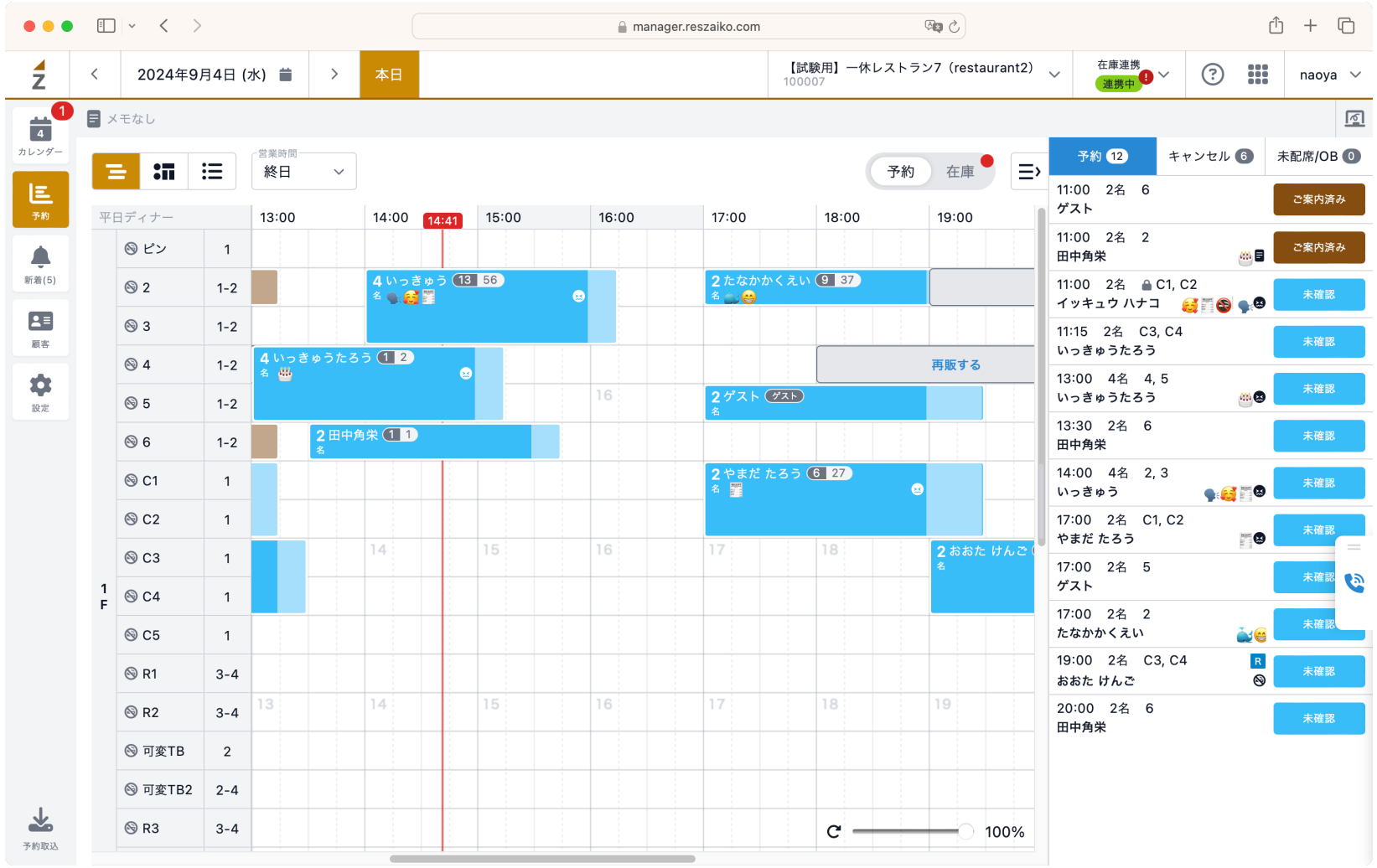
関数型スタイルを取り入れたアプリケーション開発の実践例

2024.9.07

株式会社 一休

伊藤 直也

飲食店向け「予約台帳」のバックエンドを TypeScript + 関数型スタイルで作っている



- 予約管理、空席管理、自動配席、顧客データベース、POS や CTI との連携など業務アプリケーションとしてはそこそこの規模
- バックエンドは TypeScript で記述された GraphQL サーバー
- 「Domain Modeling Made Functional」 (邦訳「関数型ドメインモデリング」) を参考に、関数型プログラミングのスタイルを取り入れて開発

What DDD source code should look like

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
              | Nine | Ten | Jack | Queen | King

    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = { Name:string;  Hand:Hand }
    type Game = { Deck:Deck;  Players: Player list }

    type Deal = Deck -> (Deck * Card)

    type PickupCard = (Hand * Card) -> Hand
```

Shared language

ドメインオブジェクトを型で表現する。Union 型を積極利用して不必要な状態を表現しない

ドメインイベントによるオブジェクトの状態遷移は関数適用

https://www.slideshare.net/ScottWlaschin/domain-modeling-made-functional-kandddinsky-2019

なお、書籍その通りにはやっていない

- ドメインオブジェクトを型で表現する → いいね
- エラーによる分岐は Result 型で扱い、エラー処理を明示的にする → いいね
- workflow に直接ドメインオブジェクトを定義する (トランザクションスクリプト?) / Repository パターンは要らない → なぜ? 🤔 参考にしていない

TypeScript で、どんなスタイルで実装しているか

- ドメインオブジェクトは型で構造を定義する
  - class ではなく interface

突き詰めると、ここが関数型プログラミング

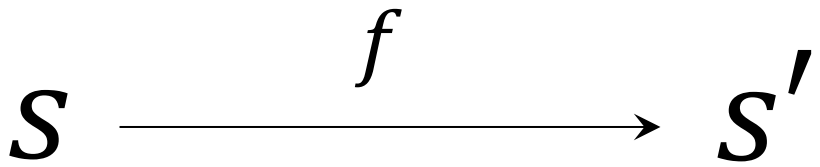
- オブジェクトの変更は「関数適用による状態遷移」としてイミュータブルにする
  - **「オブジェクトの内部状態の書き換え命令」ではなく「関数適用 (写像) により状態を移す」**

- 具体的なユースケースは "workflow" として実装
  - 入力値とドメインオブジェクトがドメインイベントに伴い新しい状態に遷移。その過程を型で定義
  - 状態遷移には失敗 (エラー) が伴う。戻り値を Result にし「Result を返す関数の合成」でフローを構築

ドメインオブジェクトは型で定義する

```
export interface Customer {
  id: CustomerId
  groupId: RestaurantGroupId
  name: CustomerName
  nameKana: CustomerNameKana | null
  nickname: CustomerNickname | null
  gender: Gender
  smoking: SmokingType
  memo: CustomerMemo | null
  archived: boolean

  phoneNumbers: CustomerPhoneNumber[]
  workplaces: CustomerWorkplace[]
  restaurantContexts: CustomerRestaurantContext[]
  emails: CustomerEmail[]
  events: CustomerEvent[]

  tagIds: TagId[]
  allergyIds: AllergyId[]
}
```

オブジェクトの変更は「関数適用による状態遷移」として実装する

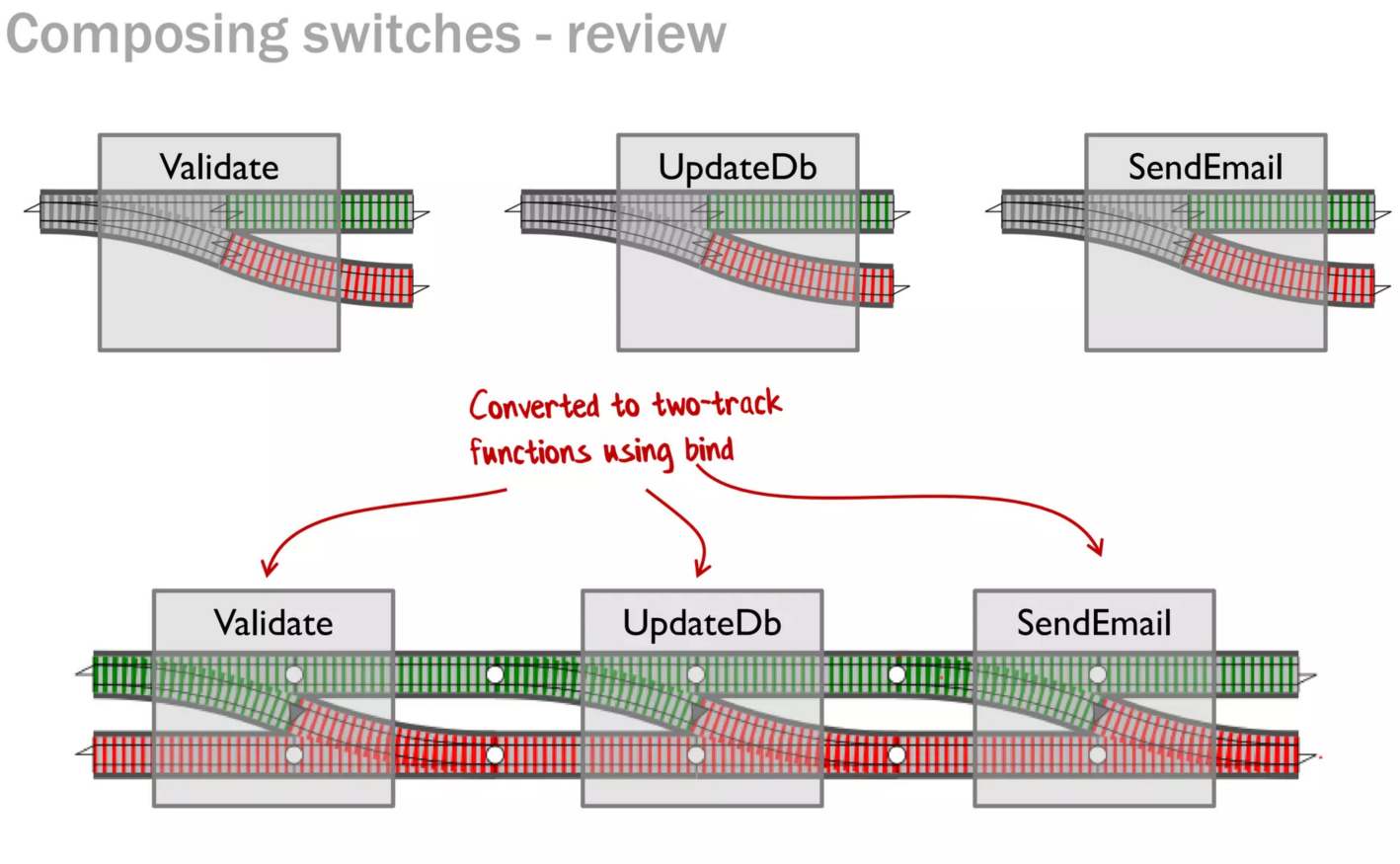


ある状態 s に関数 f を適用して、別の状態 s' を得る

```
// 顧客をアーカイブ状態にする
export const archiveCustomer = (customer: Customer): Customer => ({
  ...customer,
  archived: true,
})
```

ユースケースは "workflow" として実装する

Result を返す関数を合成して一本道の処理フローを作る Railway Oriented Programming



https://fsharpforfunandprofit.com/rop/

ワークフローのとあるサブステップ … 失敗との分岐は Result で表現

```
// 予約リクエストから来店者情報を抽出
type extractActualVisitor = (command: SiteReservationValidated)
  => Result<VisitorExtracted, ValidationError>

const extractActualVisitor: extractActualVisitor = (command) =>
  ok(command.input.reservationEvent.siteReservation.guest)
    .andThen(ReservationGuest)
    .map(choiceActualVisitor)
    .map((visitor) => ({
      ...command,
      kind: 'VisitorExtracted' as const,
      visitor: visitor,
    }))
```

ワークフローにおける状態遷移を型で定義する

```
interface UnvalidatedCommand {
  kind: 'Unvalidated'
  input: {
    groupId: RestaurantGroupId
    reservationEvent: UnvalidatedSiteReservationEvent
  }
}

interface SiteReservationValidated {
  kind: 'SiteReservationValidated'
  input: {
    groupId: RestaurantGroupId
    reservationEvent: ValidatedSiteReservationEvent
  }
}

interface VisitorExtracted {
  kind: 'VisitorExtracted'
  input: {
    groupId: RestaurantGroupId
    reservationEvent: ValidatedSiteReservationEvent
  }
  visitor: Visitor | ReservationHolder
}
```

```
interface CustomerNotIdentified {
  kind: 'CustomerNotIdentified'
  input: {
    groupId: RestaurantGroupId
    visitor: Visitor | ReservationHolder
    customer: ValidatedCustomer
    reservationEvent: ValidatedSiteReservationEvent
  }
}

interface CustomerIdentified {
  kind: 'CustomerIdentified'
  input: {
    groupId: RestaurantGroupId
    visitor: Visitor | ReservationHolder
    customer: IdentifiedCustomer
    reservationEvent: ValidatedSiteReservationEvent
  }
}

...

interface SiteReservationEventImported {
  kind: 'SiteReservationEventImported'
  reservation: CreatedReservation
  customer: CreatedCustomer | UnchangedCustomer | UpdatedCustomer
}
```

一連のサブステップを合成しワークフロー (ユースケース) を構成する

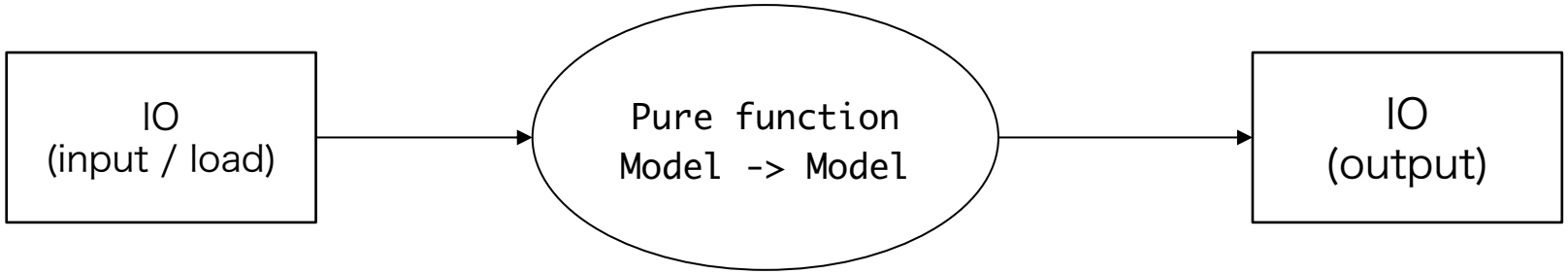
```
type WorkFlow = (command: UnvalidatedCommand)
  => ResultAsync<ImportSiteReservationEvent[], ImportSiteReservationError>

export const importReservationEventWorkFlow =
  (
    findIdenticalCustomer: findIdenticalCustomer, // ドメンサービスを高階関数としてDI
    findReservationSlotsByTableAllocations: findReservationSlotsByTableAllocations,
    findBestTablesForAssignedTable: findBestTablesForAssignedTable
): WorkFlow =>
  (command) =>
    ok(command)
      .andThen(validateReservationEvent) // バリデーション
      .andThen(extractActualVisitor) // 来店者情報の抽出
      .asyncAndThen(identifyCustomer(findIdenticalCustomer)) // 名寄せ
      .andThen((command) => {
        switch (command.kind) {
          case 'CustomerIdentified':
            return updateCustomer(command) // 顧客の更新
          case 'CustomerNotIdentified':
            return createNewCustomer(command) // 新規顧客の作成
        }
      })
      .andThen(importReservationEvent(findBestTablesForAssignedTable, isOverlappingWithOtherReservations)
      .andThen(findAffectedReservationSlots(findReservationSlotsByTableAllocations)
```

関数型スタイルでやってるのは主にドメイン層。それ以外は従来通りの作り

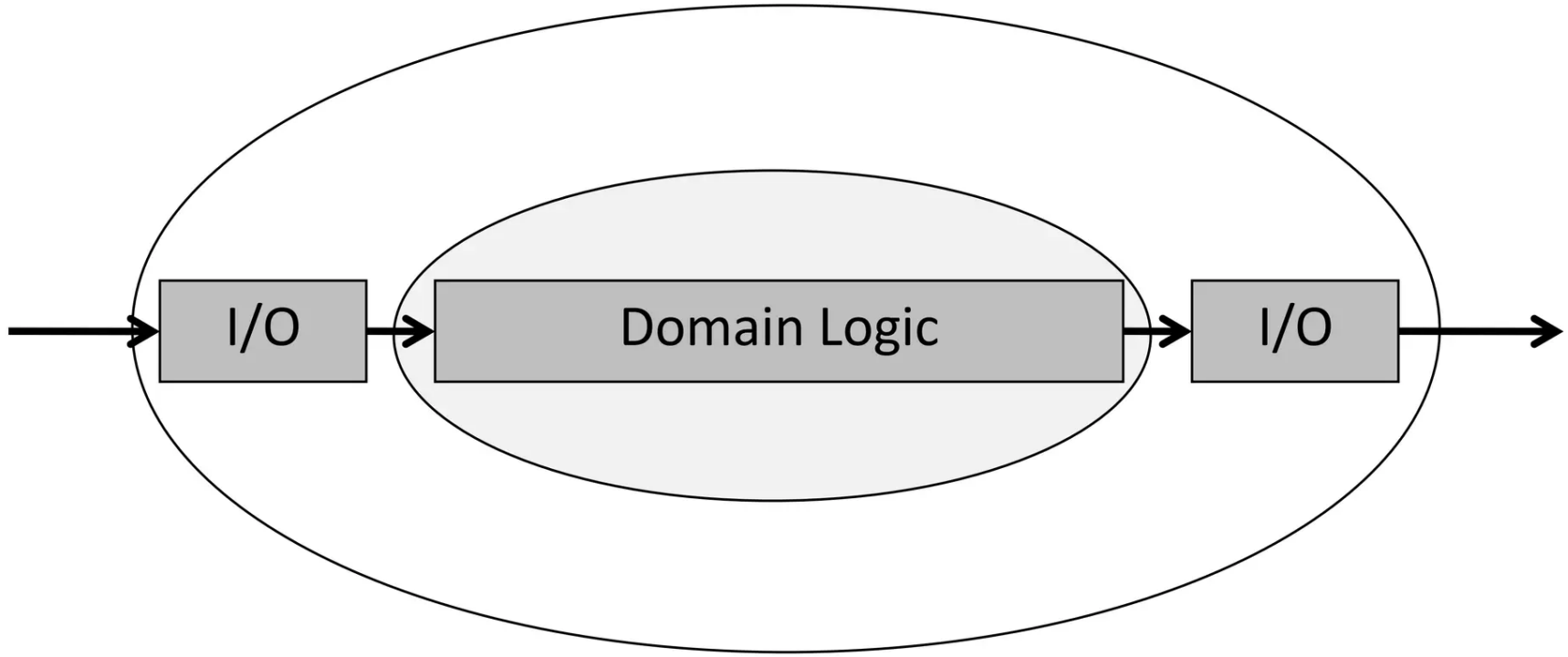
GraphQL Input

データベース

```
ok(model).andThen(workflow).andThen(saveCreatedTag(context))
```



The "onion" architecture



Core domain is pure, and all I/O is at the edges

https://www.slideshare.net/slideshow/pipeline-oriented-programming/250490001

システム全体の実装は、従来の実装からそれほど変わらない

- アプリケーション全体はこれまで通りのオニオンアーキテクチャー

- ドメインレイヤーは関数型といってもポイントは主に3つのみ
  - オブジェクトの構造を型 (interface) で表現する
  - オブジェクトの変更を、関数適用 + 状態遷移でイミュータブル/明示的に表現する
  - 失敗の分岐 (エラー) は Result で表現する

なぜドメインレイヤーを関数型スタイルにしたいのか

- 型を有効に利用したい
  - 静的検査に寄せていくと「動かさないと分からないこと」が減る
  - ドメイン層において「不必要な状態存在」を減らすと堅牢になる … Union 型を積極利用したい
- オブジェクトの変更を「関数適用による状態遷移」にすると型を記述しやすい
  - 状態遷移前、後のオブジェクトが関数の引数と戻り値に表れる
- 失敗による分岐 … つまりエラー処理も型で扱いたい。Result 型
  - エラーケースが曖昧になるのは、業務システムでは怖い

目的は「関数型プログラミングをすること」ではなく
「型、静的検査をより積極的に利用したい」ところにあります

TypeScript だと面倒なところ 😖 … Result 型パズル

```
export const Tag = (input: TagInput): Result<Tag, ValidationError> => {
  const tagId = TagId(input.id)
  const groupId = RestaurantGroupId(input.groupId)
  const label = TagLabel(input.label)
  const icon =
    input.icon && input.iconType ?
      TagIcon({
        symbol: input.icon,
        type: input.iconType,
        color: input.color,
      })
    : ok(NoIcon())
  const sortOrder = FractionalIndex(input.sortOrder)

  const values = Result.combine(tuple(tagId, groupId, label, icon, sortOrder))

  return values.map(([id, groupId, label, icon, sortOrder]) => ({
    ...input,
    id,
    groupId,
    label,
    icon,
    sortOrder,
  }))
}
```

値がだいたい Result に入っているため、複数 Result があると合成してフラットにする必要があり面倒な作業になってくる
(堅牢ではある)