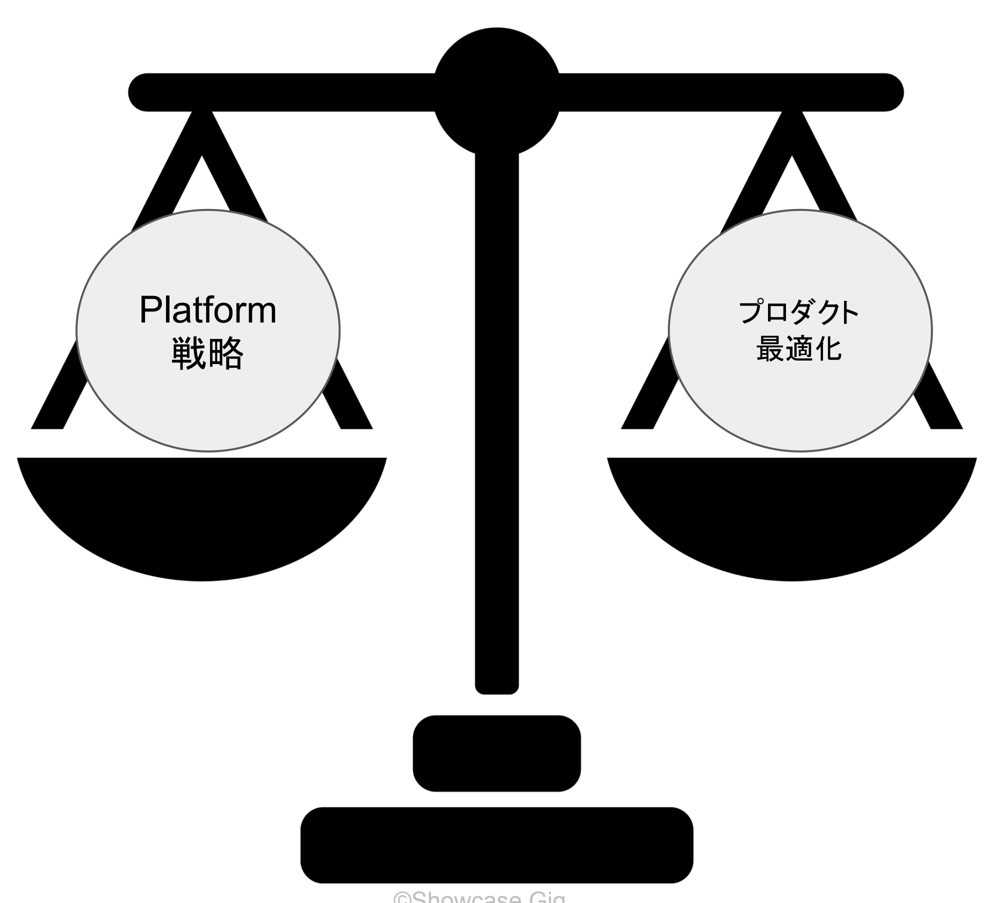
```
func (a *allItemUsecase) ListByMenuIds(
	ctx context.Context,
	in ListAllItemItemsByMenuIdsInput
) (*ListAllItemItemsByMenuIdsOutput, error) {
	// 4MBを超えることがある
	menus, err := a.pfMenuRepository.List(ctx, in.MenuIds)
	if err != nil {
		return nil, api_error.NewInternalError(err, "failed to list menus from pf")
	}

	if menus == nil {
		return nil, api_error.NewResourceNotFoundError("not found menus")
	}


	itemMap := make(map[types.ItemID]entity.Item,0)
	for _,menu := menus{
		// ここでmenusのデータを解析し、itemMapに必要な情報を格納する処理が必要
	}
	….
}
```

©Showcase Gig

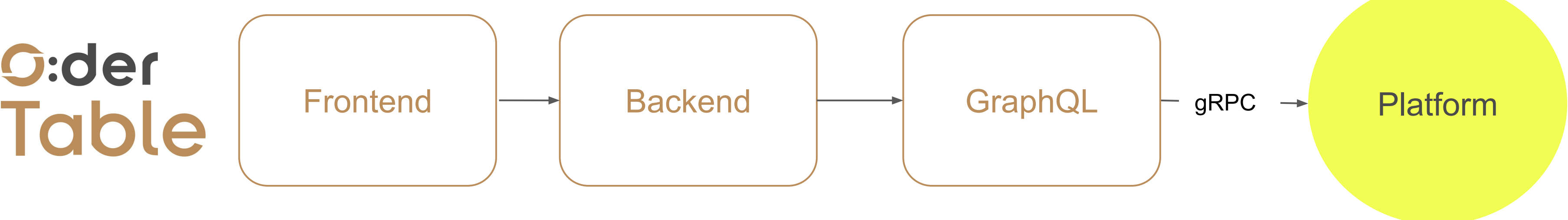
戦略の天秤



©Showcase Gig

GraphQLの選択肢

- GraphQLは以下から選択を見送った
  - レイテンシー
    - NWを挟むことによるレイテンシーの悪化
  - Github Repositoryの増加
    - コードの増加以上に認知負荷が増す可能性がある
      - repositoryの移動、protoの取り込み、CI/CD…..
    - インフラコストの増大
  - GraphQLの学習コスト
    - BFPであれば既存のgRPCの知識で達成可能



©Showcase Gig

BFP

©Showcase Gig

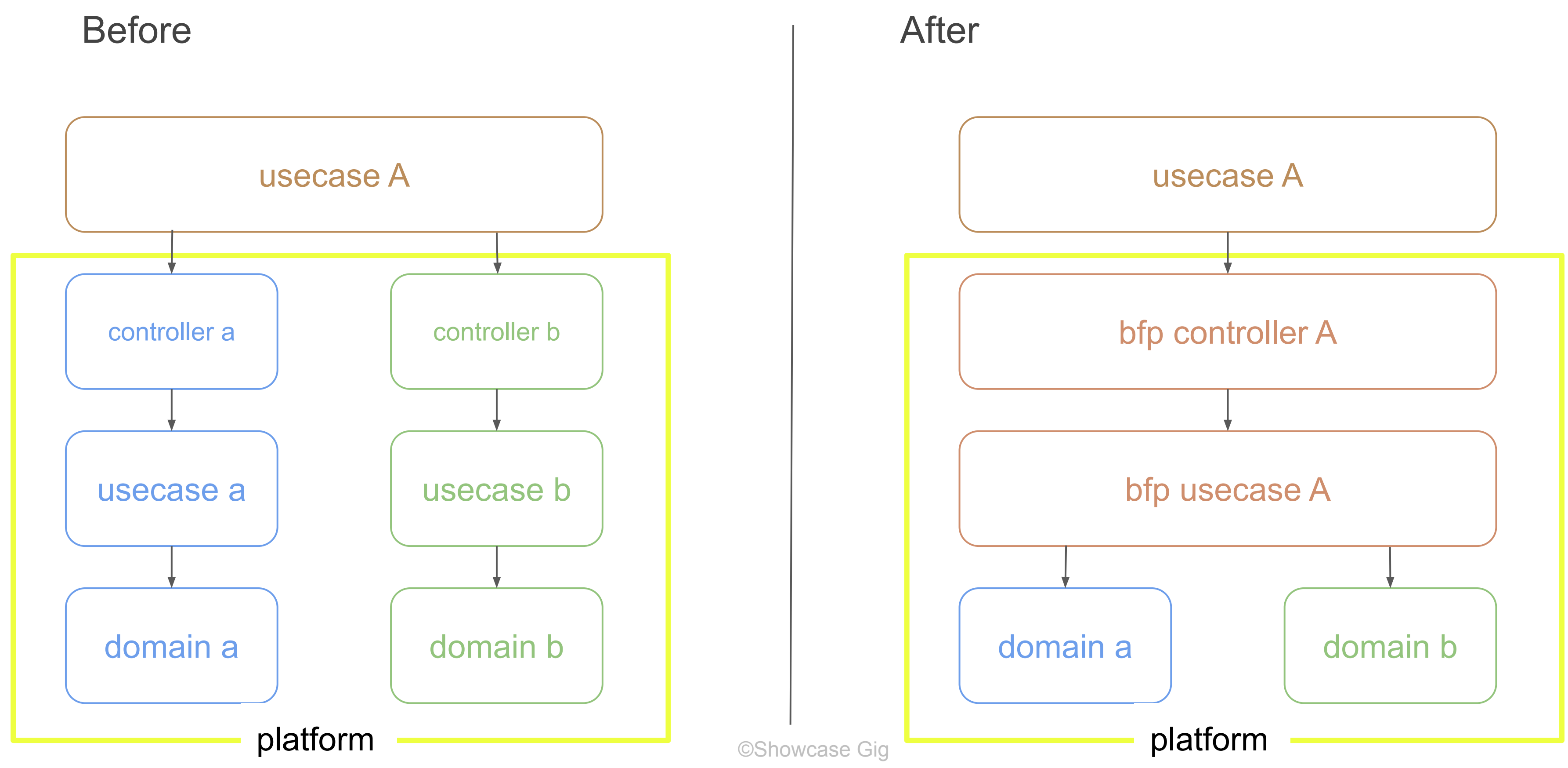
BFP（Backend For Product）

O:der Productが使いやすいAPIを提供することを目的とする。具体的には以下。

- O:der Productが欲しいデータを集めたEPの提供（集約・統合API）
  - platformのentityを跨いでデータを返す・永続化する
    - 例）AとBを呼んでO:der Productに必要なデータを形成しているが、それを1回で呼べるようにする
  - 注文・会計・決済をまとめて行うEPの提供
- O:der Productが高いパフォーマンス（データ量やレイテンシー）を出せるEPの提供（絞り込みAPI）
  - O:der Productに必要なデータのみを返す
    - 例）メニューに紐づく商品を重複なしで返す

©Showcase Gig

BFP (Backend For Product)



©Showcase Gig

Platform

飲食形態(プロダクト)に依存しないAPI設計
&
BFP

©Showcase Gig

platformの責務が拡大した

対話

©Showcase Gig

platformの責務が拡大した

対話＆対話

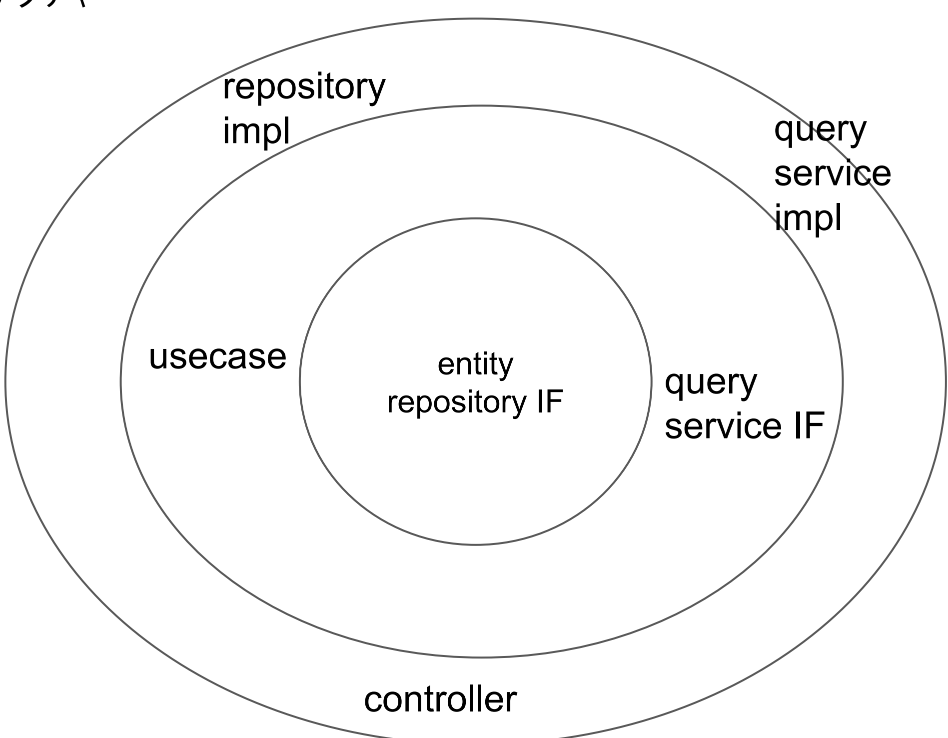
©Showcase Gig

BFPをGoで実装する

©Showcase Gig

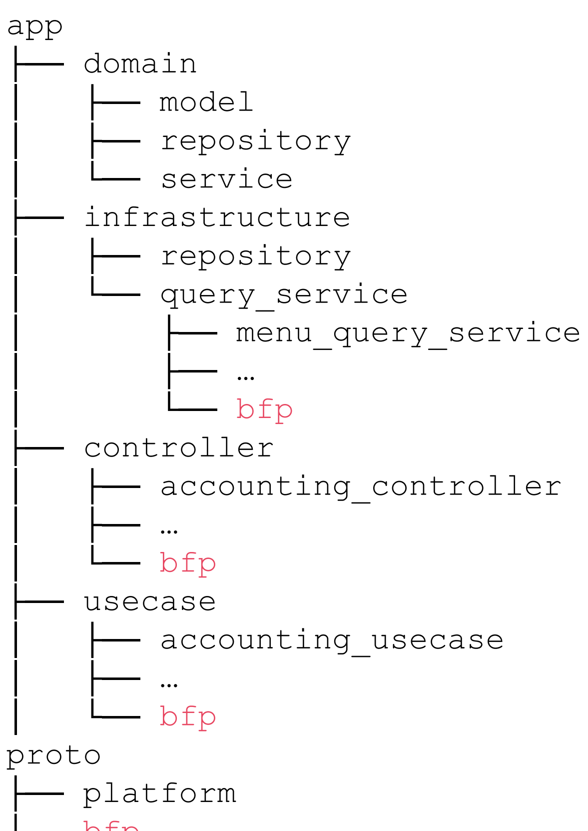
BFPをGoで実装する

- Platformのアーキテクチャ



©Showcase Gig

BFPをGoで実装する



- 既存のprimitiveなAPIに影響を与えない構成
- domainに対して変更を加えることはしない
- listの場合...
  - bfp controller→(usecase→) bfp query_service
- createの場合
  - bfp controller→bfp usecase→repository

©Showcase Gig

List処理（メニュー取得）の例

controller層

```
func (t *tableMenuController) ListUniqueItemsByMenuIds(
	ctx context.Context,
	in *tableApi.ListUniqueItemsByMenuIdsRequest
) (*tableApi.ListUniqueItemsByMenuIdsResponse, error) {

	// usecaseを意図的にスキップ（query_serviceの呼び出しだけのため）
	response, err := t.tableMenuQueryService.ListUniqueItemsByMenuIds(ctx, in.MenuIds)
	if err != nil {
		return nil, api_error.NewInternalError(err, "failed to ListUniqueItemsByMenuIds")
	}

	if response == nil {
		return nil, api_error.NewResourceNotFoundError("not found ListUniqueItemsByMenuIds")
	}

	return response, nil
}
```

©Showcase Gig

List処理（メニュー取得）の例

infra層（query_service IFの実装）

```
func (t *tableMenuQueryService) ListUniqueItemsByMenuIds(
	ctx context.Context,
	menuIds []uint64
) (*tableApi.ListUniqueItemsByMenuIdsResponse, error) {

	readQueryable := t.clients.ReadReplicaClient.ReadQueryable(ctx)

	query, params, err := sqlx.In("SELECT "+t.dao.MenuColumns()+" FROM "+t.dao.MenuTableName()+" WHERE `id` in (?) AND status != ?", menuIds, menu_entity.MenuStatusArchived)
	if err != nil {
		return nil, fmt.Errorf("failed to build query: %w", err)
	}

	// menuを元に、紐づく商品を取得するクエリ
	……

}
```

©Showcase Gig

Create処理（統合作成API）の例

- シチュエーション
  - O:der Togo(テイクアウト)のプロダクトで、注文と会計伝票の作成は同時に行われる
    - O:der Table(店内飲食）は（複数の）注文が行われ、その後別リクエストで会計伝票を作成する
- 実装
  - Platform側が提供しているAPIは、「注文の作成」「会計伝票の作成」それぞれ独立している
  - プロダクト側では1つのトランザクション内で「注文の作成」「会計伝票の作成」を行う必要がある
    - 場合によっては値引きの考慮もする
- 課題
  - 最低でも2回のEPを呼び出す必要があり、トランザクションの管理が必要になる
    - NWを経由するためにレイテンシーの悪化が懸念される

©Showcase Gig

Create処理（統合作成API）の例

controller層

```
func (c *orderToAccountingController) CreateV1(
	ctx context.Context,
	request *pb.CreateV1Request
) (*pb.CreateV1Response, error) {
	output, err := c.createUseCase.Create(ctx, order_to_accounting_usecase.CreateInput{
		…
	})
	if err != nil {
		return nil, api_error.NewInternalError(err, "failed to create resources due to internal error")
	}
	return &pb.CreateV1Response{
		OrderId: uint64(output.OrderID),
		AccountingVoucher: accounting_controller.NewVoucher(output.Voucher)
	}, nil
}
```

©Showcase Gig

Create処理(統合作成API)の例

usecase層(その1)

```
func (uc *createUseCase) Create(ctx context.Context, input CreateInput) (*CreateOutout, error) {

	err := input.validate()
	if err != nil{
		return err
	}

	orderEntity,err := entity.NewOrderEntity(....)
	if err != nil{
		return err
	}

	accountingEntity,err := entity.NewAccountingEntity(orderEntity,....)
	if err != nil{
		return err
	}
```

©Showcase Gig

Create処理（統合作成API）の例

usecase層（その2）

```
	// transaction管理内で行える
	err = uc.dbClients.WritableClient.RunInWriteTx(ctx, func(context context.Context, queryable
queryable.WriteQueryable) error {
		// 注文の永続化
		err = uc.orderRepository.Persist(ctx, queryable, orderEntity, orderedAt.Time)
		if err != nil {
			return fmt.Errorf("failed to persist order: %w", err)
		}
		// 会計伝票の永続化
		err = uc.voucherRepository.Persist(ctx, queryable, voucher)
		if err != nil {
			return fmt.Errorf("failed to persist voucher: %w", err)
		}
	}

	……
}
```

©Showcase Gig

BFPの効果

©Showcase Gig

BFPの効果

- メニュー取得・統合作成APIともにProduct EPからみたレイテンシーの改善が見られた
- 特に、メニュー取得APIに関しては、以下の成果が出た
  - EP処理を4327ms→501msに改善→**速度を8倍に改善**
  - 42.9MBだったレスポンスを1.9MBに削減→**レスポンスサイズを 1/20に改善**
    - また、gPRCのデフォルトの4MBに収まるサイズにまで短縮

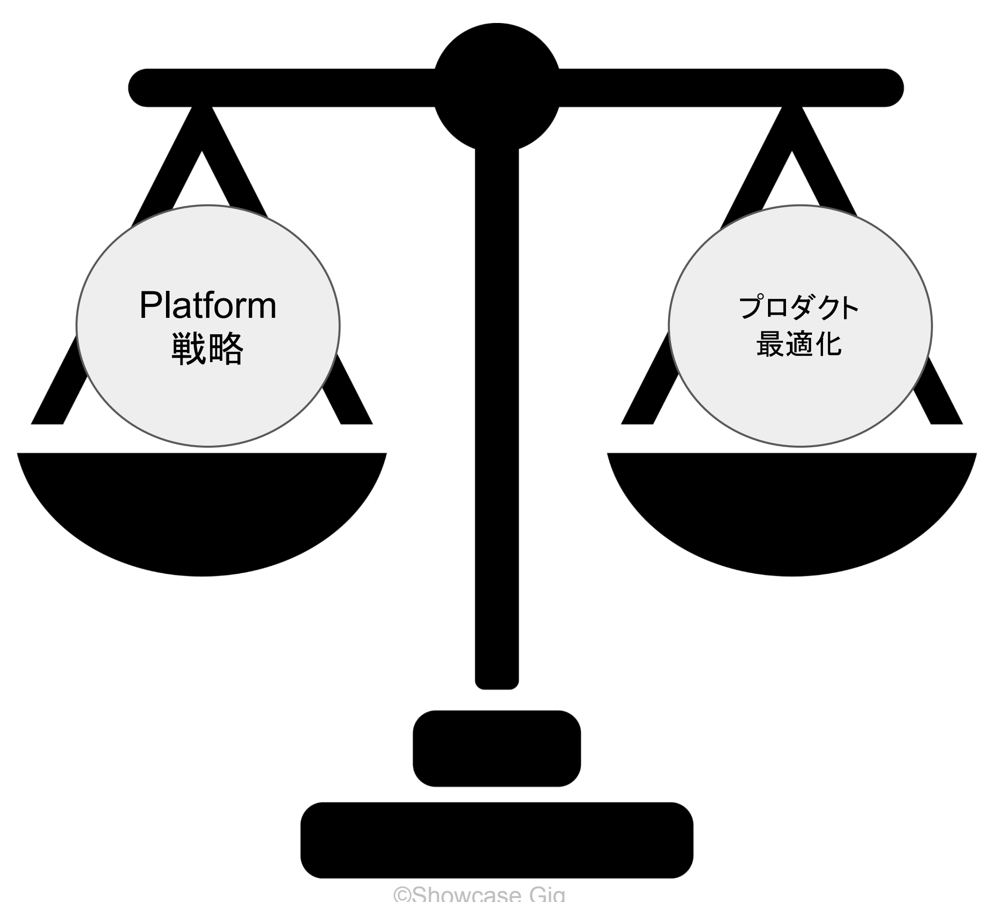
©Showcase Gig

BFPの今後

- BFPを活用することでプロダクトのレイテンシーをさらに改善し、エンドユーザーの体験をより良くする
- BFPが増えると、platformの責務・処理が増えていく
  - 各ドメインの責務を維持しつつ、疎結合を意識した設計を継続していきたい

©Showcase Gig

BFPの今後



©Showcase Gig

Thank you

©Showcase Gig