Auth0 in Microservices

~ リーガルテックの事績 ~

リーガルテック？

法律関連のレガシー・労働集約型の領域を

テクノロジーで代替していく

特に法務はレガシー

- 契約による法的拘束力は恐ろしい
- 契約書は十分なチェックが必要
    - 多大な労力と時間がかかる

- 権利義務関係の明確化
- リスクコントロール
  - 競業禁止規定
  - 任意解除規定
  - 賠償額と条件
  - etc

これら全てを
目視チェック

知識・スキルの属人化

法律は全ての人に等しく適用される

経験を積んだ人以外には法務業務は難しい

GVA TECH


gvatechHP

スタートアップ・フリーランスを中心に

サービス展開してきた

AI-CON	AI-CON登記
契約書チェックを AIで行う	法人の登記書類の作成を オンラインで行う

大企業の法務も同じ課題を抱えている

AI-CON Pro

目次

- マイクロサービス化の流れ
- Auth0の使いどころ

マイクロサービス化の流れ

契約書レビュー（ざっくり）

1. これから結ぶ契約書の草案を受け取る
2. ひな型契約書（自社にとっての理想形）と照らし合わせる
3. 草案修正
4. 修正版を契約書の締結相手に送る
5. 1~4を繰り返す

- 2👉Wordで行うことが殆ど
- 3👉条文単位で行う。草案とひな型の条文順が異なり、照らし合わせが大変

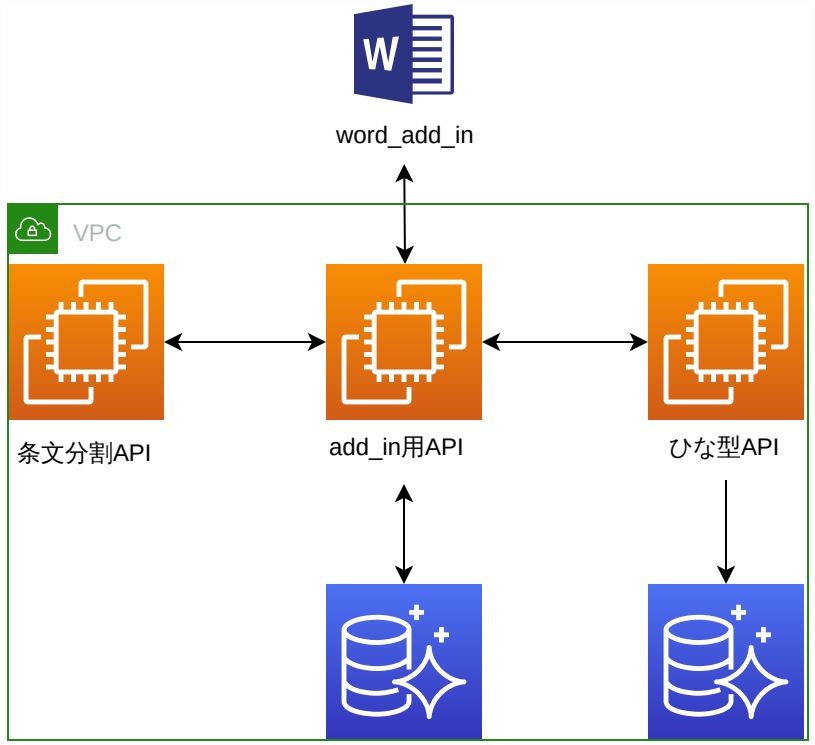
サービス概要

- 大企業法務向け
- Word add in
- セキュリティ最重視

使用技術

- client
  - 言語: TypeScript
  - FW: vue.js
- API
  - 言語: Go
  - FW: Buffalo
- DB
  - Aurora MySQL

最初期



仕様変更

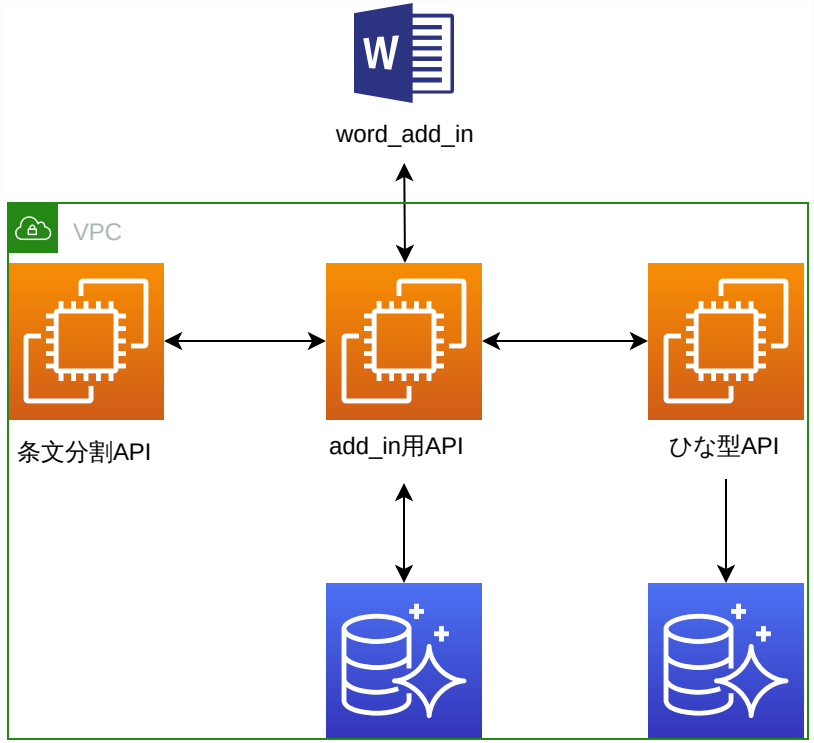
大企業	スタートアップ フリーランス
ひな型を持っていたり いなかったり	契約書のひな型を 持っていない

大企業	スタートアップ フリーランス
ひな型を持っていたり いなかったり	契約書のひな型を 持っていない
顧客企業作成・GVA TECH作成 両方のひな型でレビュー	GVA TECH作成の ひな型でレビュー

システム上の問題点

- ひな型契約書のセキュリティレベルが異なる
  - GVA TECH作成と顧客作成でひな型DBは分けるべき
- セキュリティレベルが異なる=認証のスコープが異なる
  - OAuth2フローに従うと、リソースサービスはAPIとDBの1対1対応が必要
  - つまり、 DBを分けたらリソースAPIも分ける

最初期



仕様増えた。。。

ここが潮目では。

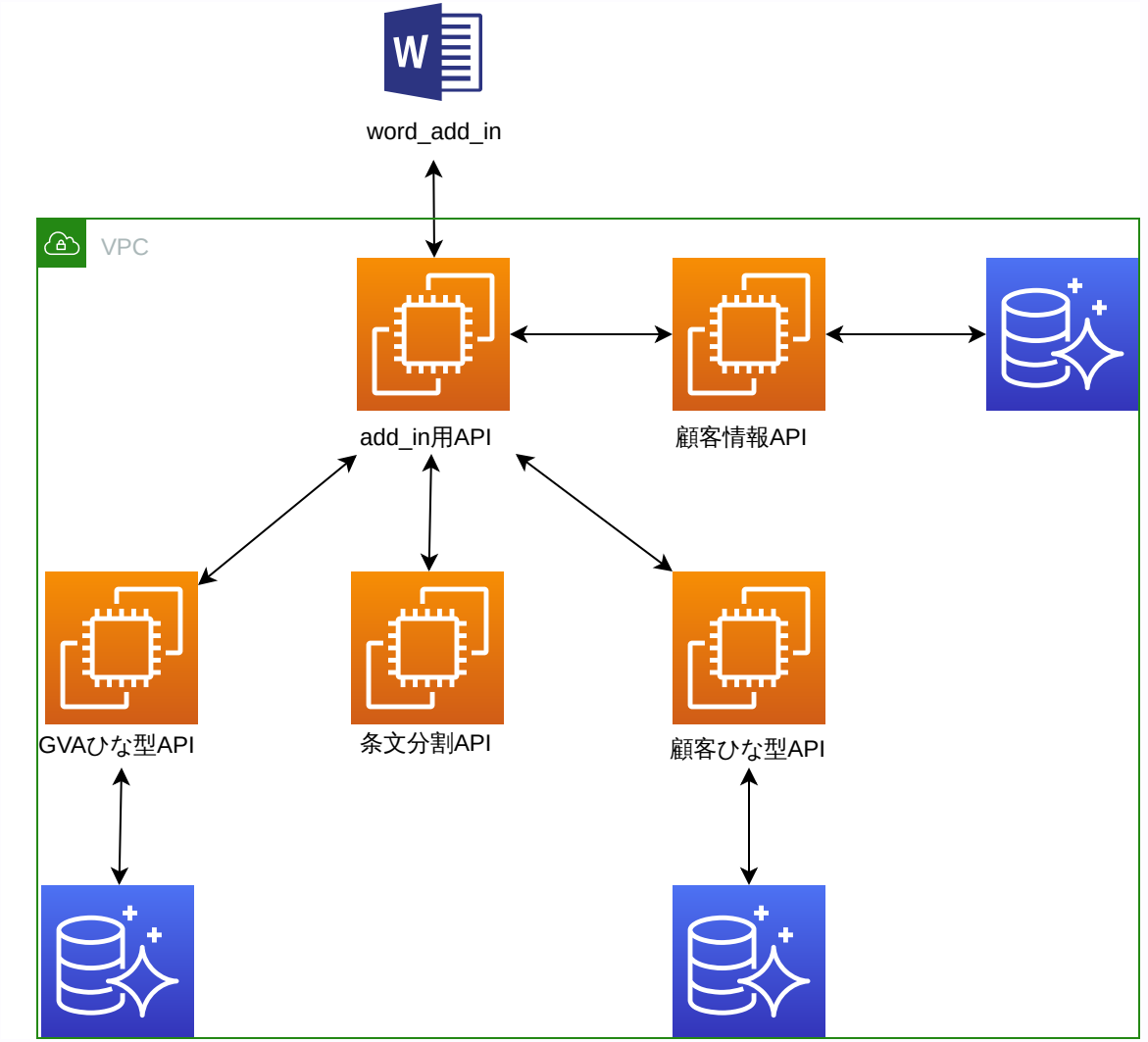
マイクロサービス化を徹底するぞ

柔軟＆堅牢な開発のため、徹底しよう

- スキーマ駆動開発
- テスト駆動開発
- アジャイル開発

どちらのひな型でも契約書レビュー

- GVA TECH作成
- 顧客作成

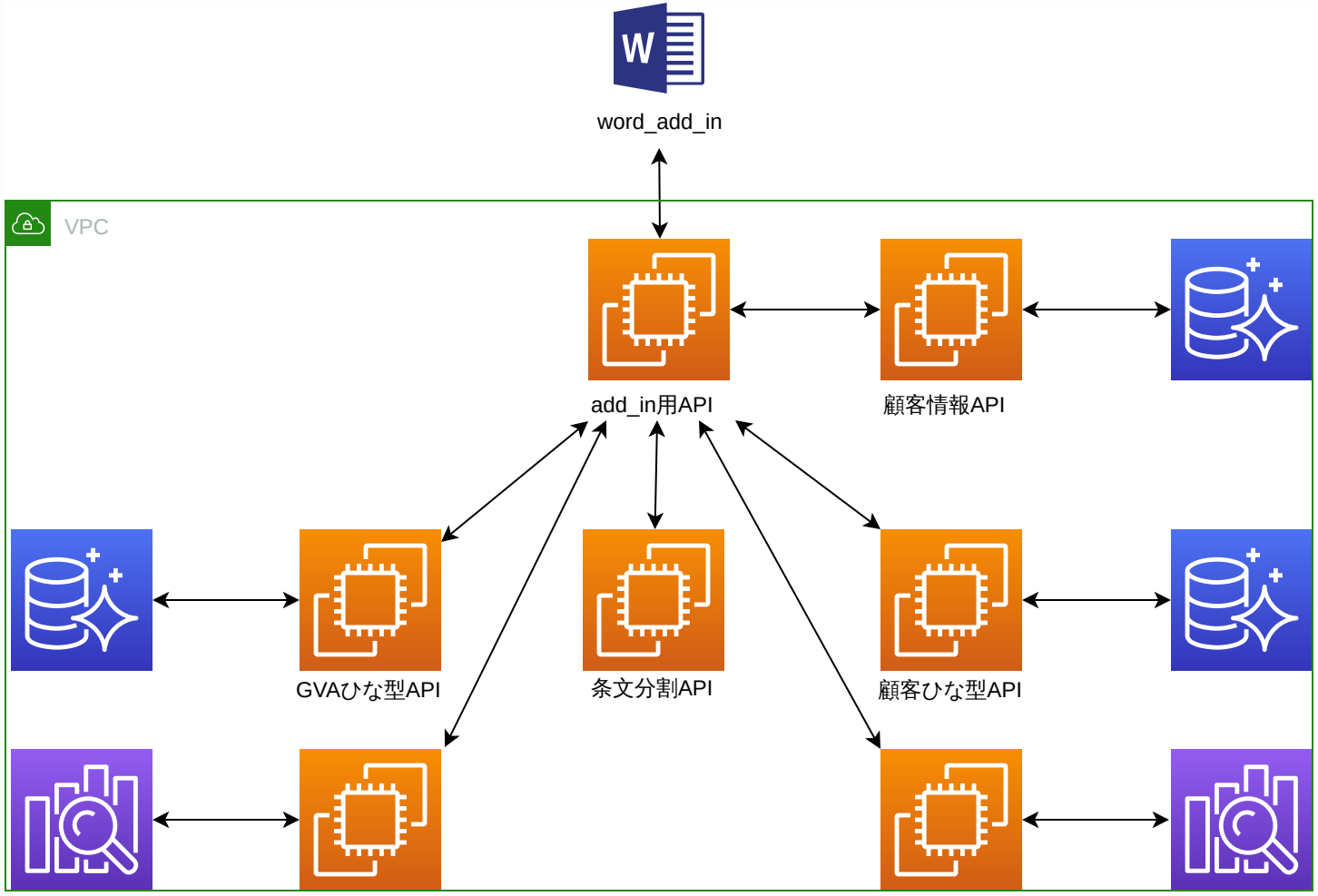


草案条文とひな型条文の
マッチ精度を上げたい

使用技術

- client
  - 言語: TypeScript
  - FW: vue.js
- API
  - 言語: Go
  - FW: Buffalo
- DB
  - Aurora MySQL
  - elasticsearch

elasticsearch投入



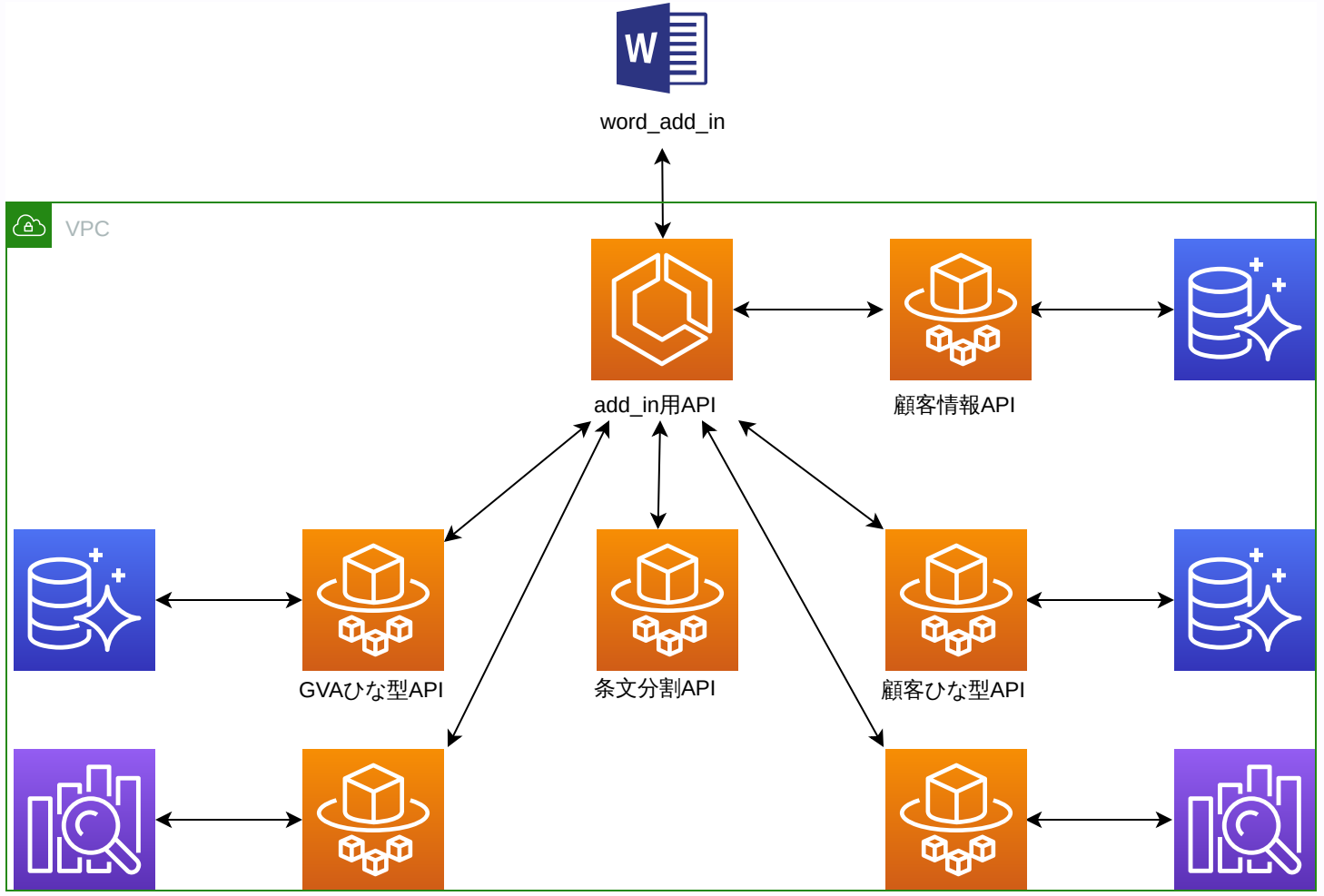
EC2が増えすぎている

コンテナ管理をしよう

追加使用技術

- IaC
  - terraform
- CI/CD
  - Fargate
  - ECS
  - CircleCI

コンテナ管理する

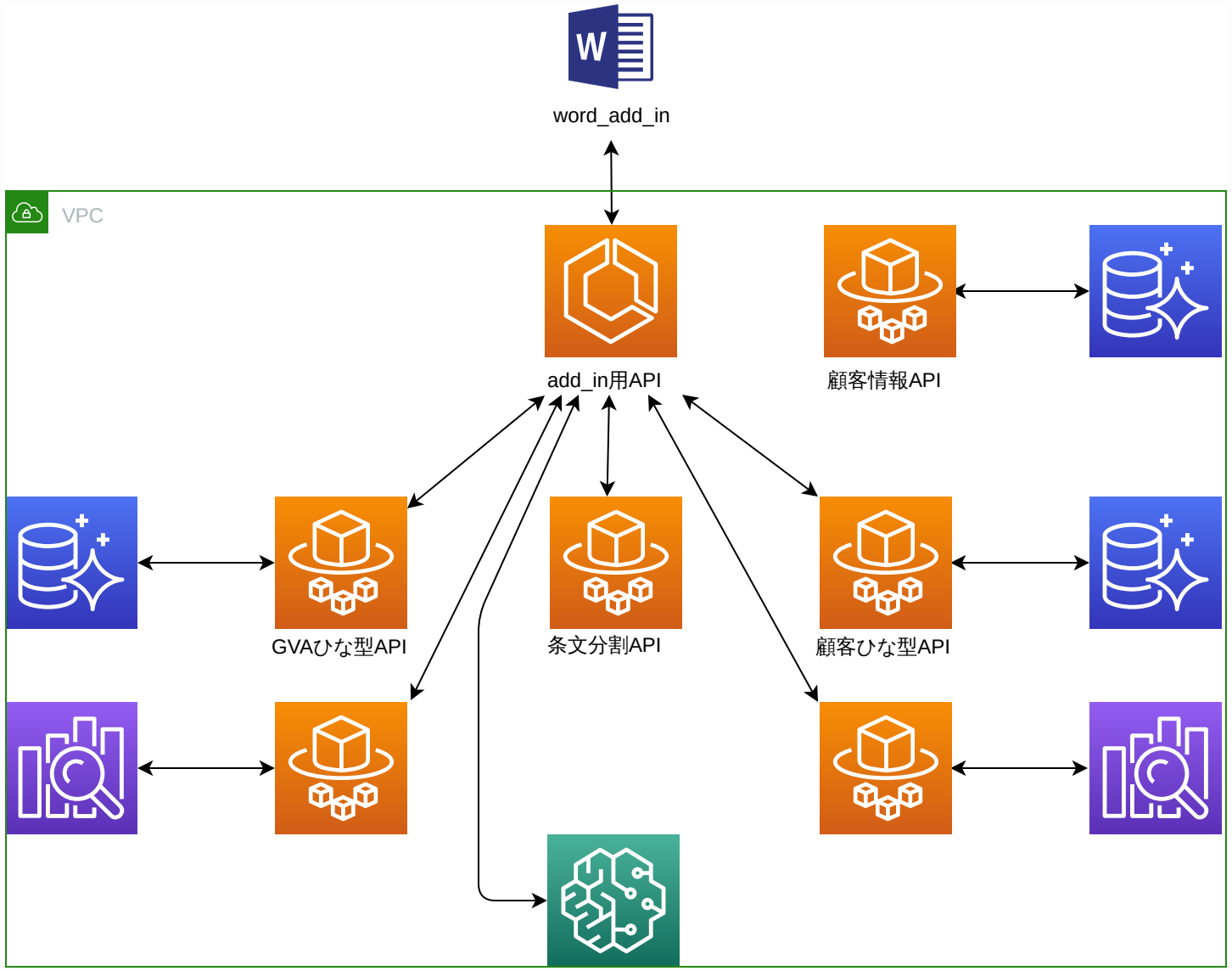

word_add_in
VPC
add_in用API
顧客情報API
GVAひな型API
条文分割API
顧客ひな型API

もっと精度を上げたい

追加使用技術

- ML
  - SageMaker

機械学習する


word_add_in
VPC
add_in用API
顧客情報API
GVAひな型API
条文分割API
顧客ひな型API

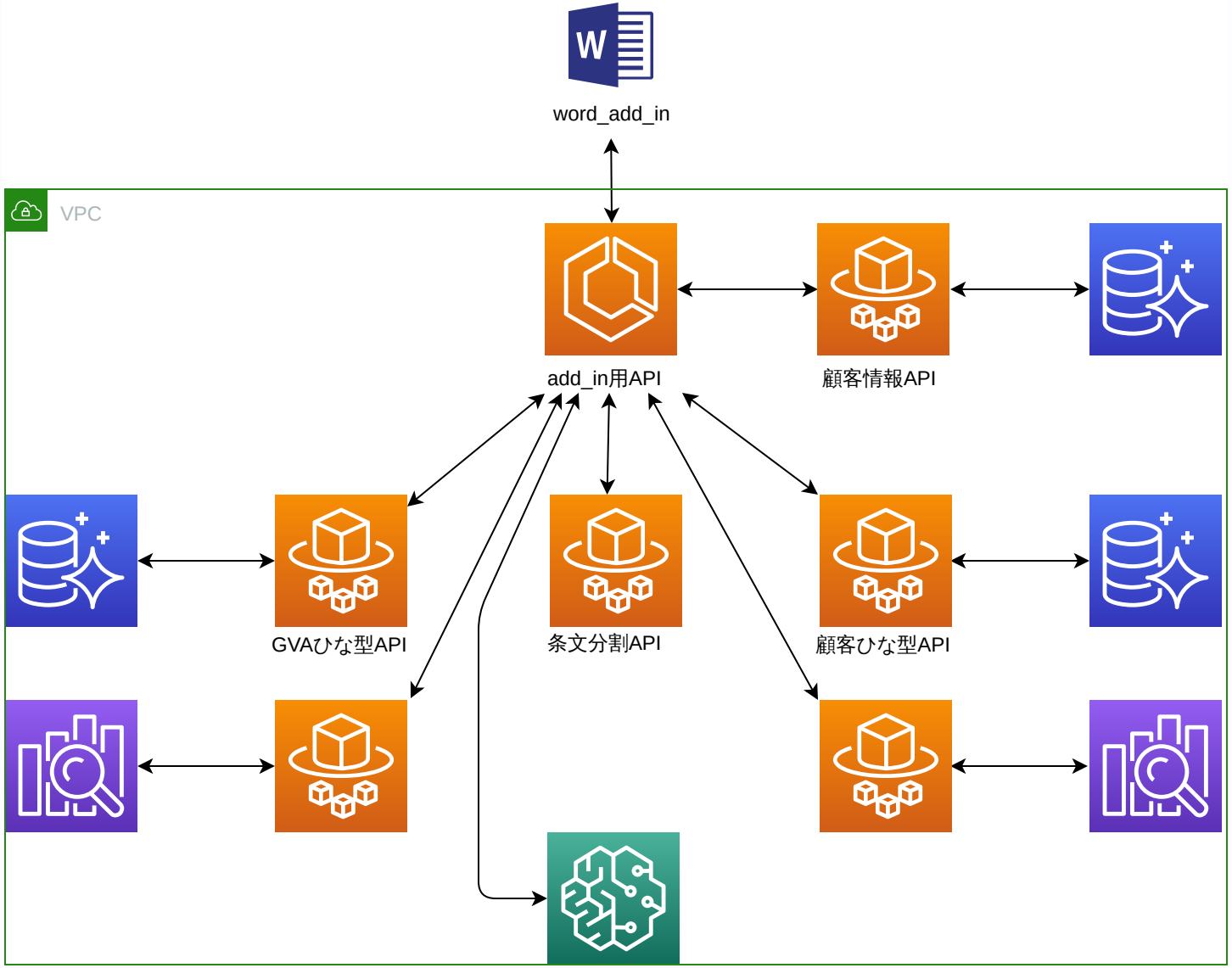
やりきった!!

やりきった…?

サービス概要

- 大企業法務向け
- Word add in
- セキュリティ最重視

さあ認証だ


word_add_in
VPC
add_in用API
顧客情報API
GVAひな型API
条文分割API
顧客ひな型API

😟どうすればいいの😟

Auth0の
使いどころ

- customDB
- トークン検証API
- 他

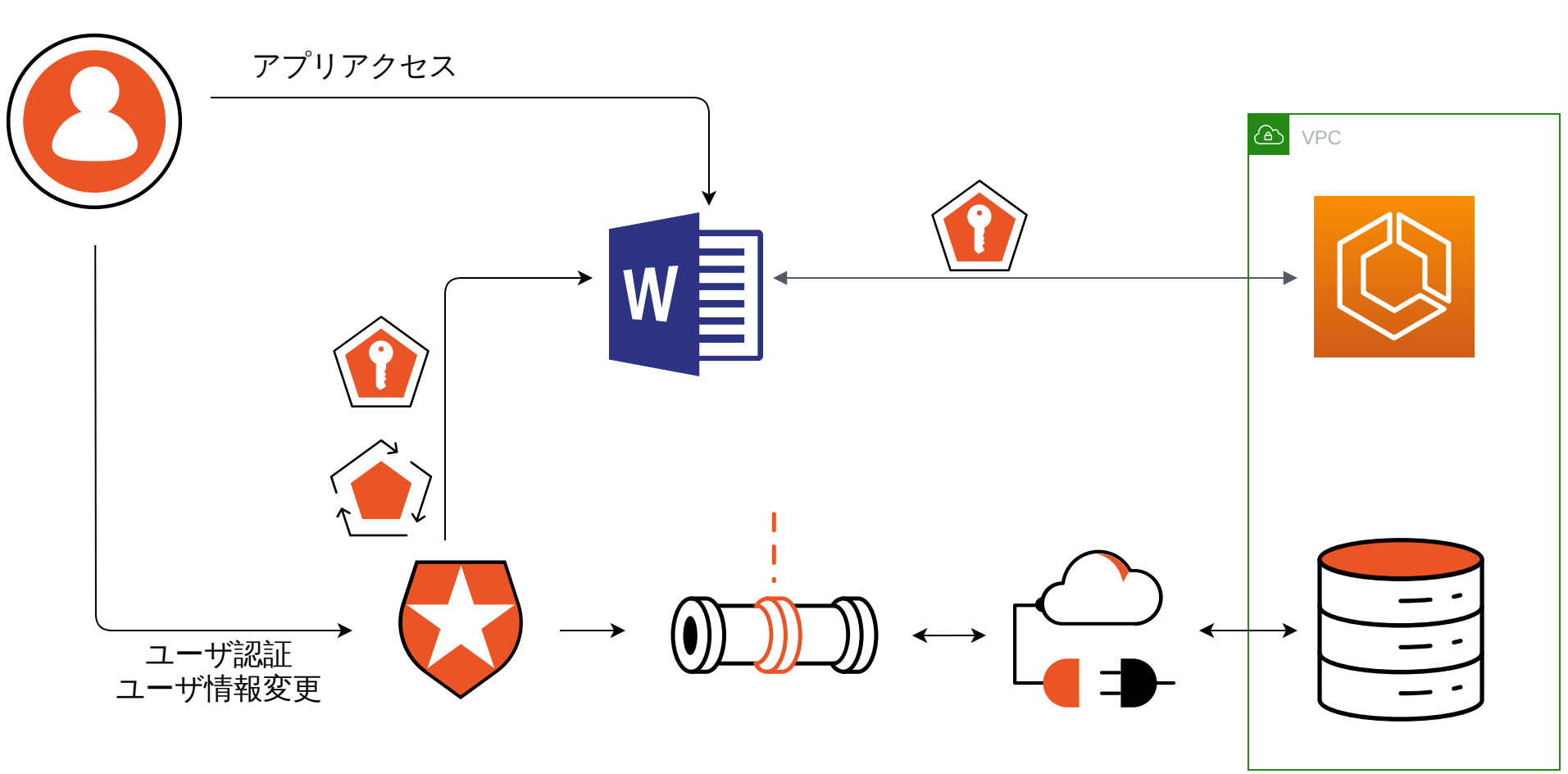
customDB

ユーザデータの格納場所

- user_metadata or app_metadata
  - profileの項目以外のデータを格納する場所
  - userに操作させたい情報はuser_metadataを使用
- 顧客データがuser_metadataに収まらないかも
  - user_metadataは合計16MBまで
  - userあたり10項目まで
  - custom DBを構築するのがよい

customDB データ連携

- customDBをpublicに晒さないため、APIを挟む
- Database Connectionsを設定し、ActionScriptsによりAPIへCRUDリクエスト



トークン検証API

構築経緯

- マイクロサービス化により、リソースAPIが複数
- リソースAPIの個数分のトークン検証処理コード
  - 処理統一の必要 ⇔ 処理がばらつく恐れ
  - DRY原則
- トークン検証処理をAPI化して、リクエストを送る

技術選定

- ~~API Gateway + Lambda~~
  - AWS SAMの導入
  - 多くのAPIからリクエストを受け続けるので、lambdaの恩恵があまり無い
- Buffallo API
  - 他のコンテナと同一基盤に載せた方が無難

躓きポイント

1️⃣Auth0 SDKがjwt-goのError情報を塗り替える

auth0/go-jwt-middlewareの抜粋
dgrijalva/jwt-goに依存する（package名: jwt）

```
func (m *JWTMiddleware) CheckJWT (w http.ResponseWriter, r *http.Reques
中略
  //jwt-goの正常値とエラー値の返却
  parsedToken, err := jwt.Parse(token, m.Options.ValidationKeyGetter)

  //jwt-goのエラーハンドリング
  if err != nil {
    m.logf("Error parsing token: %v", err)
    // error型ではなく、err.Error()でstring化して引数に渡す
    m.Options.ErrorHandler(w, r, err.Error())
    return fmt.Errorf("Error parsing token: %v", err)
  }
中略
  //jwt-goのエラー情報が失われた状態でフォーマットし、返却
  if !parsedToken.Valid {
    m.logf("Token is invalid")
    m.Options.ErrorHandler(w, r, "The token isn't valid")
    return errors.New("Token is invalid")
  }
中略
}
```

jwt-goで提供されているerrorの種類

```
// The errors that might occur when parsing and validating a token
const (
  ValidationErrorMalformed        uint32 = 1 << iota // Token is malfor
  ValidationErrorUnverifiable                        // Token could not
  ValidationErrorSignatureInvalid                    // Signature valid

  // Standard Claim validation errors
  ValidationErrorAudience      // AUD validation failed
  ValidationErrorExpired       // EXP validation failed
  ValidationErrorIssuedAt      // IAT validation failed
  ValidationErrorIssuer        // ISS validation failed
  ValidationErrorNotValidYet   // NBF validation failed
  ValidationErrorId            // JTI validation failed
  ValidationErrorClaimsInvalid // Generic claims validation error
)
```

err.Error()の実装

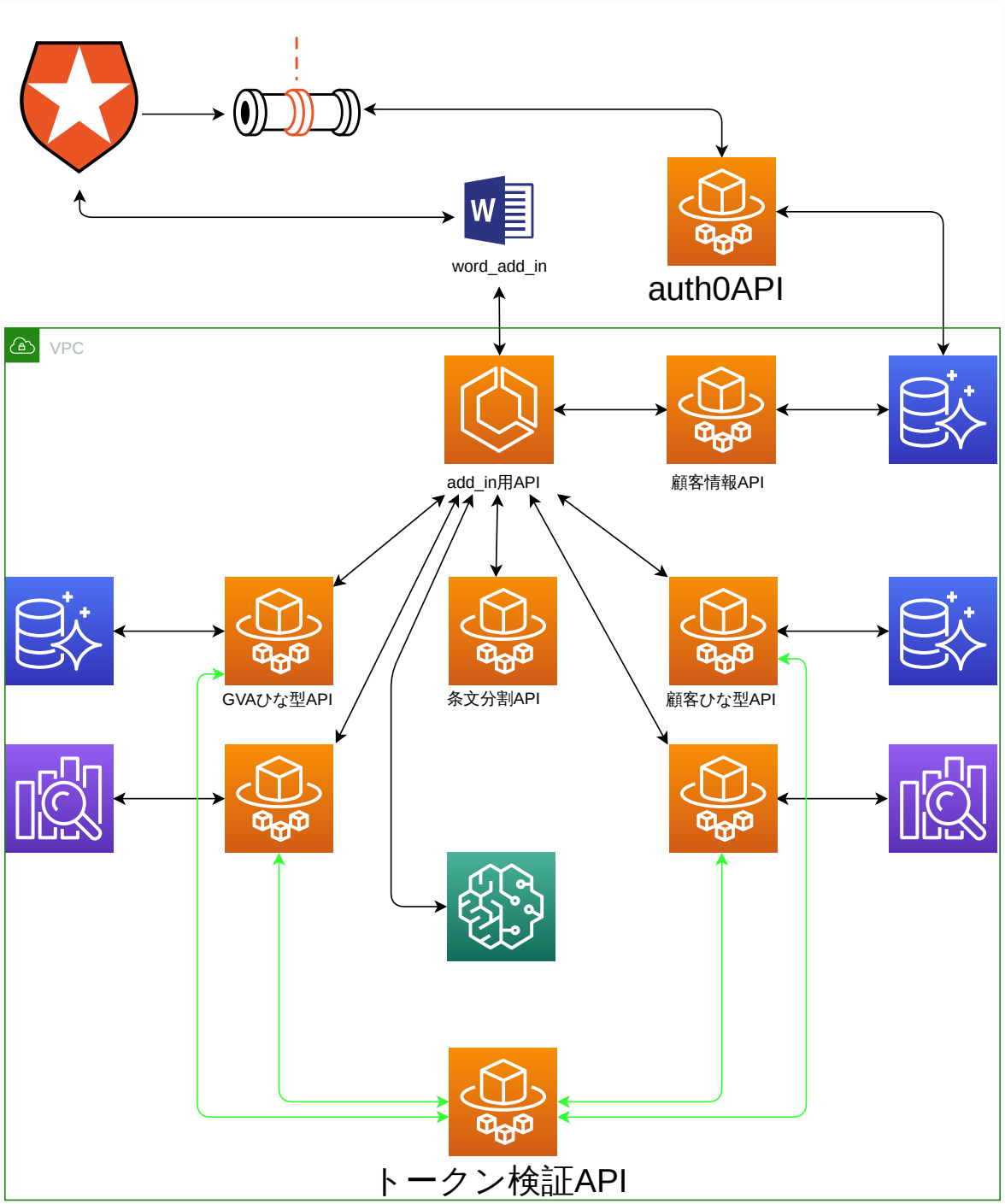
```go
// 前ページのエラーの種類を受けるメンバー
// The error from Parse if token is not valid
type ValidationError struct {
  Inner  error  // stores the error returned by external dependencies,
  Errors uint32 // bitfield.  see ValidationError... constants
  text   string // errors that do not have a valid error just have text
}

// 返り値はstring
// メンバーErrorsは返却されない
// Validation error is an error type
func (e ValidationError) Error() string {
  if e.Inner != nil {
    return e.Inner.Error()
  } else if e.text != "" {
    return e.text
  } else {
    return "token is invalid"
  }
}
```

- ErrorHandlerを独自実装可能だが、
  error型ではなくstring型が引数に指定されている
  - 値がstring化されると、
    エラーメッセージで判別することになり、脆弱
- APIでのエラーハンドリングが困難になるため、
  auth0/go-jwt-middlewareは使用しない
  - dgrijalva/jwt-goを用いて直接実装することとした

jwtの要素の型がぶれる

- Auth0から返却されるjwtの要素のうち、
  型が固定でないものがある
  - audienceの型が
    値が単一の場合string、複数の場合string配列
  - goでは、型が不定な返却値は扱いづらい
- jwt-go側のIssueにも挙げられている
  - PRも既に出ている
  - このPRのmergeが間に合わなかったため、
    該当箇所を独自実装することとした

最終的に


word_add_in
auth0API
VPC
add_in用API
顧客情報API
GVAひな型API
条文分割API
顧客ひな型API
トークン検証API

その他やっていること

- Connections
  - social login
- Rules
  - srcIP制限
  - MFA

今後やっていくこと

- トークン検証API廃棄
  - 処理のプライベートパッケージ化
- 各テナント間でCI/CDを回す
- ActionScriptのTypeScript化

AI-CON Pro

β版リリース

問い合わせ受付中