なぜAPIアーキテクチャが重要なのか

APIアーキテクチャ設計概説

山野 裕司
<Yamano_Yuji@ogis-ri.co.jp>

株式会社オージス総研
クラウドインテグレーションサービス部

本日の内容

- イントロダクション
- システムアーキテクチャ
- APIサービスのアーキテクチャ
- RESTful API (インターフェースの設計)
- Web APIの認証、認可
- 互換性とバージョニング

イントロダクション

APIとは

アプリケーションプログラミングインタフェース（API、英: Application Programming Interface）とは、ソフトウェアコンポーネントが互いにやりとりするのに使用するインタフェースの仕様である。

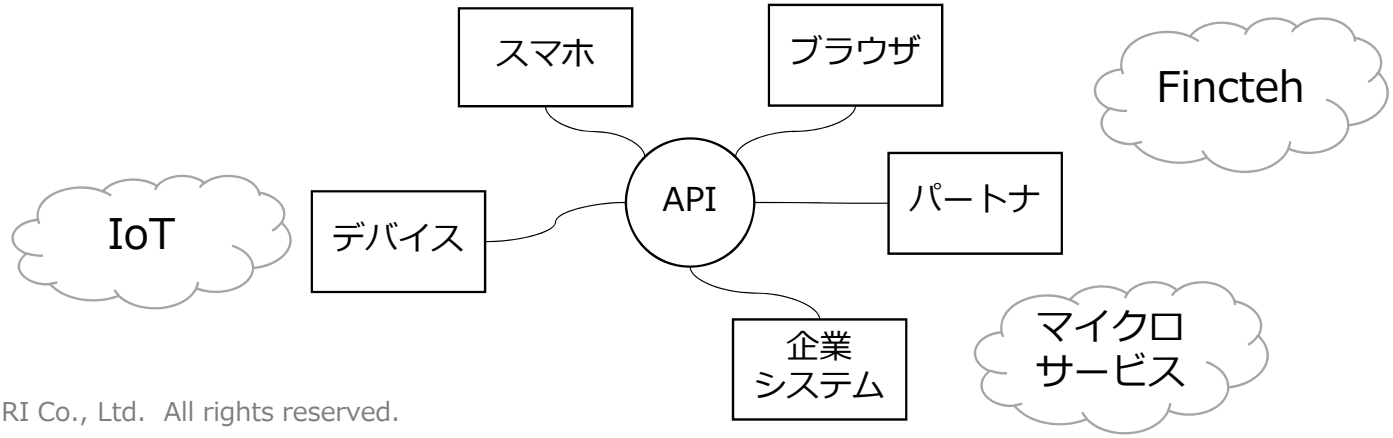
APIを使うことでコンピュータソフトウェアが他のソフトウェアと広義の意味で通信しあうことができる。

https://ja.wikipedia.org/wiki/アプリケーションプログラミングインタフェース

- 実際には、もっとルーズに使われている
  - インターフェースの仕様そのもの
  - インターフェースの仕様を実装したもの

Web APIとは

- APIのコンセプトをWeb技術を使うシステムに適用したもの
  - REST
  - HTTP
  - JSON
- 複数のクライアントに対して、プログラム可能なインターフェースを提供することにより、簡単につなぐための技術



なぜ、APIアーキテクチャが重要なのか

- 製品やサービスの導入だけでは解決できない問題
  - 既存のアプリをAPI化したい
  - どうすれば使いやすいインターフェースを設計できるのか
  - どのような認証、認可方式を採用すべきなのか
  - APIのバージョニング管理はどのようにおこなうべきか
- 巨大化するシステム、複雑化する要求に場当たり的な対応では限界
  - 品質
  - セキュリティ
  - 可用性、拡張性、保守性
- トレードオフを考慮し、最適な構造や仕組み、製品の組み合わせを設計する必要がある

なぜ、「API」？

- Webアプリケーションに似ているが、異なる部分もある
- API固有の課題
  - 既存のシステム、データのAPI化
  - インターフェースの設計(REST)
  - 三者間の認証、認可
  - 管理下にない多種、多数のクライアント
- 企業内のWebアプリケーションとの違い
  - インターネットとの接続
  - 突発的に増えるトラフィック
  - 頻繁にリリースを繰り返し、進化し続けるシステム

Web APIアーキテクチャ設計ワークショップのご紹介

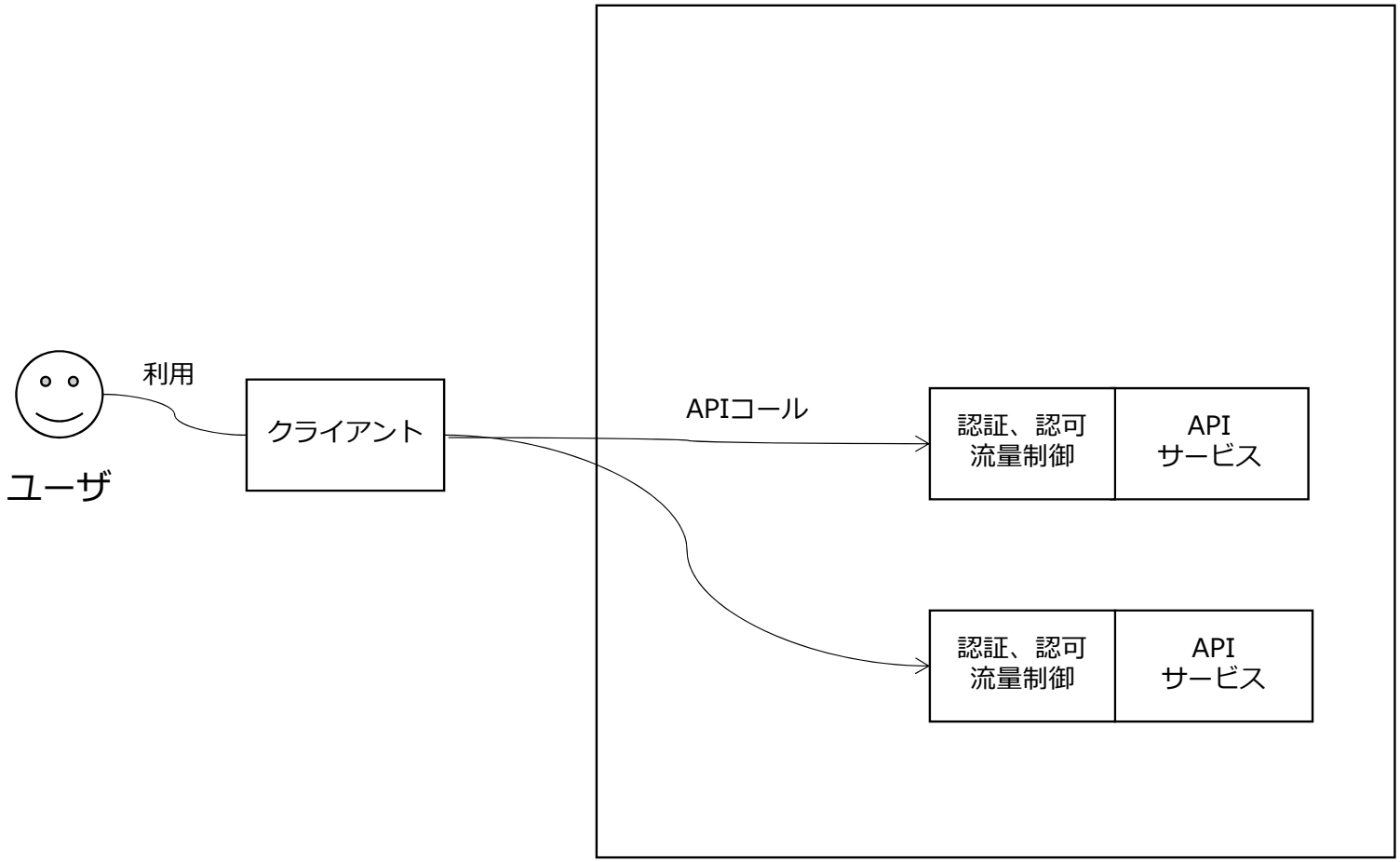
- 講義
  - システムアーキテクチャ
  - APIサービスのアーキテクチャ
  - RESTful API
  - Web APIの認証、認可
  - 互換性とバージョニング
- 演習
  - 要件の仮定
  - アーキテクチャの設計
  - 発表と議論
- フリーディスカッション

システムアーキテクチャ

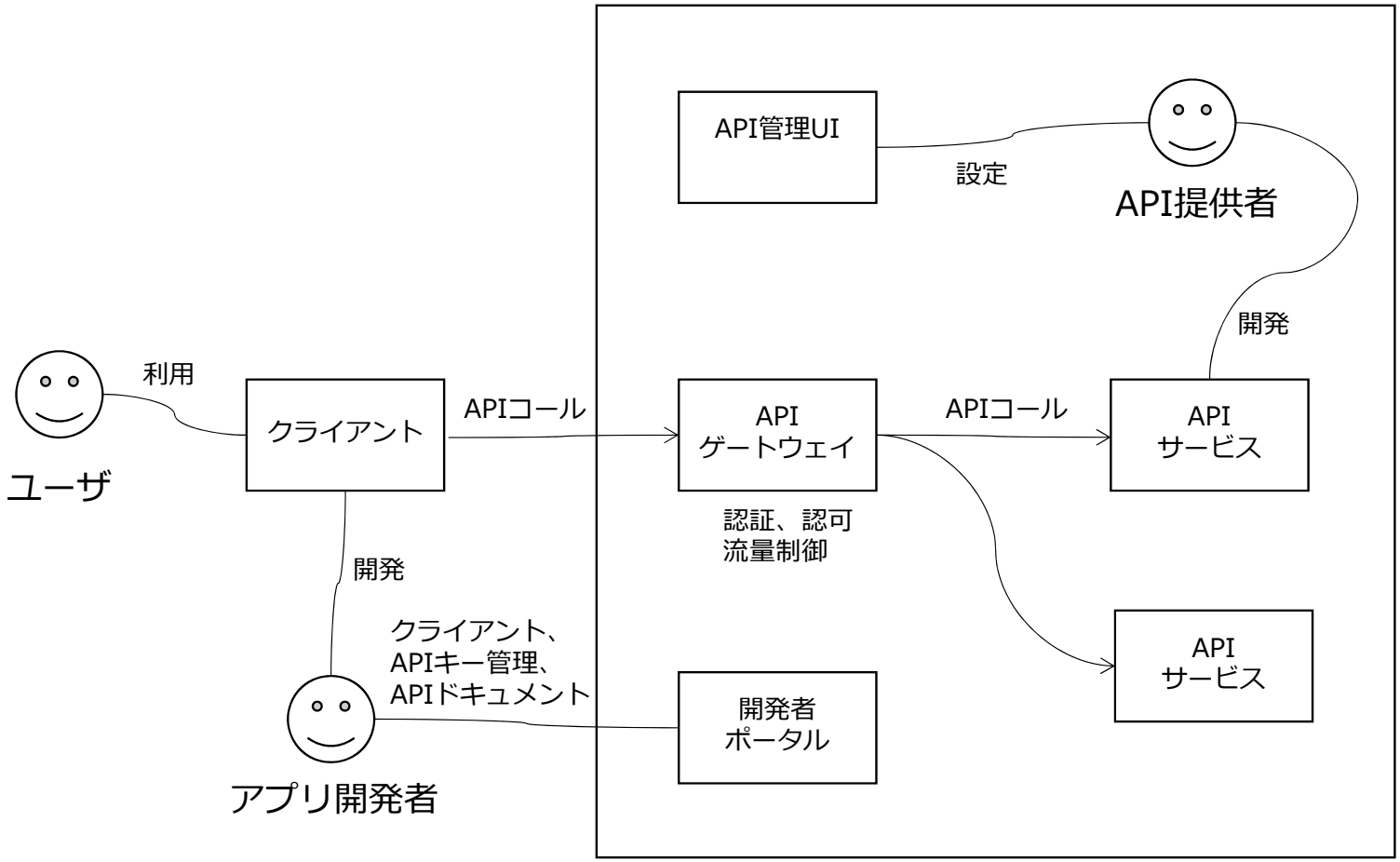
システムアーキテクチャ

- システム全体のアーキテクチャ
- APIゲートウェイが特徴的
  - 全てのAPIのエントリーポイントとなるAPIゲートウェイをもうける
  - 複数のAPIに共通する処理をおこなう
    - 認証、認可
    - 流量制御
    - バージョン管理
    - 課金や分析データの収集
    - キャッシング
    - オーケストレーション
    - フォーマットやプロトコルの変換

APIゲートウェイを使わない場合



APIゲートウェイを使う場合



システムアーキテクチャの設計ポイント

- APIゲートウェイを使うか
  - 流量制御やキャッシュが必要か
  - 複数のAPIサービスが存在するか
- 開発者ポータルを使うか
  - クライアントやAPIキー管理をAPI提供者がおこなえるか
  - APIドキュメントを提供する方法があるか
  - 多数の開発者に使ってもらいたいか
- APIゲートウェイでオーケストレーションや変換をすべきか
  - 一般的にはやるべきではない
    - APIサービスの変更がAPIゲートウェイに影響する
    - テストしにくい
  - リリース時期やコストを優先するのであれば、負債になる覚悟を持つ

APIサービスの
アーキテクチャ

APIサービスのアーキテクチャ

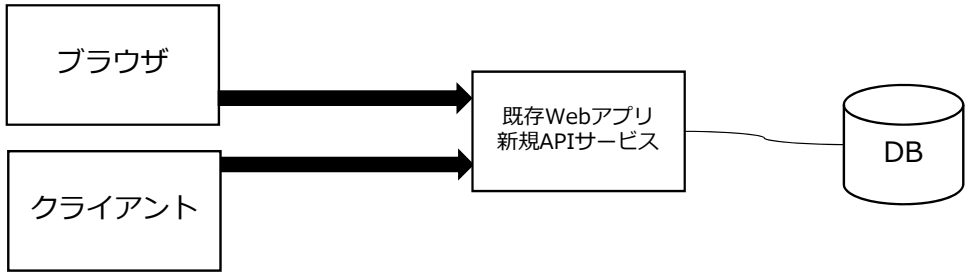
- APIの実装(APIアプリケーション)のアーキテクチャ
- ありがちな要件
  - 既存のシステム、データをAPI化したい
    - データベースを共有する/レプリケーションする/共有しない
    - 既存のアプリにAPIを追加/APIアプリを新規開発
    - 既存のAPIの前に新規のAPIを追加
  - スモールスタートしたい
    - 最初にどこまで実装するか、どこまで拡張性を考慮するか
    - いずれにせよどこかでコストを払う必要がある

APIサービスのアーキテクチャのベース

- Webアプリをベースに考える
  - ただし、Web APIはステートレスなので、セッション管理のようなサーバ側の状態管理のことは忘れる
- 必要に応じて検討
  - 負荷分散、冗長化
  - キャッシュ
  - 非同期処理
  - RDB以外のストレージ
  - イベント駆動、ノンブロッキングI/O アプリ基盤

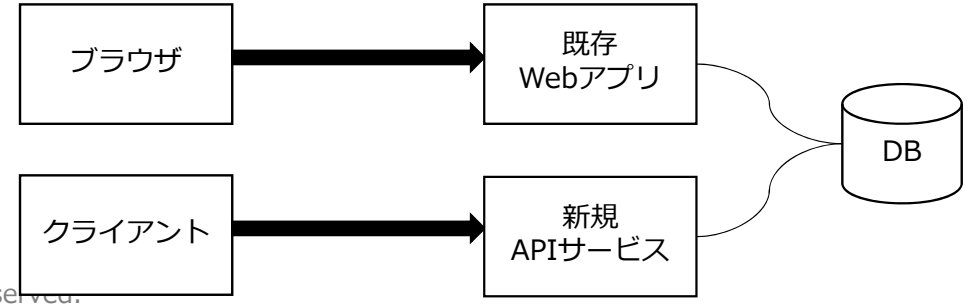
既存Webアプリを修正

- 既存のWebアプリと
  - 開発、管理組織が一緒
  - SLAが一緒
  - 変更サイクルが一緒
- 小規模のアプリで使われることが多い

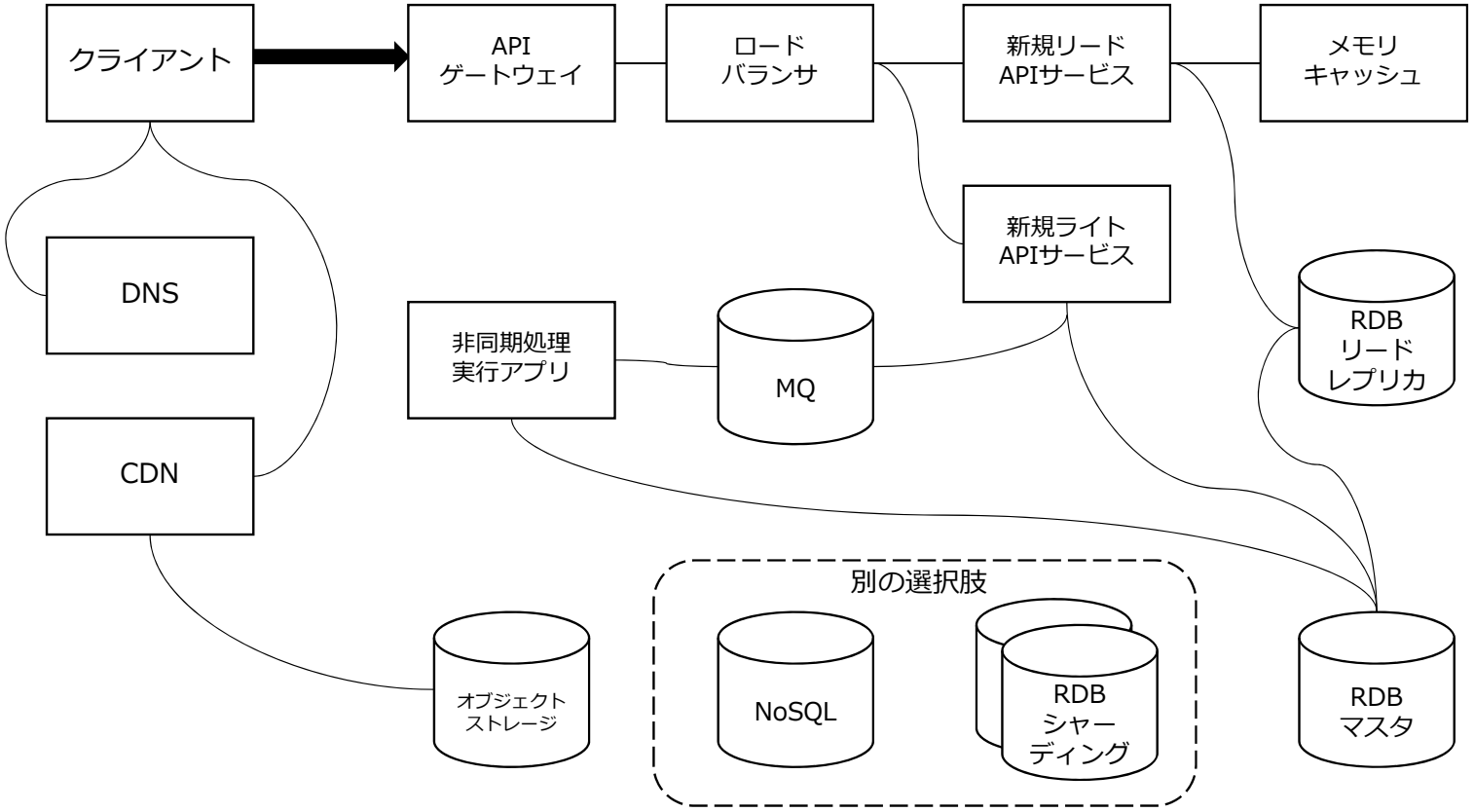


既存DBを利用し、新規にAPIを開発

- 既存のWebアプリと
  - 開発組織が異なる
  - SLAが異なる
  - 変更サイクルが異なる
- 既存アプリへの影響を少なくしたい
- DBに関する制約を許容できる
  - 管理組織が同じ
  - テーブルスキーマの変更タイミングを調整できる
  - SLAが同じで、負荷も大きく変わらない



成長するアーキテクチャ



APIサービスのアーキテクチャの設計のポイント

- 既存Webアプリ、APIが存在する場合は、境界を意識する
  - 開発、管理組織
  - SLA
  - 変更サイクル
- 既存DBを共有する場合は、テーブルをリソースに素直にマッピングできない場合が多いので、解決方法を検討しておく
  - ビュー
  - アプリケーションレイヤで実装する
- 将来のために準備しすぎない

RESTful API
(インターフェースの設計)

RESTful APIとは

- REST = Representational state transfer
- Webアーキテクチャの基本的な考え方
  - 仕様や実装ではない
- REST を Web APIにも適用
  - 単純さ
  - ユーザビリティ
  - スケーラビリティ
  - ツールが充実
- 主な特徴
  - ステートレスなクライアント/サーバシステム
  - 全てのもの(リソース)に一意の識別子(URI)を割り当て
  - 統一されたインターフェースでリソースにアクセス

```
GET /users

{
  "users": [{"id": 123, "name": "John Doe"},
            {"id": 124, "name": "Jane Doe"}],
  "nextLink": "http://example.com/users?offset=100"
}
```

RESTful APIで考慮すべきこと

- 共通のクエリパラメータ
- HTTPステータスコード
- レスポンスボディ
- エラーレスポンス
- リソース設計
  - サブリソース
  - 非リソースの扱い
- 非同期処理
- API仕様の定義フォーマット
- Hypermedia API

RESTful APIの設計のポイント(1)

- 先行者の知恵を借りる
  - クラウドサービスのAPIドキュメント
  - ベスト・プラクティス、API設計ガイド
  - 書籍
  - ただし、前提条件や文脈が異なる、過去の常識が通用しない可能性には要注意
- フレームワークの提供する機能を使う
- REST純粋主義に陥らない

RESTful APIの設計のポイント(2)

- HTTPの仕様を有効に使う
- リソースモデリングをおこなう
  - 内部モデルをそのまま見せない
  - 一貫性のある名前、構造
  - 一般的なAPIのユースケースにとっての使いやすさ
- APIの共通仕様を決める
  - HTTPメソッド
  - リクエスト、レスポンスのフォーマット
  - HTTPヘッダ
  - ステータスコード
  - エラーレスポンス
  - クエリパラメータ
  - コレクションリソースを返す場合の上限

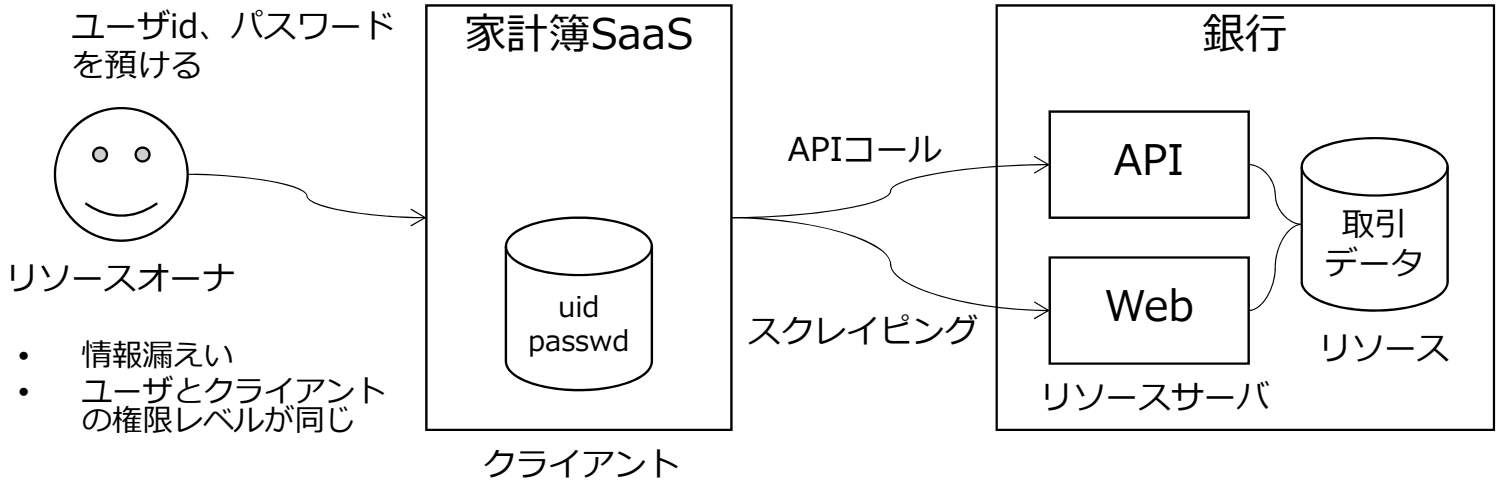
Web APIの認証、認可

Web APIの認証、認可

- 何を認証、認可するのか
  - クライアントアプリそのもの
  - クライアントアプリのユーザ
  - クライアントアプリ + ユーザ
- クライアントが秘密鍵を保持できるか
  - confidential: サーバ
  - public: モバイルアプリ、ブラウザのJSクライアント
- 選択肢
  - OAuth 2.0
  - ベーシック認証、APIキー
  - リクエストの電子署名
  - SSLのmutual authentication

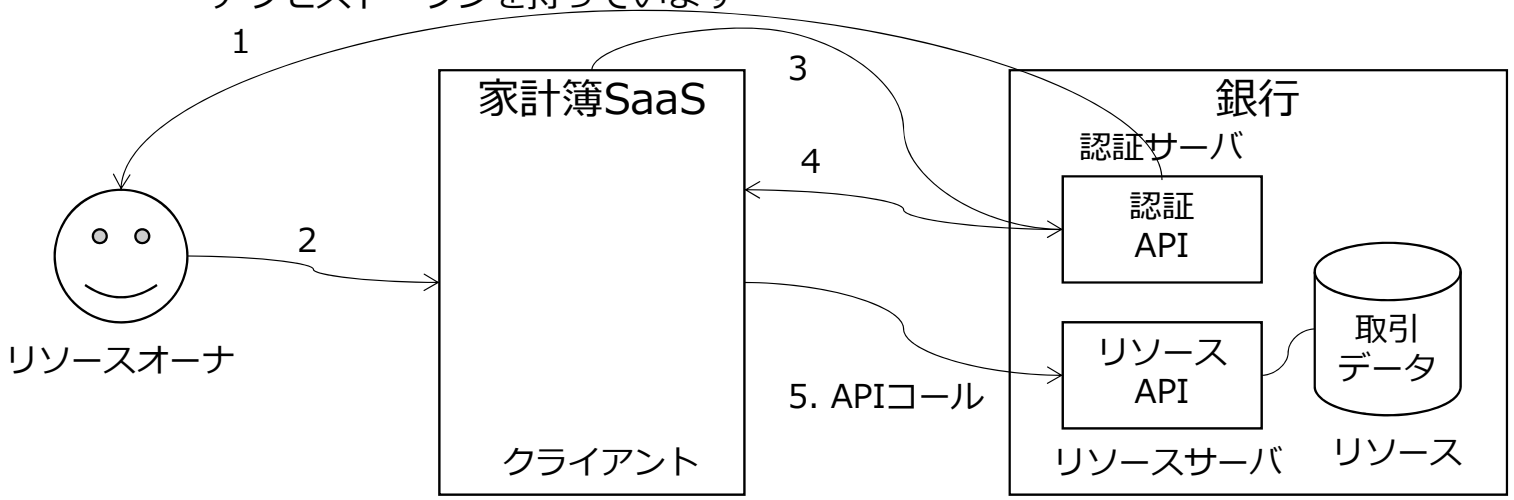
OAuth 2.0とは

- 認可のフレームワーク
- サードパーティのクライアントによる限定的なHTTPリソースへのアクセスを可能にする
- サードパーティクライアントがユーザのパスワードを保持する必要がない



OAuth2のAuthorization Code Grant

1. 家計簿SaaSがあなたのリソースにアクセスする許可をもとめています
2. いいですよ (auth code)
3. 許可をもらったことを証明できるものをください
   - 私は家計簿SaaSです (client_id、client_secret)
   - リソースオーナの許可をもらいました (auth code)
4. いいですよ (access token)
5. リソースをください
   - アクセストークンを持っています



認証、認可の設計のポイント

- 何を認証、認可するのか
- クライアントは秘密鍵を保持できるのか
- 自分で作らず、標準仕様や実装を使う
- OAuth 2.0
  - 安全性と利便性のバランス
    - アクセストークンの有効期間
    - リフレッシュトークン利用の有無
  - スコープ
  - トークンの失効

互換性とバージョニング

APIの互換性は重要

- APIはインターフェース
  - 提供者と利用者の契約
  - 互換性が重要
- API提供者はAPIクライアントを完全にコントロールできないことが多い
  - 古いクライアントが使われ続ける
  - 新しいバージョンの利用を強制できない場合もある

互換性とバージョニングの方針

- APIの仕様と実装のバージョンを区別する
- 仕様のバージョン
  - できる限り仕様の互換性を保つ
  - 互換性を保てない場合、バージョンをあげる
  - 古いバージョンも並行稼動する
  - クライアントは仕様のバージョンを意識して、APIを呼ぶ
    - パスの一部: /api/v1/users
    - クエリパラメータ: /api/users?version=1
    - ヘッダ: GData-Version: 1
- 実装のバージョン
  - それほど気にする必要がない

互換性とバージョニングの設計のポイント

- APIはインターフェースであることを意識する
- APIの仕様と実装のバージョンを区別する
- API仕様の互換性を保つ
- 互換性を維持する仕組みを作る
  - Consumer Driven Contract Testing
  - ベータ版を提供し、早い段階で発見する

まとめ

まとめ

- Web APIはつなぐための技術
- アーキテクチャは重要
  - 巨大化するシステム、複雑化する要求に場当たり的な対応では限界
  - トレードオフを考慮し、最適な構造や仕組み、製品の組み合わせを設計する必要がある
- APIアーキテクチャにおける重要な設計項目
  - システムアーキテクチャ
  - APIサービスのアーキテクチャ
  - RESTful API (インターフェースの設計)
  - Web APIの認証、認可
  - 互換性とバージョニング

お問い合わせ

株式会社オージス総研

営業本部 東日本営業部 営業第三チーム

【 TEL 】03-6712-1201

【 E-mail 】info@ogis-ri.co.jp

【 URL 】http://www.ogis-ri.co.jp