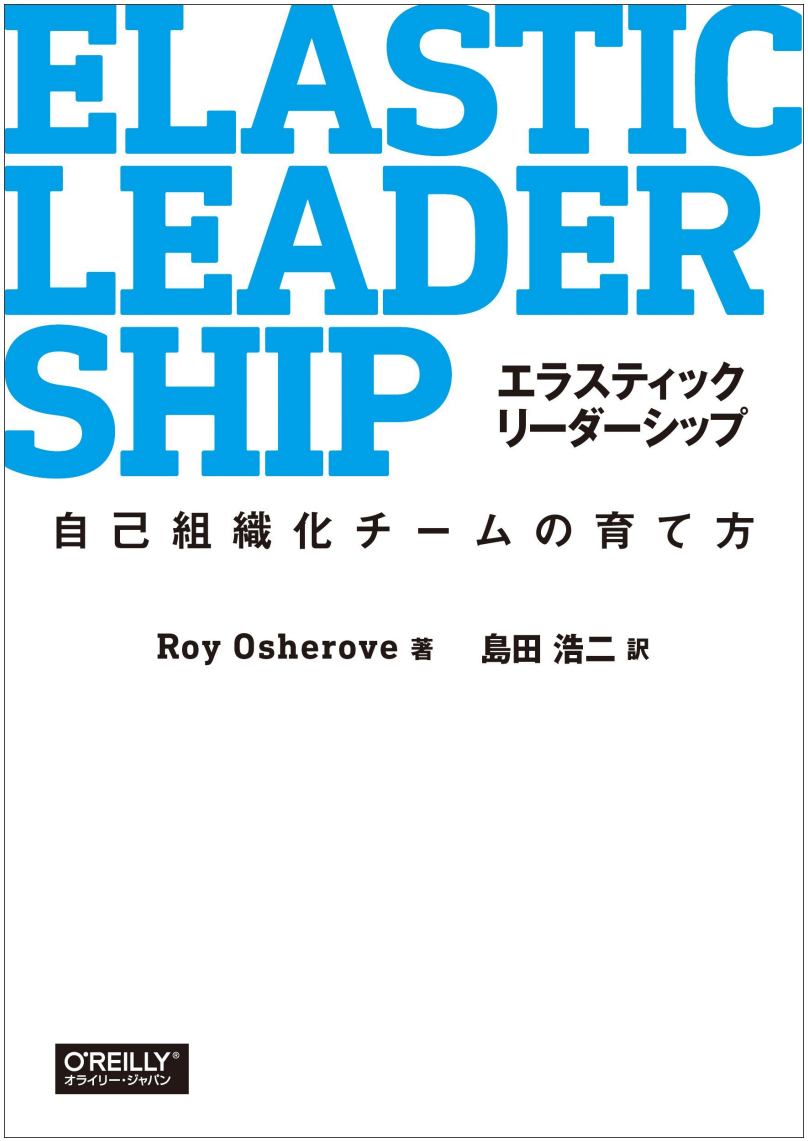
「問題から目を背けず取り組む」
一休の開発チームが6年間で学んだこと

Naoya Ito

株式会社 一休 CTO

2/14 2020

一休.com
こころに贅沢させよう。



“「エンジニアがもっと働きやすくなったら、今の君の会社の問題は本当に解消されるの?」”

“チームが良い状態になれば、自分が良きリーダーになれば物事がうまくいくというのは、悪魔の誘惑です。**プロダクトの問題、事業の問題、技術の問題など、より難易度の高い問題から目を背けることを正当化させる格好の言い訳**なのです。

私たちはこの誘惑に打ち克つ必要があります。”

— エラスティックリーダーシップ 第46章 「大事な問題にフォーカスする」

ハレクラニ沖縄

★★★★½ 4.66 - すばらしい (152件) 沖縄県／恩納村 地図

Halekulani OKINAWA

宿泊プラン | 宿のご紹介 | クチコミ | アクセス | 空室カレンダー | フォトギャラリー

天国の名にもっともふさわしい楽園へ

2019年夏、ハレクラニ沖縄は、半世紀以上にわたり守られてきた沖縄・恩納村の美しい海岸線で、ラグジュアリーの新時代を切り拓きます。
目前に広がるのはエメラルドグリーンに輝く海と白砂のビーチ。
ロビーに到着した瞬間からすべてのゲストを、ハワイで100年以上の歴史を紡いできた「ハレクラニ」ならではのホスピタリティで包み込み、優美に祝福します。

IN 15:00～24:00 OUT 12:00 室数 全360室 ペット 不可 お子様 可

設備・特徴 フィットネス, ルームサービス24H対応, コンビニまで徒歩5分以内, 屋内プール, 屋外プール, エステ施設

基本情報を全てみる

フォトギャラリー

すべてみる

ポイント即時利用料金 OFF ON

2020年2月

3月 >

日	月	火	水	木	金	土
						1 -
2 -	3 -	4 -	5 -	6 -	7 -	8 -
9 -	10 -	11 -	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

Restaurant Sant Pau／ザ・キタノホテル東京

永田町駅 4番出口より徒歩1分　スペイン料理

メニュー・プラン　店舗情報　クチコミ　アクセス　写真

4.47（36件）- 良い　10,000円〜11,999円　20,000円〜

シェフ、カルメ・ルスカィエーダがプロデュースした食べるアート

スペイン、カタルーニャは"進取の土地"といわれ、地中海貿易の拠点として古くから外界との交渉を通じて新しい文化を受け入れてきた地方です。この土地が持つ新進的な精神は、そこで暮らす人々にも強く受け継がれ、現代アートの分野でも前衛的な作風で知られるダリやミロなど世界的に有名なアーティストを数多く生み出してきました。このカタルーニャ地方、サン・ポル・デ・マルは、海の幸、山の幸ともに豊富。この地方では、四季がもたらす自然の恵みに感謝し、その表れとして肉料理にも魚料理にも季節の野菜やフルーツをふんだんに使います。中には肉と魚に季節のフルーツを合わせた、私達日本人にとっては一見イレギュラーな料理もありますが、これも自由なカタルーニャの気風を反映した伝統的な料理法の一つ。決して奇をてらった味わいではなく、素材の味わいを素直に表現した料理は、どれも今までにない新鮮な印象を与えてくれます。東京店についても、現地からスタッフを招聘し、自社輸入により入手する現地の食材を用いることにより、この繊細で芸術的な一皿を完璧に再現し、本物の「SANT PAU」の料理をご提供いたします。

ランチ　ディナー

2020年 2月

日	月	火	水	木	金	土
						1
2	3	4	5	6	7	8
9	10	11	12 —	13 ○	14 ○	15 ○
16 ○	17 ○	18 ○	19 ○	20 ○	21 ○	22 ○
23 ○	24 ○	25 ○	26 ○	27 ○	28 ○	29 ○

特別にプロデュース

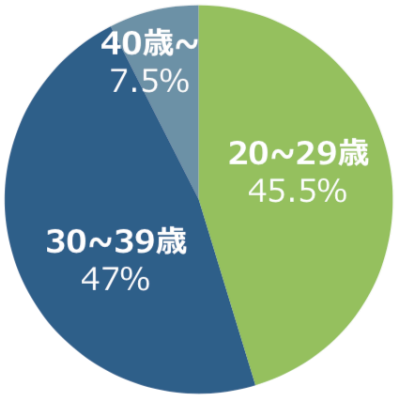
四季の変化を繊細に

従業員 400名、うちエンジニアは全体の2割程度

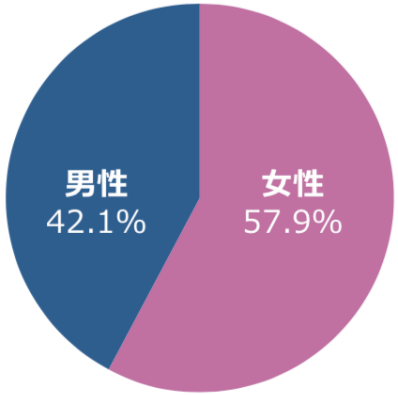
一休.com 社員データ

20代、30代が中心でエンジニアは全体の２割程度

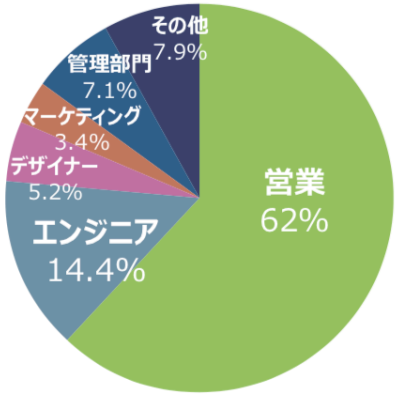
平均年齢



男女比



職種



一休.com
こころに贅沢させよう。

TypeScript

Vue.js

Node.js

Kubernetes

Apache Airflow

Python 3

Linux

GitHub

CircleCI

AppVeyor

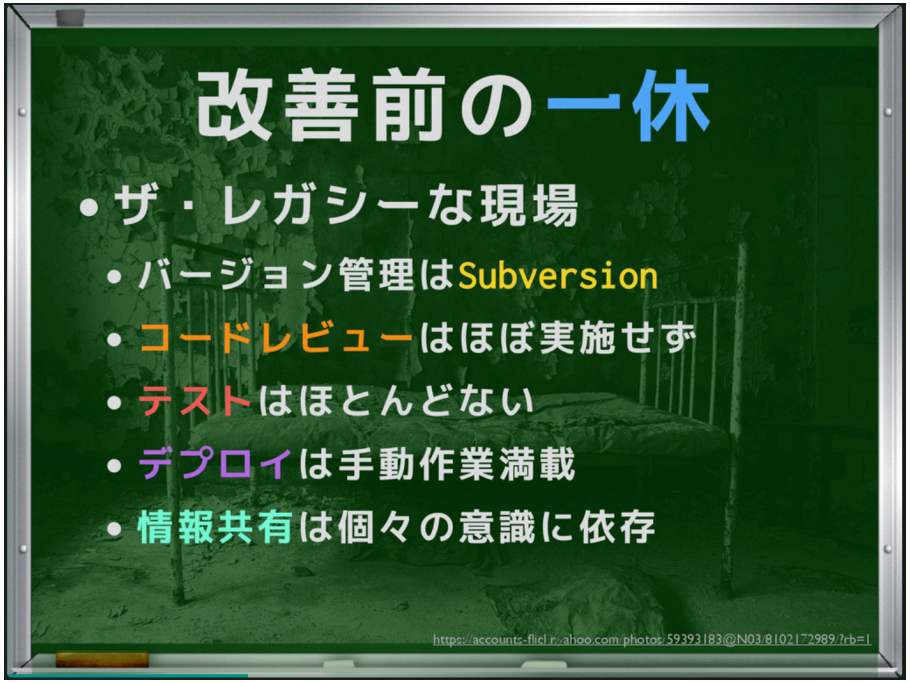
Datadog

遡ること6年前・・・

当時抱えていた課題

- ユーザへ価値を提供するスピードが低下していた
- 品質低下 / 開発効率低下 / 運用負荷 / 権限分掌
  - サービス開発/運用に追われる日々で、自動化をはじめとした運用改善や新しい技術・サービスの導入が追いつかない
  - 運用負荷が徐々に上がり、段々と開発速度が低下
  - プロパーよりもパートナーに開発ナレッジが溜まり、技術が空洞化
  - DevとOpsの権限分掌が進み「Devができないこと」が増加

一休.com



一休での開発における改善の取組み

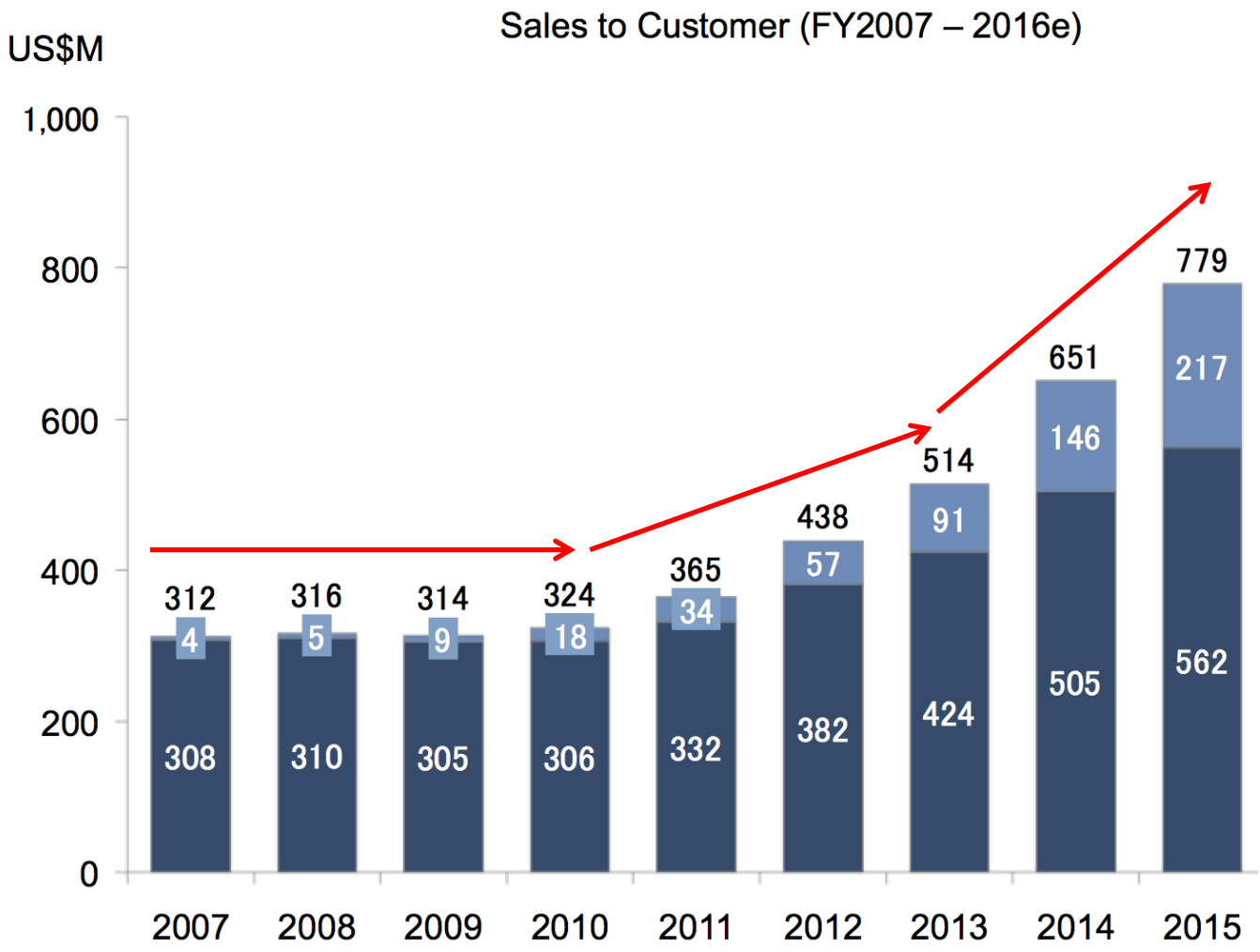
https://speakerdeck.com/kensuketanaka/devops-at-ikyu

一休.com
こころに贅沢させよう。

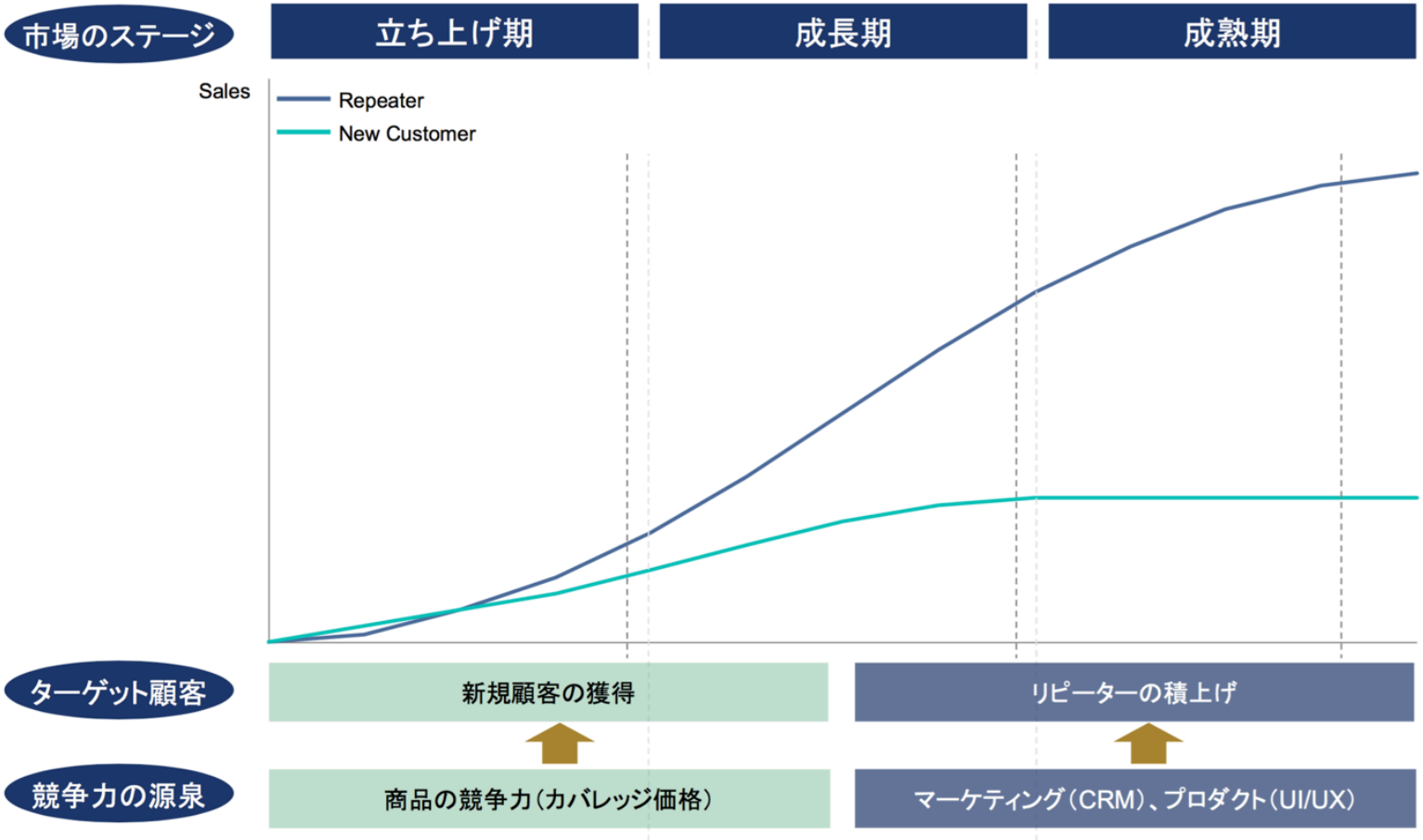
お話したいこと

- ep.1 開発組織構造の転換
  - 働きやすさではなく、事業理解から、開発組織の構造を考えること
- ep.2 レガシーコードからの脱却
  - 複雑で困難な問題に斬り込むリーダーシップ
  - 自分たちに合った、バランスの取れた技術選択の大切さ
- ep.3 基幹システム刷新プロジェクト
  - 問題の解決こそを第一目的としたチームビルディング

ep1. 開発組織構造の転換



市場の成熟と競争軸の変化



市場が成熟するに従い、競争軸が大きく変わる

- 市場の立ち上げ期は商品カバレッジが重要
  - 最初は Amazon は本のみを売っていた。生活用品は他で買う
  - ゲームを売るようになり、ゲームが欲しいときも Amazon に行くようになる
  - 新しい商品を棚に並べると、新しいお客さんが来る

- 成熟してくると商品がコモディティになり、体験が重要になる
  - 市場が成熟するにつれて、Amazon で生活用品が買えるように。他EC では逆に本やゲームが買えるように
    - 「買いやすくて、すぐに届く」(体験) → Amazon を使う
    - ポイントが溜まりやすいから (体験) → ポイントが溜まるサービスの方を使う
  - より使い易いところに、リピーターが集まる

SoR と SoE

	System of Record	System of Engagement
性向 / 適合	・安定性重視 ・予測可能業務 ・リスクを抑えて安全運転 ・要件を事前に明確化	・ユーザーインタフェース品質、開発速度重視 ・探索型業務 ・スピード重視で運転 ・トライ & エラー、プロトタイピング
システム領域	・バックエンド ・認証、在庫管理、決済、個人情報保護	・フロントエンド、スマートフォンアプリ ・UI、CRM、ダイレクトメール
開発手法	・比較的ウォーターフォール寄り ・開発前に要件を明確に定義 ・品質の確保を優先	・比較的アジャイル、リーン寄り ・トライ&エラーで正しい問題・論点を探り当てる ・迅速なリリースを優先
プロダクトマネジメント	・トップダウン寄り ・「正しい仕様」に基づいたプロジェクト管理 ・経営や開発部門が主導することが多い	・ボトムアップ寄り ・「正しい仕様」は存在しない ・マーケティング、デザインが主導することも
ケイパビリティ	・要件定義力、調整力、プロジェクトマネジメント ・SI のみなさんが得意そう	・ユーザー視点、デザイン思考、アナリティクス ・Web 系が得意そう

かつての一休 ・・・ SoR な開発ばかりやっていて伸び悩んだ

- 初期市場では予約できるホテルを増やすことで、新規会員が増えた
  - ホテル向けの開発 ・・・ 在庫管理、決済処理などバックエンド。SoR
  - 当初の成長のレバーはこの領域だった

- 市場が成熟してきて、成長領域が変化しているのに気づいてなかった
  - 成熟市場の成長のレバー ・・・ UI/UX やマーケティング
  - 使い易さ、CRM ・・・ ユーザーエンゲージメント。SoE

かつての一休 ••• 機能別組織

営業

マーケティング

デザイン

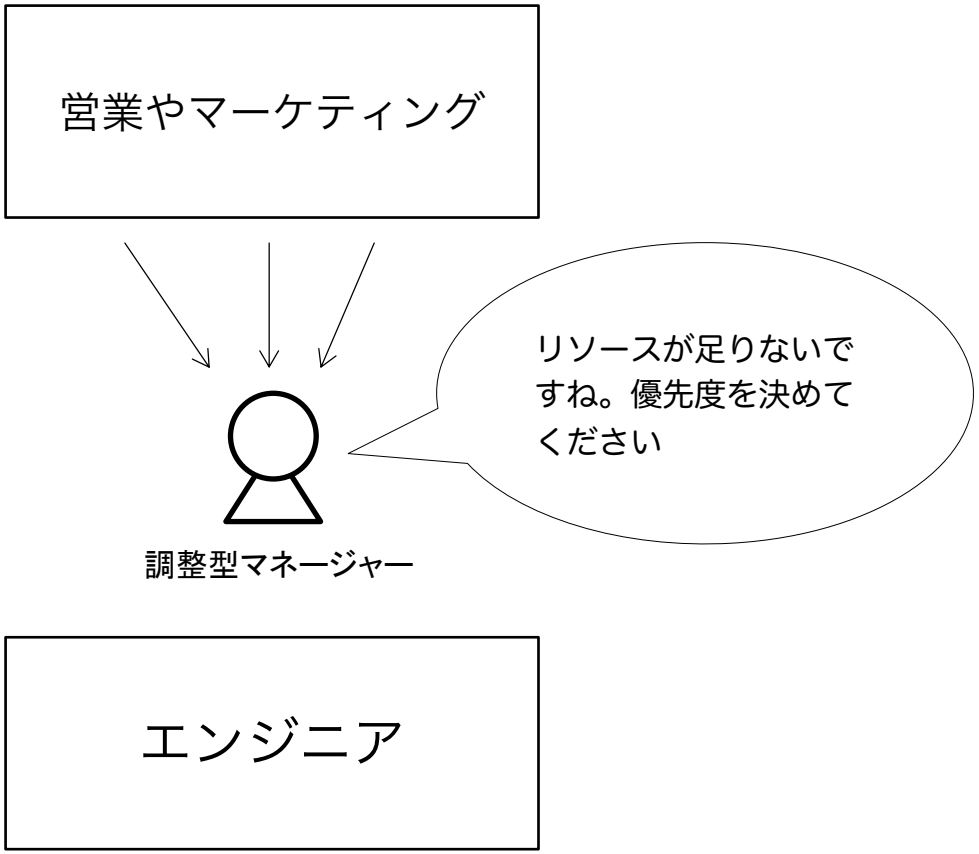
システム (エンジニア)

機能別組織の Pros/Cons

- Pros ・・・ マネジメントが楽
  - エンジニアはエンジニアのボス、デザイナーはデザイナーのボス
  - ボスは自分の得意領域だけみていればいいので、楽
  - 組織的な枠組みが明確

- Cons ・・・ デザイナーやエンジニアが共有財産化し、お役所仕事に
  - 「リソースが足りないのでできません」「それに着手できるのは○ヶ月後ですね」「要件が明確になってないと」・・・

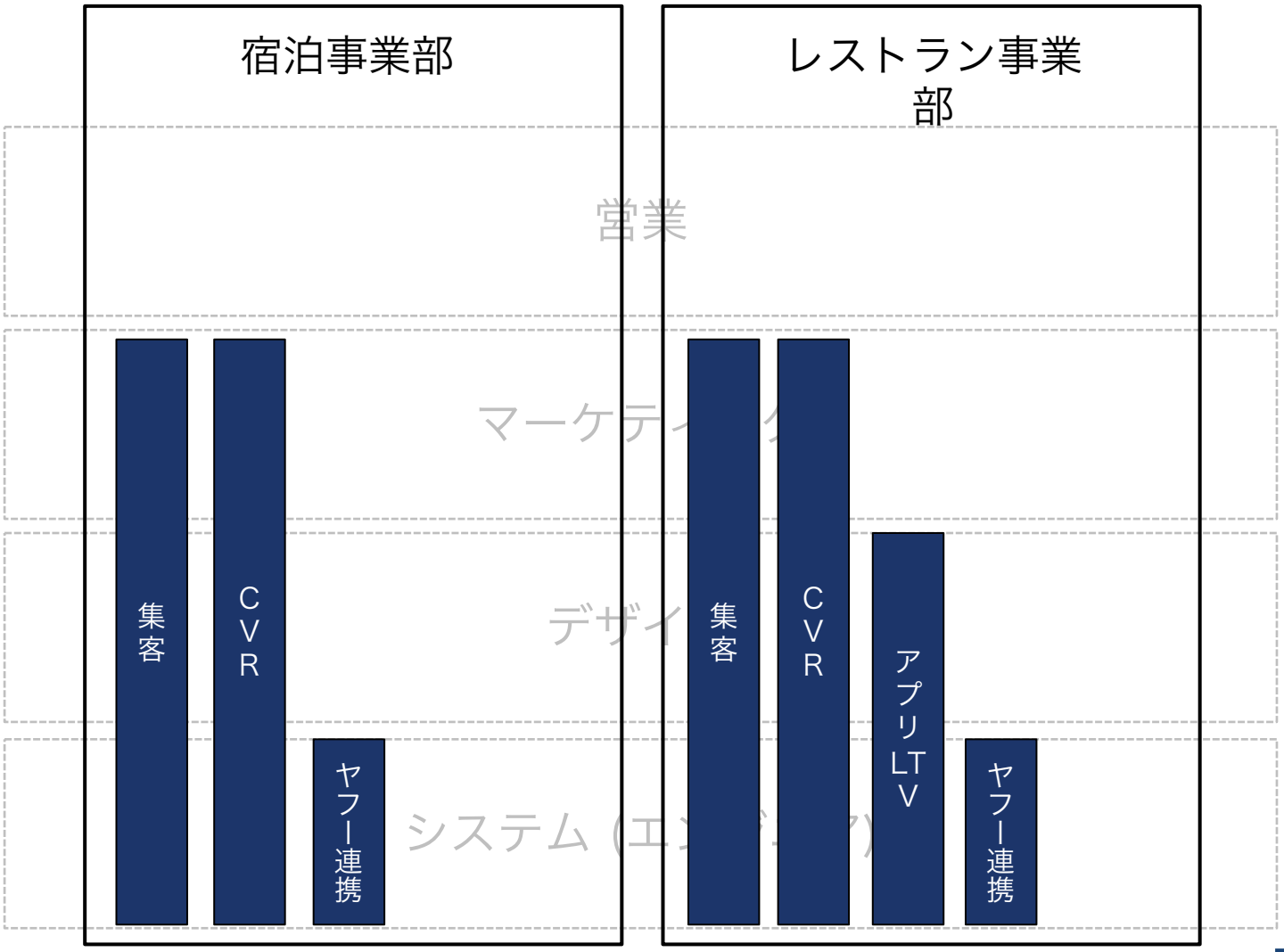
機能別組織では調整型マネージャーがボトルネックになる



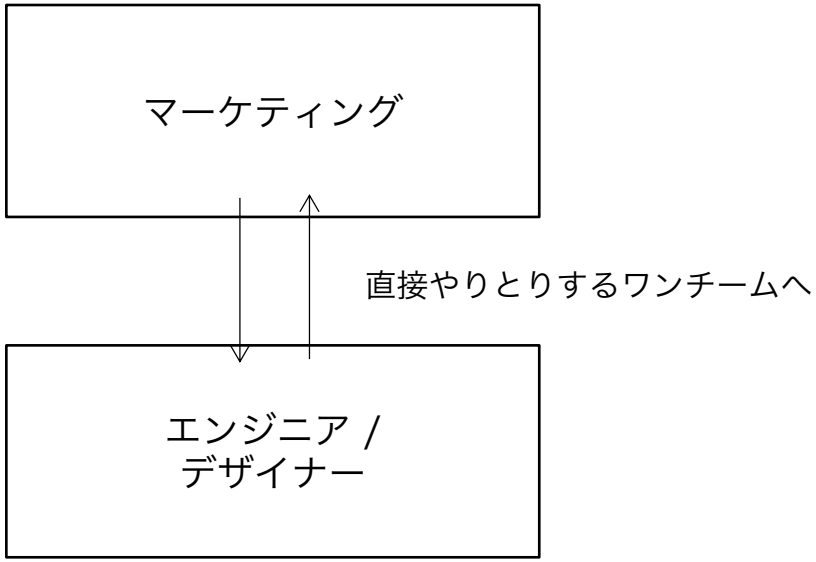
本人はよかれとやっているが、組織を越えたオーバーラップや個人のオーバーアチーブメントが起こる可能生を奪ってしまう。

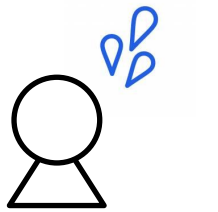
組織全体の生産性が、調整型マネージャーのキャパシティで頭打ちになる。チームが成果にコミットしづらい

目的型組織に転換。職能を跨いだ、注力領域に適応した小さなユニットに



調整型マネージャーの関与を減らした





調整型マネージャー

個々のエンジニアがビジネス課題に直接向き合うので、
プロダクトオーナーシップ (当事者意識) が醸成される

営業主体 (SoR) からマーケティング主体 (SoE) の戦略に切り替え業績が伸びる

- 「目的型組織」は単にミッションごとにチームを割ったわけではない
- **デザイナーやエンジニアが働きやすくなることを目的にしたわけでもない**
- 「事業としてもっとも重要な領域を、素早く、高品質に開発できること」を中心に組み立てた
  - 当然の帰結として、業績は伸びる

“事業をスケールさせるためには、まず組織とプロダクトの設計が相互に影響し合うことを理解する。その上で**事業構造の理解から、より素早く改善すべきレバレッジが効くポイントを見出し、その改善が最も素早く進む組織とプロダクトのアーキテクチャを同時に設計する。**”

— cf: CTOの大切な役割の一つ https://note.com/y_matsuwitter/n/na346c75028c0

(DMM.com CTO 松本さん)

ep2. レガシーコードからの脱却

6年かけて、まずまずヘルシーな状態になった

- 最初の2年間 ･･･一休に技術顧問として関わる
  - Git および GitHub の導入。Slack の導入。コードレビュー ･･･
  - CI / CD ･･･ デプロイ自動化、E2Eテスト自動化

- 4年前 ･･･ ここから CTO
  - フロントエンド開発をモダナイズ ･･･ ビルドパイプライン / Vue.js + TypeScript
  - オンプレミスのインフラを AWS へ移行

- 3年前
  - 一休レストランのレガシーアーキテクチャ改善 ･･･ Classic ASP から Python + Nuxt への移行

- 直近
  - 一休.com (ホテル予約) のレガシーアーキテクチャ改善 ･･･ .NET から Go + Nuxt への移行
  - インフラを k8s (Amazon EKS) でコンテナクラスタ化完了

詳しくは・・・

https://speakerdeck.com/naoya/xiu-falsexian-zai-to-kokomadefalsedao-falseri

一休.com
こころに贅沢させよう。

今思えばこういう順番だった

- 初期
  - (1) 開発者のコミュニケーション改善 ・・・ GitHub、Slack、コードレビュー ・・・
  - (2) 日頃の開発オペレーションのコスト削減 ・・・ リリース・テスト作業の自動化
- 中期 ・・・ 周辺領域から中心に斬り込む
  - (3) 開発に使う道具の刷新 ・・・ フロントエンド開発、AWS
- いざ本丸 ・・・ 「業務ロジック」の巣窟へ
  - (4) 一休レストランのサーバーサイドのレガシーアーキクチャ改修
  - (5) より業務ロジックの多い、一休.com の同上

今思えばこういう順番だった

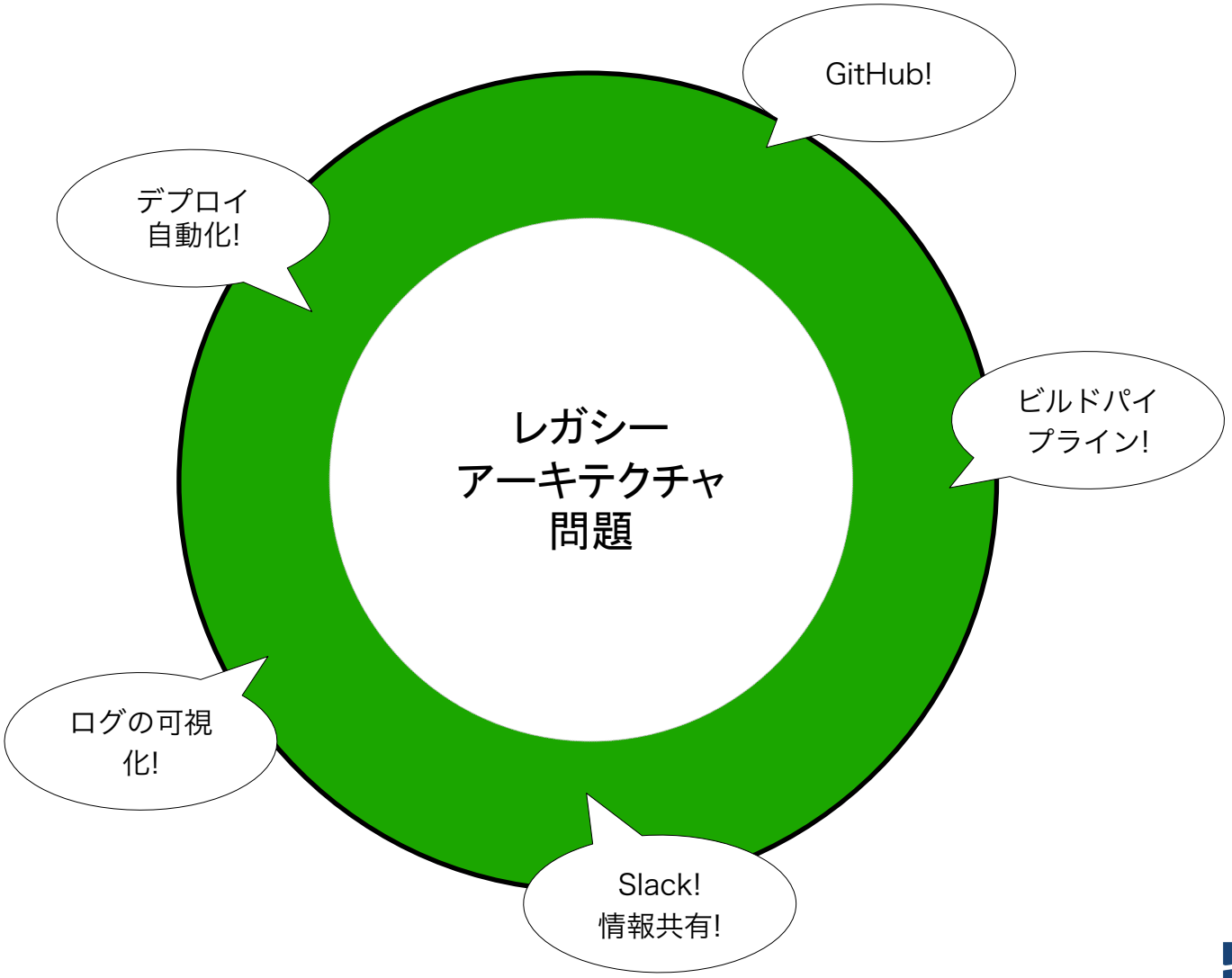
難易度

- 初期
  - (1) 開発者のコミュニケーション改善 ･･･ GitHub、Slack、コードレビュー ･･･
  - (2) 日頃の開発オペレーションのコスト削減 ･･･ リリース・テスト作業の自動化
- 中期 ･･･ 周辺領域から中心に斬り込む
  - (3) 開発に使う道具の刷新 ･･･ フロントエンド開発、AWS
- いざ本丸 ･･･ 「業務ロジック」の巣窟へ
  - (4) 一休レストランのサーバーサイドのレガシーアーキクチャ改修
  - (5) より業務ロジックの多い、一休.com の同上

何でこの順番になったのか

- 本丸がレガシーアーキテクチャであることは最初から分かってはいた
- 理由はふたつ
  - (1) いきなり本丸に斬り込んでもやれる気が全くしなかった
  - (2) ある程度時間が経ったら**感覚が麻痺して、大きな問題から目を背けて着手が遅れた**

ボトムアップだけでは一番大きな問題が解決されていかなかった



良くなかったアプローチ

- 組織をマネジメントし、活力を上げれば、レガシー問題もやがて (誰かが立ち上がり?) 解決するだろうと思っていたことがあった
- そんなことは決して起こらなかった
  - 今思えば、戦術でもなんでもなく、ただの「願望」だった
  - 組織改善に執着することが、大きな問題から目を背ける言い訳になっていた

最初の一歩を、トップダウンで風穴を開ける必要があった

- 組織的な改善アプローチに頼らず、技術的問題に正面から取り組む必要
- CTO == テックリードの長として

レガシーコード改善をどう進めるか

- いろいろな戦略があり得る
  - ビジネスをしばらく停めて、ビッグリライト?
  - 継続的リファクタリング?
  - 徐々にマイクロサービス化しレガシーを切り離す?
  - ・・・

具体的にどう始めたか

- まずは自分ひとりで「問題の理解」から始めた
  - 既存の実装を**とにかく、ひたすら読み**、構造を分析し、課題を明らかに
  - アーキテクチャの落としどころを描くためにプロトタイプを作る
    - どんな風に書くかのプロトタイプ。それを基に徐々にアーキテクチャを固める

自社の過去の反省から

- 過去に主力事業の大規模なシステムリニューアル
  - ビッグリライト
  - リニューアルに対する期待が高まりすぎて、大小あらゆる要件がリニューアルに寄せられる
  - プロジェクト規模が大きくなりすぎ収束までにとても苦労したらしい
    - アーキテクチャは必要よりも複雑なものになってしまった

- 反省から
  - 何の問題を解決したいのか、そのフォーカスをはっきりさせる
  - 関心事を明確にしスコープを小さく。再設計に KISS 原則を忘れない

パターン (PofEAA) と比較しながら構造や課題を明らかにする

一休の現状 ･･･ シンプルで分かり易いパターンの組み合わせ

http://d.hatena.ne.jp/asakichy/20120611/1339366061

プレゼンテーション層はページコントローラ

- サーバーページ (.asp) 自体が入力コントローラになるパターン
- あるリソースにアクセスするURLが専用のパスになる。シンプルでわかりやすいが他コントローラとの処理の共通化に難がある
- 一休では View と Controller が分かれていない

ドメイン層はトランザクションスクリプト

- ビジネスロジックひとつにつき専用のスクリプトを用いるパターン
- シンプルなので分かり易いが、大きなシステムでは他の業務ロジックとの重複を誘発するのが欠点
- 一休ではデータソース層と十分に分離されていない

データソース層は行データゲートウェイ (に近い)

- 行データGW ではテーブルのレコードとオブジェクトが一対一で対応
- RDBMS の構造にデータモデルが強く影響されすぎるのが欠点
- 一休では
  - RDBMSの構造がベストな状態ではないため、その構造にアプリケーションが強く影響されているのが難
  - データオブジェクトに階層がないため、一度に必要なデータをJOIN で引いて複雑な SQL を発行。可読性が低い

15

ドメイン層の課題 ･･･ (1) コードの重複 (2) SQLの混在 (3) 複雑な SQL

- トランザクションスクリプトであるため、共通処理がうまく括り出せていない箇所が散見される
- ドメイン層とデータソース層が分離できていないため、ドメイン層に直接 SQL が記述されている
- JOIN を利用して一度に DB からデータを引こうとするあまり、参照系クエリが複雑になっている

トランザクションスクリプトは分かり易い (かついきなりドメインモデルに移行は難しい) ので維持しながら
・ドメイン層とデータソース層の分離
・データソース層の SQL の書き方の見直し
・ドメインオブジェクトを導入し、ある程度のロジックはドメインオブジェクト側に実装
を行って解決したい

30

パターンにはそれぞれ長所・短所がある。パターンを基準に分類していくと自分たちのシステムの構造的な長所や短所を言語化しやすかった

レイヤごとに問題を理解しながら言語化していき・・・

現状の View の問題点 (1)

```
'// HTMLタグ開始
Call writeUsrl0Head()

'// OGP Metaタグ出力
Call writeOgpMetaData(tmpStrMetaTitle, tmpStrMetaDescription, tmpStrMetaUrl, tmpStrMetaOgImage, tm
rMetaOgType, tmpStrMetaTwitterCard)
%>

<link rel="stylesheet" href="/rsStyle/guide/rsPlanInfo.css<%= CSS_VERSION %>" type="text/css">
<link rel="stylesheet" href="/rsStyle/rsCommu.css<%= CSS_VERSION %>" type="text/css">
<link rel="stylesheet" href="/rsStyle/guide/rsGuideCoupon.css<%= CSS_VERSION %>" type="text/css">
</head><%

'// 通常遷移時の表示
 IF blnLimitedDispFlg = False THEN
%>
<body><%

    '// ヘッダーメニュー
    Call writeUsrHeadMenu()
```

- 繰り返し表れるHTML構造 (例: ヘッダやフッタ) を定義する手段が、VBScript での関数 (命令的) しかない
  - 結果、宣言的な HTML と命令的な処理が混在
- かつ、コードのスコープがグローバルでありあらゆる処理が呼べるため View と Controller が密結合している

現状の Controller の問題点

```
'// mパラメータの取得
Call getInParam()

'// m入力パラメータチェックと初期値設定
strErrMsg = checkInParam()
IF (isErr(strErrMsg)) = True THEN
  Call endProc()
  Call dispUserErr(strPanelID, strErrMsg)
END IF

'// m画面表示判定
blnLimitedDispFlg = fncIF(objClsInput.getValue("dispType") = "1", True, False)
IF blnLimitedDispFlg = False THEN
  IF cmnDispJudg(LTD_KBN_PRE_PAUSE, objClsInput.getValue("rstId")) = False THEN
    Call endProc()

    '// SEO対策のため、営業停止中だった場合404ページに飛ばす
    Response.Redirect "/rsNotFound.htm"
  ELSE
    '// 店舗ステータスが公開中の場合
    blnIsOpenFlg = cmnDispJudg(LTD_KBN_OPEN, strRstId)
  END IF
ELSE
  IF cmnDispJudg(LTD_KBN_PRE_STOP, objClsInput.getValue("rstId")) = False THEN
    Call endProc()

    '// SEO対策のため、営業停止中だった場合404ページに飛ばす
    Response.Redirect "/rsNotFound.htm"
  END IF
END IF

'// m携帯ユーザーチェック
IF IsMbAvailableBrowser() THEN
  strMobileUrl = rsDomain & "/rsMb/rsMbHelp/rsMbClosed.htm"
  Call endProc()
  Response.Redirect strMobileUrl
  Response.End
END IF
```

スコープがグローバル

グローバルスコープに処理が記述されており、View からそのグローバルスコープにアクセスしている

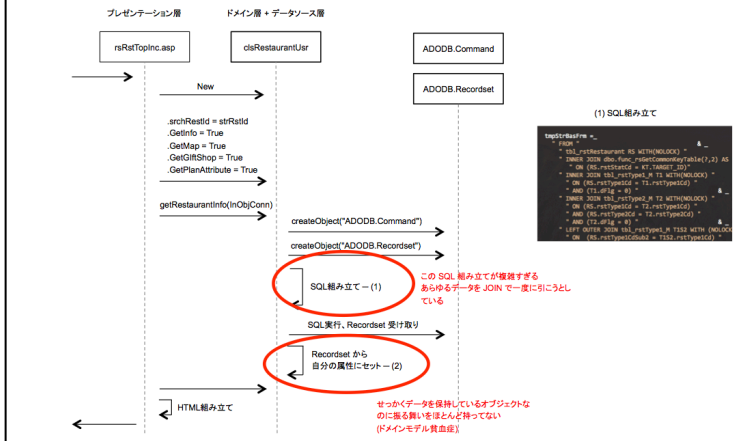
同じ手続きを他コントローラでも繰り返している

他コントローラと同じ手続きを、各コントローラで呼び出している。コピペの温床になっている

ドメイン層の課題 ・・・ (1) コードの重複 (2) SQLの混在 (3) 複雑な SQL

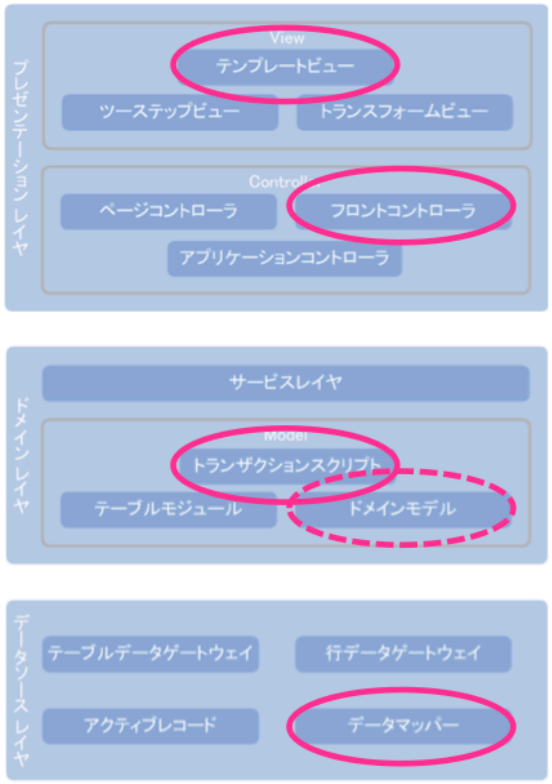
- トランザクションスクリプトであるため、共通処理がうまく括り出せていない箇所が散見される
- ドメイン層とデータソース層が分離できていないため、ドメイン層に直接 SQL が記述されている
- JOIN を利用して一度に DB からデータを引こうとするあまり、参照系クエリが複雑になっている

トランザクションスクリプトは分かり易い (かついきなりドメインモデルに移行は難しい) ので維持しながら
・ドメイン層とデータソース層の分離
・データソース層の SQL の書き方の見直し
・ドメインオブジェクトを導入し、ある程度のロジックはドメインオブジェクト側に実装
を行って解決したい



現状が言語化できたら To Be も考えてみる

改善例: 各レイヤーのアーキテクチャパターンをより構造的なものに変更



http://d.hatena.ne.jp/asakichy/20120611/1339366061

Webフレームワークとテンプレートエンジンを導入、C と V を分離

- テンプレートエンジンにより、テンプレートビューを獲得
- [獲得] より安全で、より機能豊富で、より短く書ける View に

ページコントローラからフロントコントローラへ変更

- 現状フリーダムなコントローラーの処理を構造化
- [獲得] 認証や端末判定などの前後処理を毎回呼び出さなくても良いように → フロー制御だけに集中できる Controller に

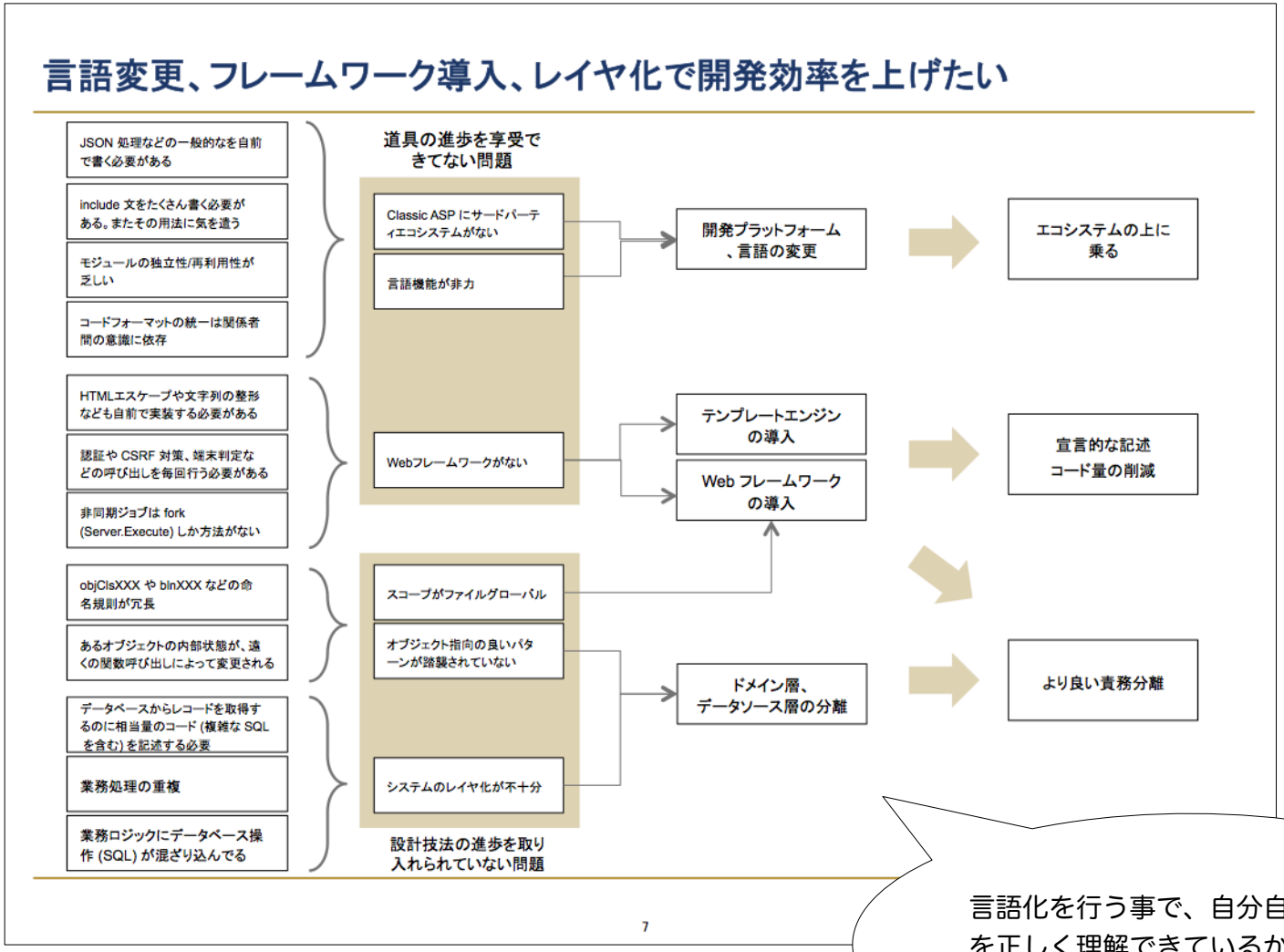
ひとまずトランザクションスクリプトを維持

- ただし、ドメインロジックからデータベース関連処理を分離し、データソース層に移す
- ドメインモデルを導入し基本的なドメインロジックは集約

行データゲートウェイからデータマッパーへ変更 (一案)

- モデルとテーブル構造の乖離をデータマッパーが吸収
- [獲得] テーブル構造を維持したまま、より適切なデータモデルに。データモデルに階層を作り、SQL を単純化する
- [獲得] プレゼンテーション層やドメイン層に必要な共通処理をオブジェクトの振る舞いに集約。適切なスコープを獲得する

言語化を行い、整理する



言語化を行う事で、自分自身が問題を正しく理解できているかを明らかにしていった

問題の理解が進むにつれ現実的な進め方が見えてくる

- レガシーを作っている根本原因はいわゆる「スマートUI」
- ビッグリライトをしなくても少しずつ塗り替えていけそう
  - ページコントローラ + トランザクションスクリプトの組み合わせなのでページ単位での実装の独立性が高い
  - Data Mappter 的な考え方で抽象化することで物理層 (データベース) はそのまま、新しいドメインオブジェクトを導入できそう

一休はどうやることにしたか

- ビジネスを常に優先しながら徐々に塗り替えていく戦略
  - 開発都合をビジネスよりも優先しない ・・・ ビッグリライトは選択肢から消える

- ビジネスの文脈で、大きめの開発が発生するときに、技術基盤ごと入れ換える
  - その開発が始まるまえに、開発に必要な部品を作っておく
    - 例: 認証、API、汎用ライブラリ、データモデル・・・

新しい実装は薄いフレームワーク、薄い実装を目指す

- **× 理想のアーキテクチャ ○ 課題に合わせる**
  - データ構成を強制するフルスタックなものは向いていない
  - もともと設計は複雑ではない。学習コストは低い。学習コストを跳ね上げるのは本末転倒
  - 学習コストの低いプラットフォーム + 薄い実装がベスト

結論、Python 3 + Flask + レイヤードアーキテクチャでいくことにした。
ただし、DDD にはそこまで拘らない

データベースをどうしたか

- 開発都合のビジネスの停止 (新規開発案件の凍結) を行わないのは絶対条件
- データベースはやり直さず、既存のデータベースをそのまま使う
  - 当然、おかしな設計なテーブルはたくさんあるが、そこはやり直さない
  - データベース設計をやり直すなら、コードベースが整った後

テーブル設計の欠点が、モデルに極力影響を与えないようにする

- ORM は採用しない
  - ActiveRecord パターンの ORM が多いが、ActiveRecord はデータベース設計がそのままモデルになってしまう
  - ただのクラスの Repository + Plain Old な Python オブジェクト (dataclass)

- Data Mapper や CQRS の考え方を参考に、テーブル設計の悪さを回避
  - Data Mapper ・・・ 物理層の構造とモデルの間にマッピングの層をひとつ挟むパターン
  - CQRS ・・・ 読み取り (クエリ) のフローと、更新 (コマンド) のフローで分けるパターン

Repository

```python
class RestaurantOperatorRepository(Repository[RestaurantOperator]):
    table = "tbl_rstDealOP"
    fields = ['*']

    @classmethod
    def to_model(cls, row: Dict[str, Any]) -> RestaurantOperator:
        return RestaurantOperator(
            restaurant_id=row["rstId"],
            id=row["rstOpeId"],
            name=row["rstOpeName"],

            password=Password(
                hashed_value=row["rstPswd"],
                changed_at=row["rstHdate"]
            ),

            read_permission=OperatorPermission(row["rstRlvl"]),
            write_permission=OperatorPermission(row["rstWlvl"]),

            enabled=row["rstTflg"],
            admin=row["rstPflg"],

            users_updated=row["hRecs"],  # dmemo には「ログイン記録」とあるが実際はただの変更履歴
            updated_at=row["hDate"],
            deleted=row["rstDflg"],
            api_client=row["isApi"],
        )

    @classmethod
    def find_by_restaurant_id_and_operator_id(cls,
                                              db: DataBase,
                                              restaurant_id: int,
                                              operator_id: str,
                                              include_deleted: bool = False
                                              ) -> Optional[RestaurantOperator]:
        return cls.find(
            db,
            {
                "rstId": restaurant_id,
                "rstOpeId": operator_id,
                "rstDflg": None if include_deleted else False
            }
        )

    @classmethod
    def get_by_restaurant_id_and_operator_id(cls,
                                             db: DataBase,
                                             restaurant_id: int,
                                             operator_id: str) -> RestaurantOperator:
        operator = cls.find_by_restaurant_id_and_operator_id(db, restaurant_id, operator_id)
```

DBレコードからオブジェクトはマニュアルで作る。
自動でのマッピングは行わない。
列名がいけてないとかはここで変換する

これは実際には SQL
SQL の条件を Python のデータ構造で記述できるライブラリを自作し、ORM がない欠点をカバー

モデル

Plain な Python オブジェクト
(dataclass)
DB の構造に依存していない

```python
@dataclass
@partially_loadable
class RestaurantOperator:
    """
    施設（店舗）オペレータモデル
    """
    restaurant_id: int  # 店舗ID
    id: str  # オペレーターID
    name: str  # オペレーター名（個人情報）

    password: Password

    read_permission: OperatorPermission  # 読み取り権限
    write_permission: OperatorPermission  # 書き込み権限

    enabled: bool  # 登録状況（有効/無効）
    admin: bool  # 管理者フラグ

    api_client: bool = False

    users_updated: Optional[str] = None
    updated_at: Optional[datetime.datetime] = None
    deleted: bool = False

    restaurant: Restaurant = Loadable[Restaurant].field()

    def is_admin(self) -> bool:
        return self.admin is True

    def is_api_client(self) -> bool:
        return self.api_client

    def check_password(self, raw_password: str) -> bool:
        return self.password.verify(raw_password)

    def timedelta_since_password_changed(self, now: datetime.datetime = None) -> datetime.timedelta:
        return self.password.timedelta_since_changed(now)

    def password_expiration_datetime(self) -> datetime.datetime:
        return self.password.expiration_datetime()
```

読み取り (クエリ) モデルと割りきることで、テーブル設計の問題を回避

```python
class RestaurantRepository(Repository[Restaurant]):
    table = """tbl_rstRestaurant RS
               INNER JOIN tbl_rstRestExt R1 ON RS.rstId = R1.rstId
               INNER JOIN tbl_rstRestExt2 R2 ON RS.rstId = R2.rstId
            """
    fields = ["*"]

    @classmethod
    def to_model(cls, row: Dict[str, Any]) -> Restaurant:
        """
        行データから Resetaurant オブジェクトを構築する Factory Method
        """

        return Restaurant(
            #
            # tbl_rstRestaurant
            #
            id=row['rstId'],
            name=row['rstName'],
            name_alt=row['rstNameKn'],
            name_kana=row['rstNameKana'],

            display_type_name=row['dispRstTypeName'],
            type_code_pairs=(
                (row['rstType1Cd'], row['rstType2Cd']),
                (row['rstType1CdSub2'], row['rstType2CdSub2']),
```

3つに分かれていたテーブルに対し、
JOIN した結果からモデルを構築
モデルは読み取りのみに利用される

更新 (コマンド) モデル

```python
@dataclass
class Reservation(CommandModel):
    reservation_summary: ReservationSummary
    reservation_detail: ReservationDetail
    plan_option_orders: List[PlanOptionOrder]
    child_menu_orders: List[ChildMenuOrder]

    def __post_init__(self) -> None:
        super().__init__()

    def hide_for_member(self) -> None:
        self.reservation_summary.hide_for_member()
        self.reservation_detail.hide_for_member()

        for plan_option_order in self.plan_option_orders:
            plan_option_order.hide_for_member()

        for child_menu_order in self.child_menu_orders:
            child_menu_order.hide_for_member()


class ReservationCommandRepository(CommandRepository[Reservation]):

    @classmethod
    def find_by_reservation_id_and_member_id(cls,
                                             db: DataBase,
                                             reservation_id: str,
```

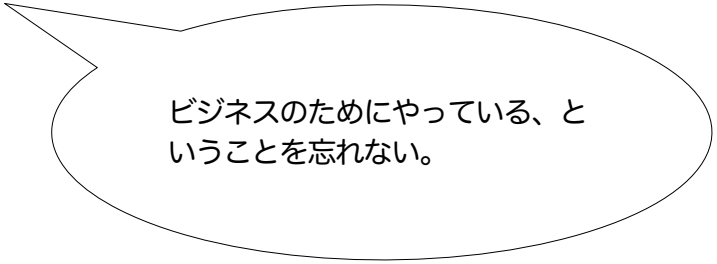
複数のコマンドモデルを集約し、トランザクション境界を表現。
クエリモデルとは別に実装。
クエリはラピッドに開発し、コマンドは慎重に開発できる

多少の妥協は発生する

- 多少の妥協は発生するが、それは許容する
  - テーブル設計がうまくないために、モデル設計がややいびつになることもある
  - CQRS で悩むポイント ・・・ クエリモデル、コマンドモデルで実装が重複する
    - クエリモデルとコマンドモデルで似たような振るまいがあるとき、DRY にできなかったり・・・
- ビジネスを停めないという制約の中で 80 〜 90点が取れれば良い

目指した考え方

- 必要以上に学習コストを上げすぎない、開発に参画しやすいシステム
  - なるべくシンプルに
  - 教条的になりすぎない



ろうそくの小さな灯火が、大きな火にまで拡がれば勝ち

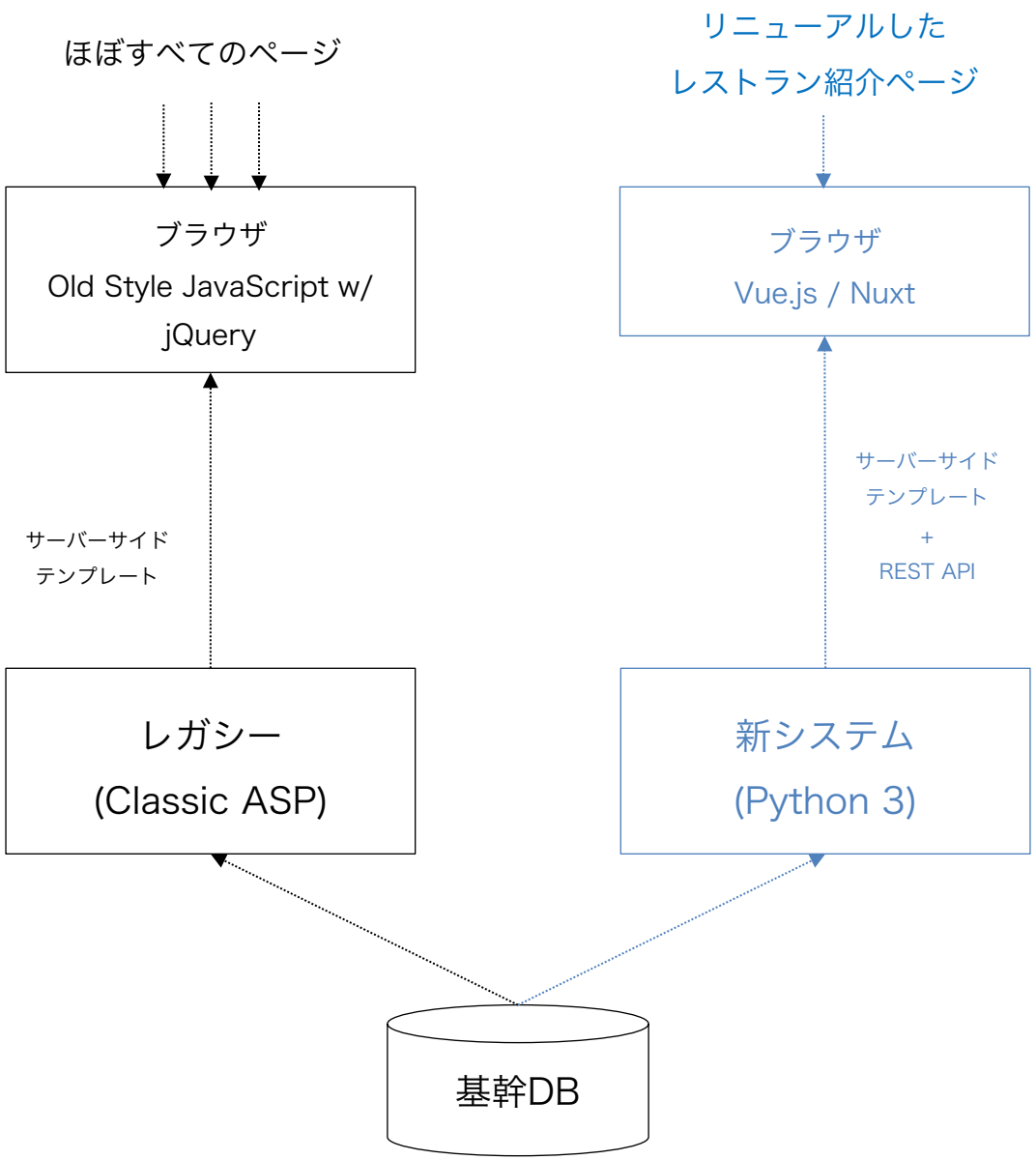
- 最初はひとり (半年ぐらい、一人で作っていた)
  - 課題分析、技術選定、基本的なアーキテクチャ方針
  - 認証、データアクセス、API の枠組み、コアなドメインモデルを実装

- ビジネスで新しいプロジェクトを行う際、新基盤を投入
  - そのプロジェクトのメンバーを巻き込み、ペアプロ
  - 変更に対してはコードレビューで、設計意図の継承を担保する (徹底的に･･･)

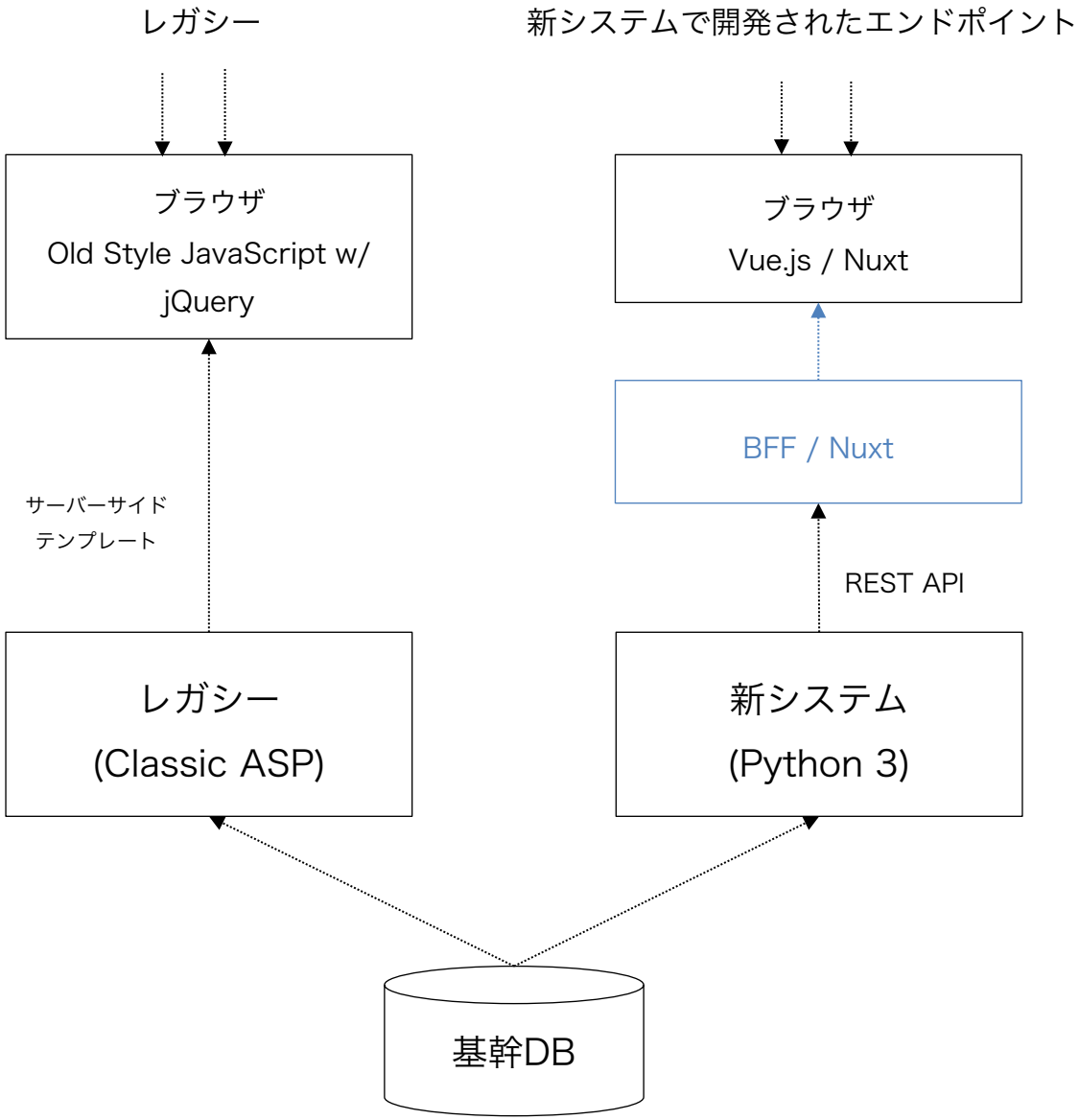
- 同様の活動を行うメンバーを増やす
  - レビューをみっちりやることで、同じレベルで設計、レビューできるメンバーが増えた
  - フロントエンドのテックリードが自然と頭角を現した

- このサイクルを回していたら、いつのまにか自然に新基盤での開発が行われるように

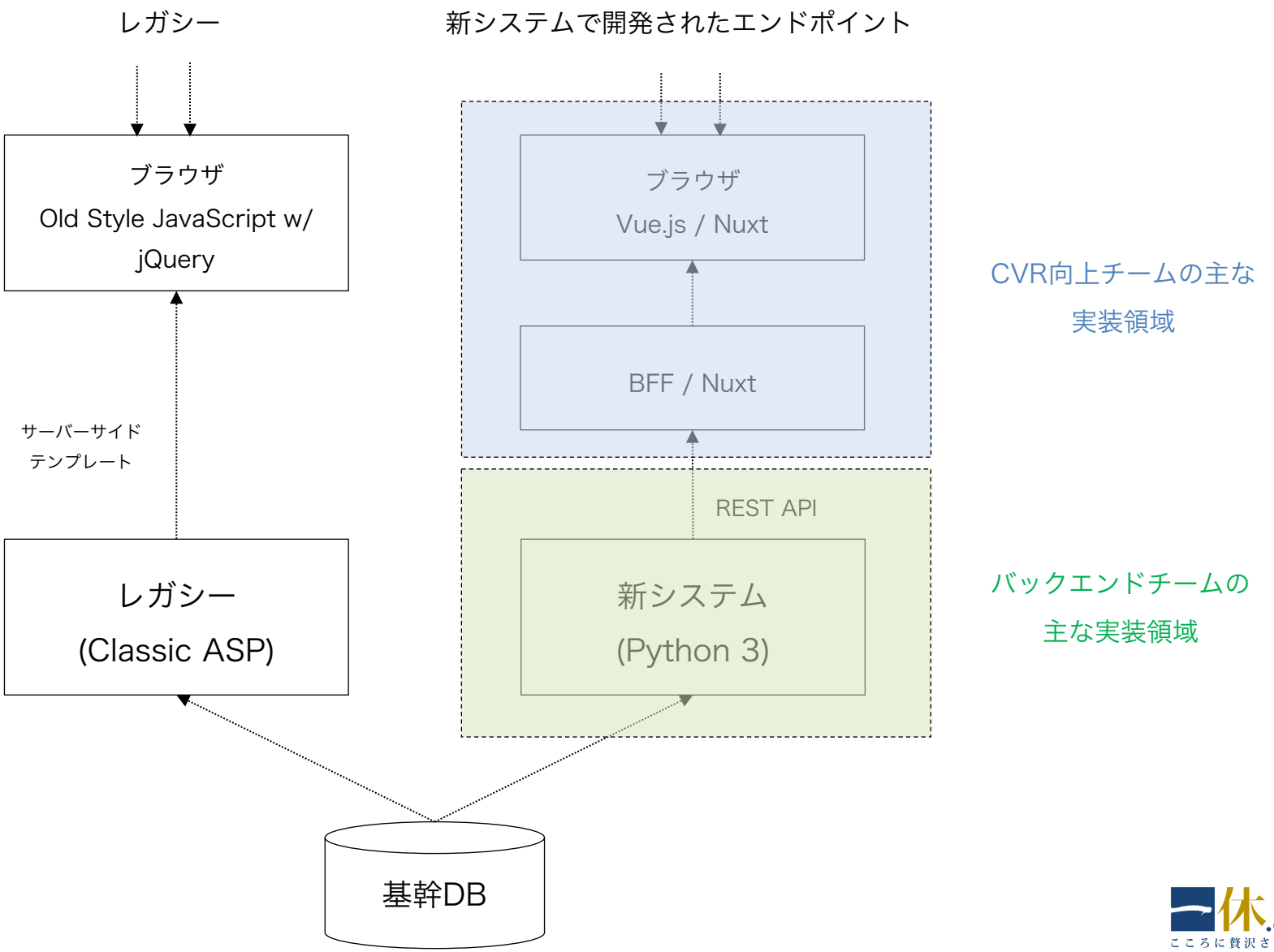
プロジェクト開始当初



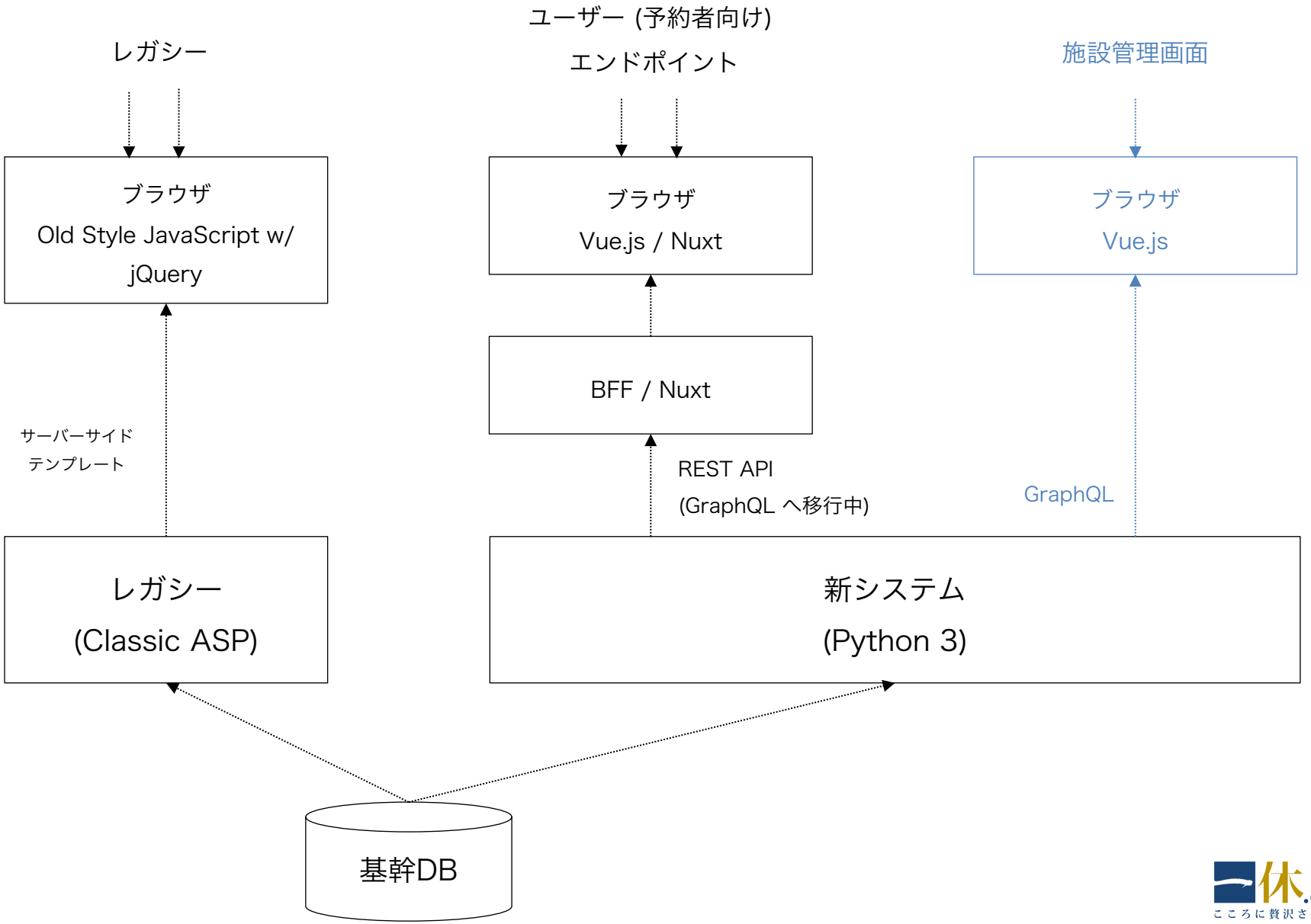
API が出揃ってきたところで BFF / Nuxt が導入された



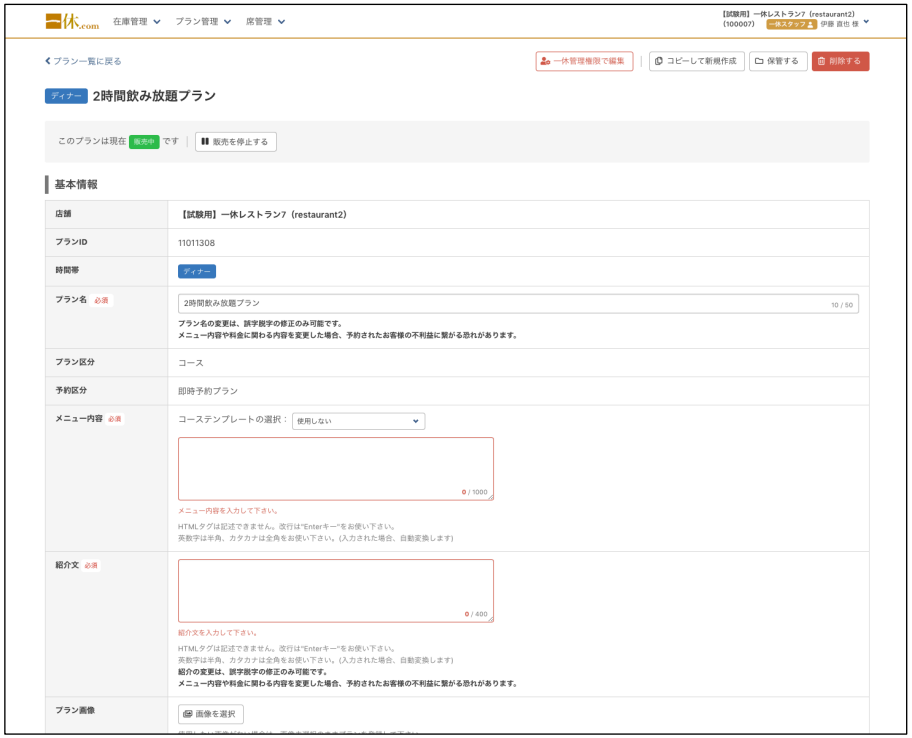
目的型組織の形にあったシステム構成に。より効率が上がった



ユーザー向け、施設管理でフォーカスが異なる。2つの方式を確立

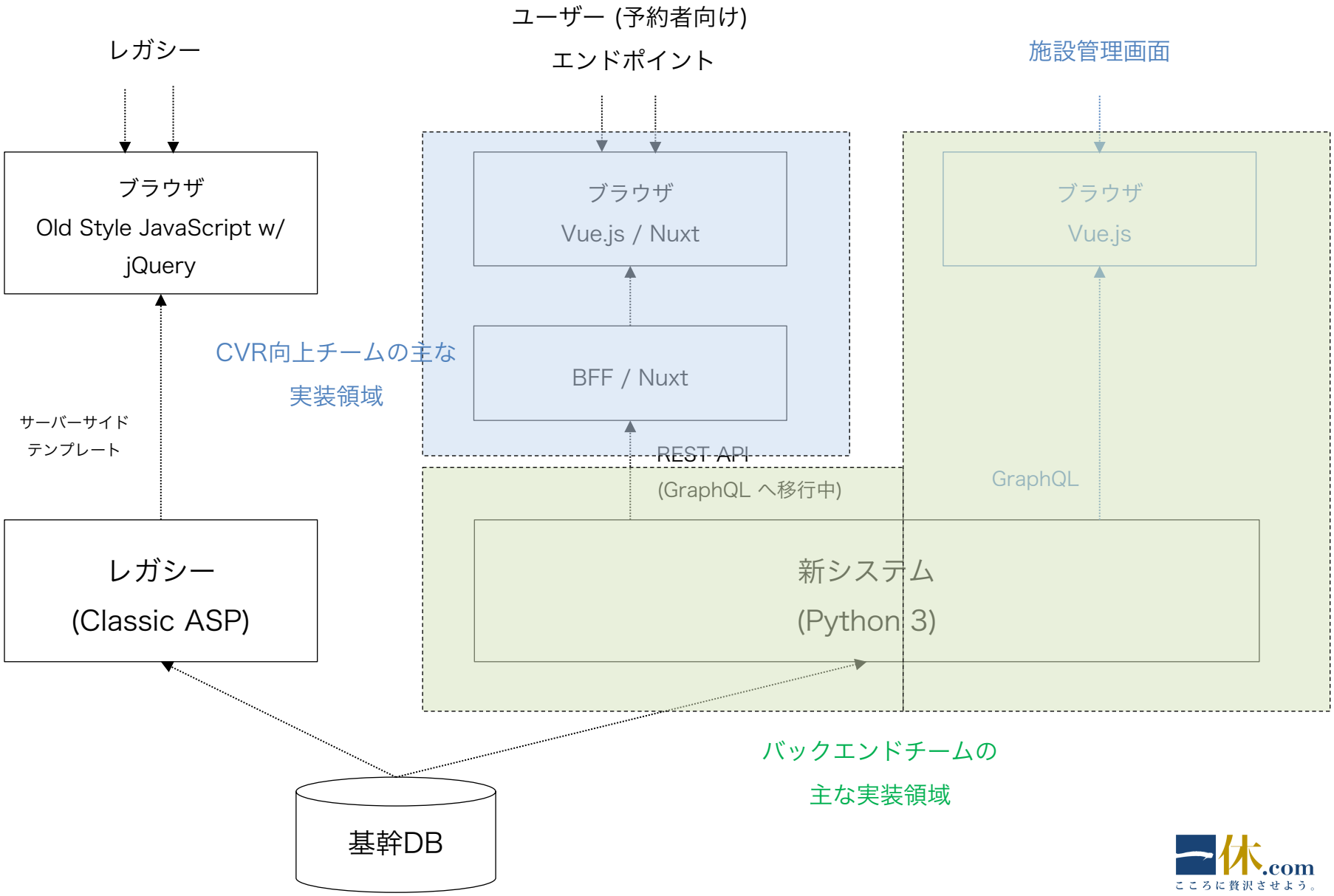


UI こそが事業上重要な箇所 / 業務処理こそ重要な箇所で2通りの作り方



一休.com
こころに贅沢させよう。

フォーカスの異なるチームが共創できるシステム構成に

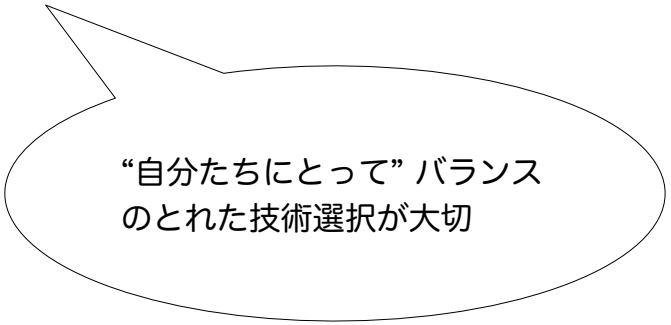


マイクロサービスは、すぐには目指さない

- 現時点の組織規模 (各事業、エンジニアは20名程度) では下手にマイクロサービスにするとオーバーヘッドが大きすぎる
- マイクロサービスで何か問題を解決するのではなく、結果的にマイクロサービスになった･･･が正解と考えている
  - すでにカード決済、ポイント業務などはマイクロサービスになっている

この方式が一般的にも正解か？ いいえ

- いろいろな変数がある
  - 一休の事業構造
  - 一休の組織規模、今のチーム構造
  - どういう順序でビジネスプロジェクトを進めて来たか、これから進めるつもりでいくのか
  - 技術の、その時期のデファクトスタンダード



何がしたかったのか、を改めて考える

- 技術的に完璧な実装 ・・・ False
- レガシーをなくしたい ・・・ False
- 技術的な成長機会 ・・・ True だけど、これはただの願望
- **開発しづらいのをどうにかしたい。ビジネスの実現に回り道したくない ・・・ True**
  - 開発頻度の高い箇所で、それが実現できることが大事 (極論、頻度が低ければ無視してもよい)
  - **「開発しやすい」には「敷居が高すぎない」「意志決定しやすい」も含まれる**
    - ある程度の学習コストで開発に参画しやすい
    - 必要以上の技術力を (全員には) 要求しないでも開発できる
    - ミッションが異なるチームメンバーとの調整が少ない

ep3. 基幹 (在庫) システム刷新プロジェクト

レガシーは、技術だけではない

- 10年前に設計された一休レストランの在庫 (空き予約枠) 管理システム
  - データ設計が素朴すぎて、レストランの要望にそのまま応えるのが難しい
  - 「ウォークインでの集客が弱い、遅い時間だけネット集客の在庫を多くしたい」など

- 営業による運用でのカバーが常態化し、一休社内に謎知識が蓄積される
  - 「タイムセール用席」「30分ごとにずらした席を作ってプランを紐づけて〜・・・」
  - レストランは使い方が難しすぎて、自力で運用できない
  - 結果、一休側での代行運用が常態化し、社内オペレーションが高コスト化・・・

この問題を解消するには、在庫のデータ設計を根本からやり直す必要あり・・・

一休.com
こころに贅沢させよう。

在庫システムの刷新に乗り出す ••• だがしかし

- プロジェクト着手は2019年4月下旬
- 問題を見極めるため、店舗に足繁く通い、イシューを明らかに
- 開発工数を見積もり、役員会議にてレポート ••• **なんと、一蹴される (笑)**

社長「naoya さんの見積もったスケジュールは、要するに、エンジニアが必要な作業を積み上げたらこれぐらいかかります、ってことだよね」

naoya 「・・・ そうですね (とはいえ結構ストレッチしているん思うんだけど)」

社長 「顧客が新しいシステムをいつ欲しいのかから考えようよ。」

naoya 「あっ・・・」

※決して無茶なスケジュールでやれと言われたわけではなく、スケジュール算定の思考プロセスがエンジニアリング中心に偏っていたことに対する指摘です

顧客がいつ欲しいか

- 一休レストランの最大の繁忙期はクリスマス
  - お店の運用の複雑性、トラフィック量、取扱高すべてが MAX になる時期
- お店 「クリスマスまでに欲しい。さすがに今年は無理でも、来年のクリスマスには･･･」
- 今年のクリスマスでの運用に耐えうるシステムとしてターゲットしなければ

さあ、どうする

- クリスマスにある程度のお店が新在庫システムを使えるようにするには、10末にはローンチする必要がある
- 5月時点での感触では半年でリリースできる見込みは全くない・・・
  - もともと2020年2月といってたものを10月・・・

とりあえず「やります」とコミットしてみる

- やれるかどうか分からないが「やります」とコミットしてみることにした
  - 冷静に考えると、たとえ遅れたたとしても、かすり傷 (自社サービスですし)
  - 自らプロジェクト責任者として開発をリード
  - ディレクター含めて 7名で、開発スタート

- イシューは「10月までに全てを完成させる」ではないと捉え直す
  - 「顧客がいつ欲しいか」ではなく「どう作り直すか」をフォーカスにしていたので、全部作りきるつもりでいた
  - 先行導入店舗にターゲットを絞ったときに優先的に開発すべき要件に絞ることができるはず

しかし、やはり難しいは難しい

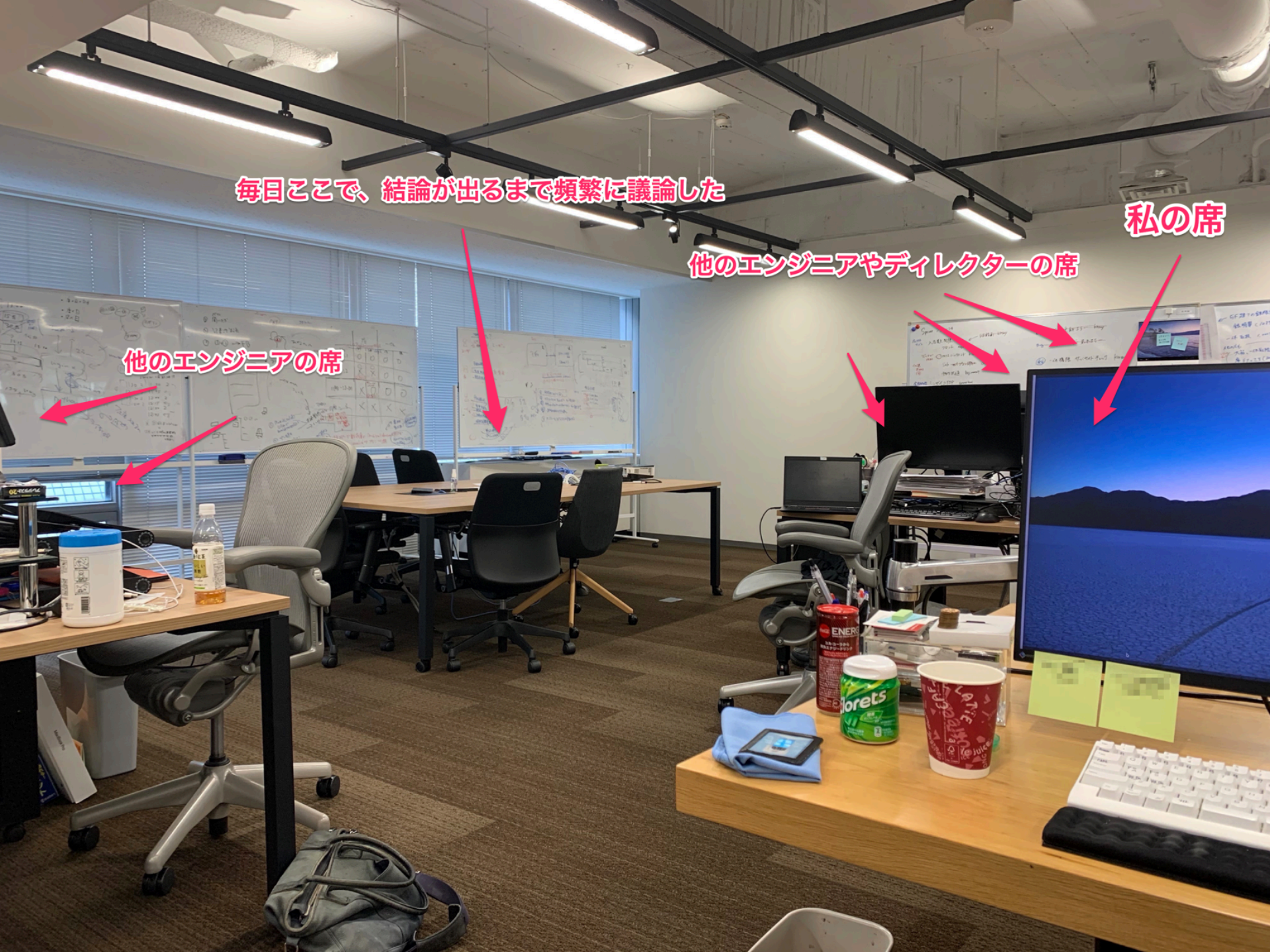
- レストランのオペレーションは小さな店舗から大きな店舗まで様々
- 日によって席のレイアウトを変えたり、ネット集客できる時間を限定したり、拡げたりほんとうにお店によって運用形態がいろいろある
- ひとりで考えていても、全く回答に辿り着ける気がしない･･･

結果的に、チームビルディングされた

- プロジェクト開始直後は、とにかく議論をして何を作るかを明らかにした
  - いろいろなお店にヒアリングにもいった
  - とにかく議論の毎日

- 自分たちが何を作っているのか見えるよう、一番最初に UI を作った
  - 動くもので確認。常に結合できるように
  - 間違ったものを作っていたらすぐに軌道修正できるよう、できたらすぐに動きを確認

- タスク管理はホワイトボードでざっくり (カンバンですらない)
  - 外せない大事なコミットメントにフォーカス。細かいことは各自に任せる
  - 細かく管理しないかわりに、1 〜 2週毎に半日ミーティングをして、プロダクトのチェック、要件の棚卸し、課題管理、どう間に合わせるかを話し合う

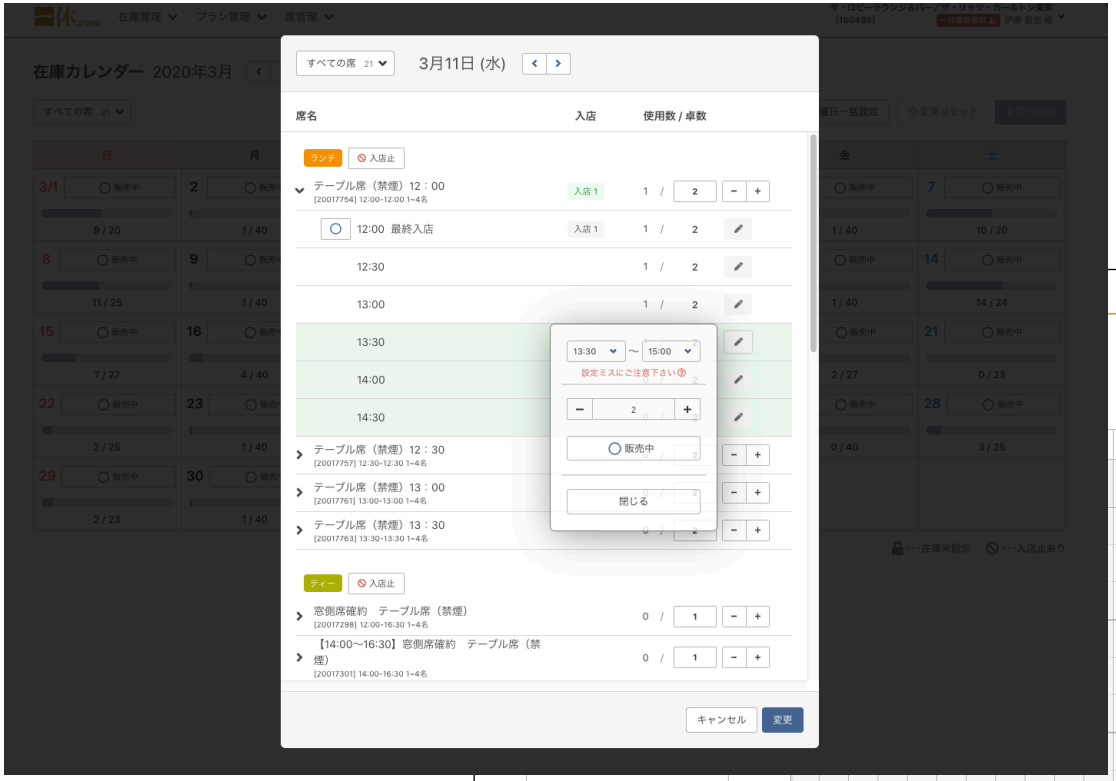

毎日ここで、結論が出るまで頻繁に議論した
私の席
他のエンジニアやディレクターの席
他のエンジニアの席

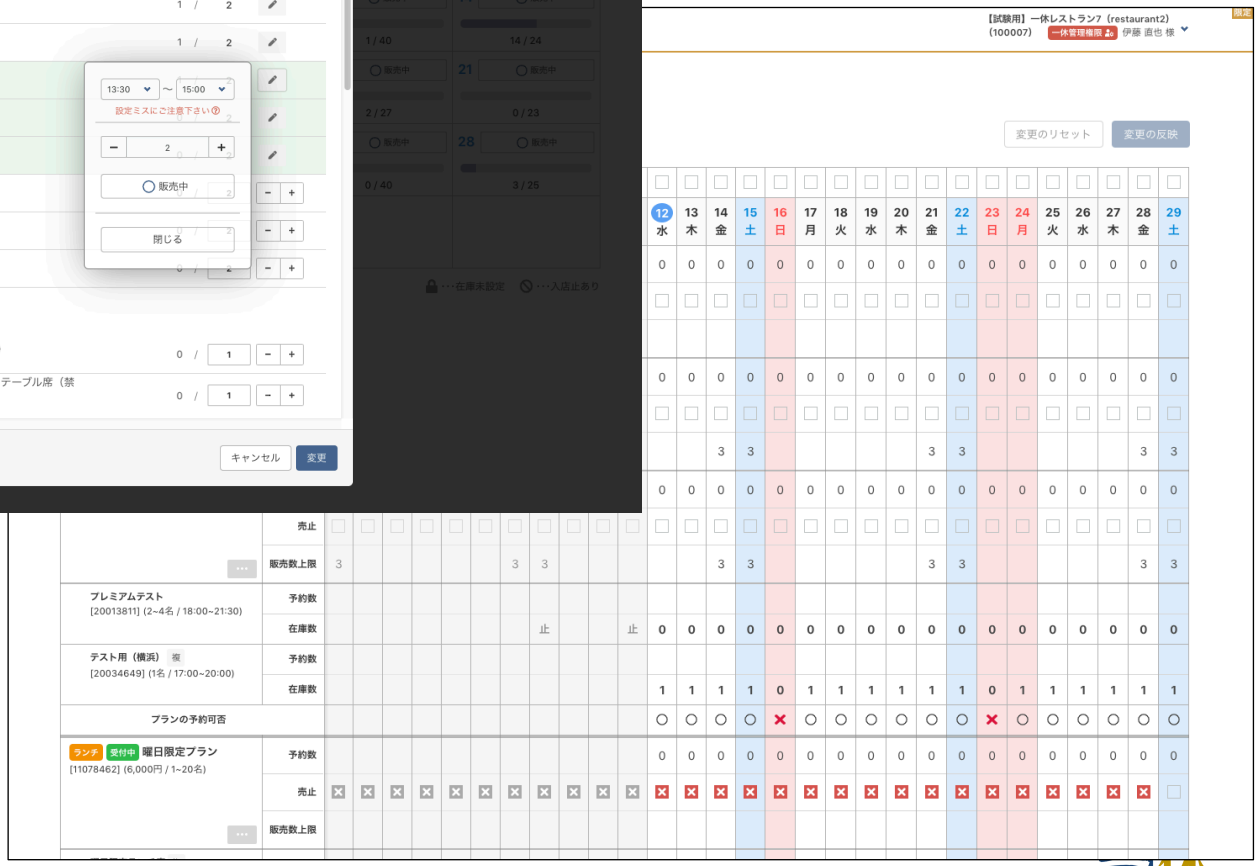
CTO として

- 自ら要件定義から実装まで一気通関でリードすることで、チームの開発効率を引き上げることを意識
  - 意志決定スピード、各自のストレッチ具合を引きあげるにはそれが一番効率的だと考えた
  - 組織マネジメントに割く時間を最小化し、このプロジェクトに一番時間を使える状況を作った

- 同様に困難なプロジェクトを達成した成功体験があり、その肌感をチームに伝えられればと思った
  - **世間で語られる方法論に惑わされず、目標達成のために自分たちが良いと確信できることの実践を細かく積み重ねること**

結果 ••• 10末に無事ローンチできた。ビジネス課題も解消



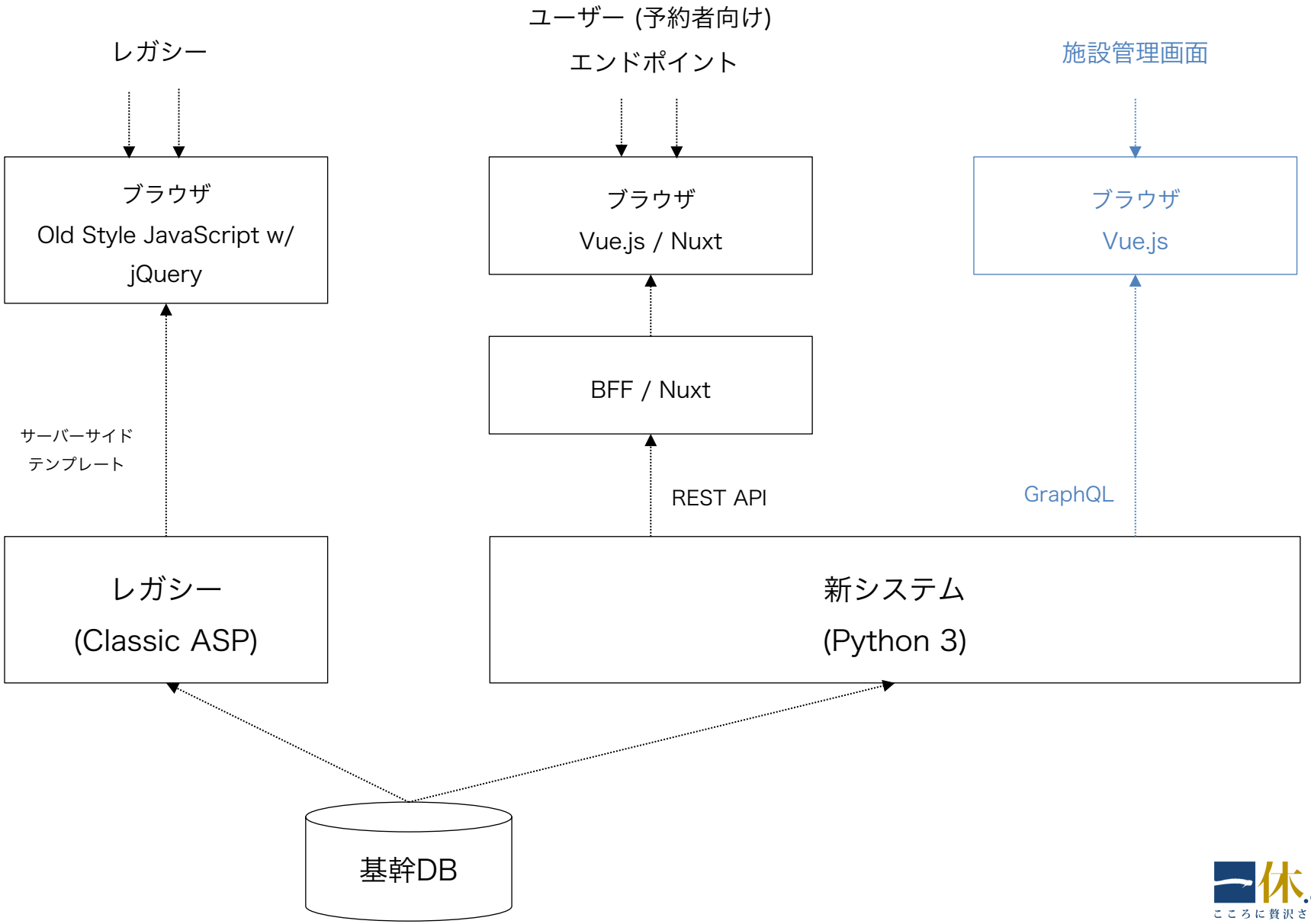


プロダクト品質、技術面での妥協はあったか?

- なし ・・・ 技術面、品質面では、むしろ進捗した
  - 新在庫システムは2年間、レガシー脱却のため作ってきた「restaurant2」で開発
  - GraphQL、EKS その他の採用

- UI やデザインも、理にかなったものになるよう妥協なく開発
  - 運用で毎日のように使う画面は SPA に
  - それ以外は今後もラピッドに開発できるよう、非 SPA に

【再掲】ユーザー向け、施設管理でフォーカスが異なる。2つの方式を確立



なぜ、そんなに速く開発できたのか

- **スケジュールと技術品質をトレードオフとは考えず**、レガシー撤廃を同時に実施したこと
  - プレゼンテーション・ドメイン分離、GraphQL の採用、ドメインモデル中心の設計が大きく開発速度に寄与した
  - 一方で「バランスの良い技術選択」は変わらず志向し続けた

- **難易度の高い課題だったからこそ、チームがチームとして機能した**
  - いつも通りやっていたら間に合わない中、妥協せずにどう完成させるかを日々ディスカッションし続けた
  - タイトなスケジュールの中、品質に拘りながらみんなよく頑張ったと思う。誰か一人欠けても達成できなかった

アジャイル? スクラム?

- 自分たちはそういうプロセスを実践しているという意識は全くなかった
- 限られた (困難な) スケジュールを前に、暗中模索の中、プラクティカルに試行錯誤を続けたら、それっぽいやり方になっていた
  - よく言われるように、よくあること。

考察 ・・・ 私たちが6年間で学んだこと

レガシーや組織にまつわる複雑な問題解決の方法に一般解はなさそう

- 大切なのは解決策ではなく、問題を理解すること
  - 問題が高度になればなるほど、<一般解> がそのまま当てはまることはまずない
  - ほとんどの問題は、その組織、システムに固有の問題。一般論を持ってきて当てはめるだけで解決することは難しい
    - わかっていても、我々は解決策 (正解) を求めてプロセスに飛びつきがちな生き物

チームビルディングやレガシー改善に関する、私たちのスタンス

- チームビルディングは、チームビルディングによってなし得るのではなく、物事を解決することを目的に、その過程で達成するのが良い
  - ･･･と、一休では考えている。(いろんな考え方はあると思います)
  - 課題がなければ、チームはチームとして機能する理由を見失う
  - 数ある目的達成手段のひとつでしかないと割り切る。より大事な問題にフォーカスできているか、つねに自己批判を続ける

- チームビルディングやレガシー改善そのもので生産性を上げようと考えるから、それで生産性が上がったかどうかを問われて苦しくなる
  - 問題を解決する過程で「そのほうがよかったからそうした。結果、問題は解決された」なら誰も文句は言わない

「技術やチームビルディングはただの手段、大事ではない」とは思わない

- 技術、チームビルディングはもちろん、大切
- 大切がゆえに人はそれに情熱を持つ。それ故にバイアスに囚われ、盲目的になってしまう
  - これに気をつけたい
  - 一休はソフトウェアの会社ではなく「宿泊・レストラン予約ビジネス」の会社。それを忘れない

一休のリーダーの志: リーダーが、リーダーでなくなる言い訳に打ち克つ

- 「自分より優秀なエンジニアが･･･」とか「自分がいらなくなるのが･･･」とか「メンバーのパフォーマンスを上げるのが･･･」とかマネージャーは綺麗事をいろいろ言いたがる
- 自分より現場のエンジニアの方が優秀なのはわかった。だからといって自分が開発をしない理由にはならない
- 自分自身が何かをやらない理由みたいなのを考え始めたら、それはただの言い訳である可能生を考える
  - 事業やプロダクトをリードするのに、本当にそれがベストなのか、ゼロベースで考えるべき
  - 組織を動かす、権限委譲するだけでは全ての問題は解決できなかった (･･･ 一休の場合は。)

まとめ

- 一休で6年かけて、レガシー解消や基幹システム刷新など、まずまずやってきました
  - ビジネスの会社でも、結構やれるということは証明できたかも

- 大事なこと、学んだこと
  - 問題の解決を急ぎすぎないこと。問題の理解につとめること
  - 一方でリスクを取って動くこと
  - 教条的になりすぎないこと。状況に適応した対応こそが好況を生むこと
  - 何がしたかったかを忘れないこと ･･･ **エンジニアにありがちなバイアスに打ち克つこと**