

REST WebAPI サービス設計

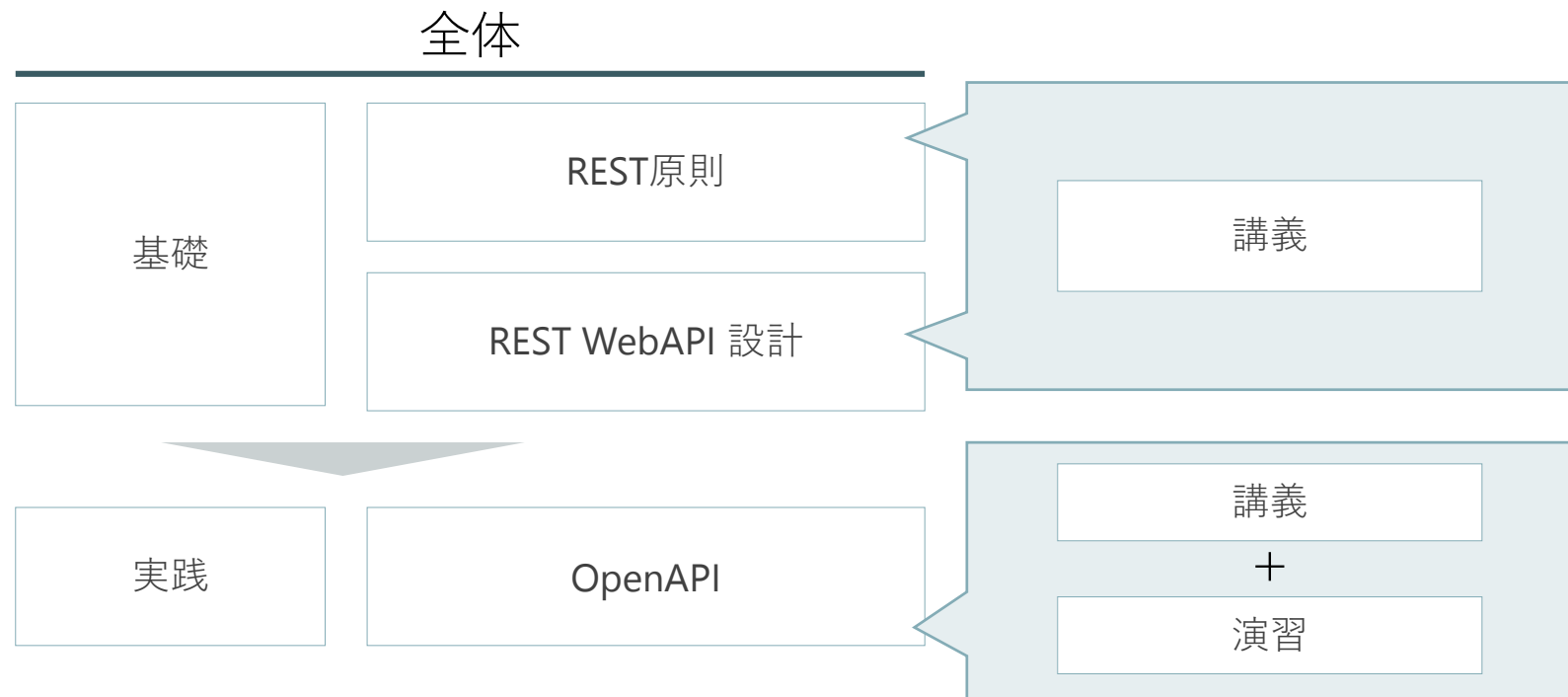
目次

1. はじめに
2. Webサービスの基本
3. REST制約
4. REST WebAPI サービス設計(基本)
5. REST WebAPI サービス設計(応用)
6. OpenAPI & Swagger基礎
7. おわりに

はじめに

学習の進め方

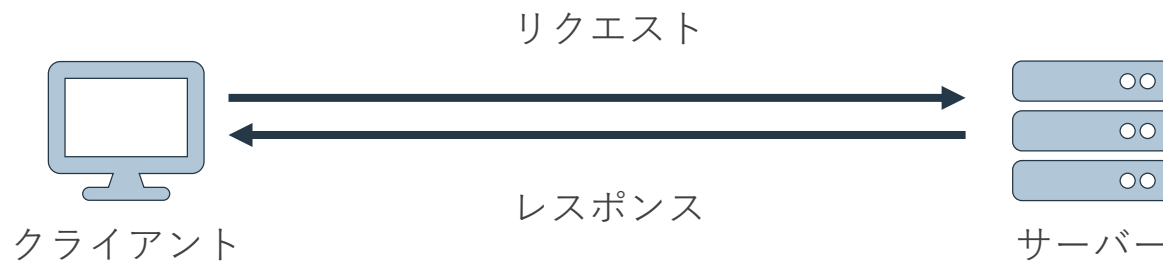
全体は「基礎→実践」の構成



Webサービスの基本

Webの仕組み

"クライアント"から"サーバー"へリクエスト、
"サーバー"から"クライアント"へレスポンスすることで情報伝達。



Webとは

“

HTTPなどのインターネット関連技術を利用してメッセージ送受信を行う技術，または それら技術を適用して展開されたサービス。

APIとは

Application Programming Interface



機能やデータを外部から呼び出して利用できるよう定めた規約

WebAPIとは

“

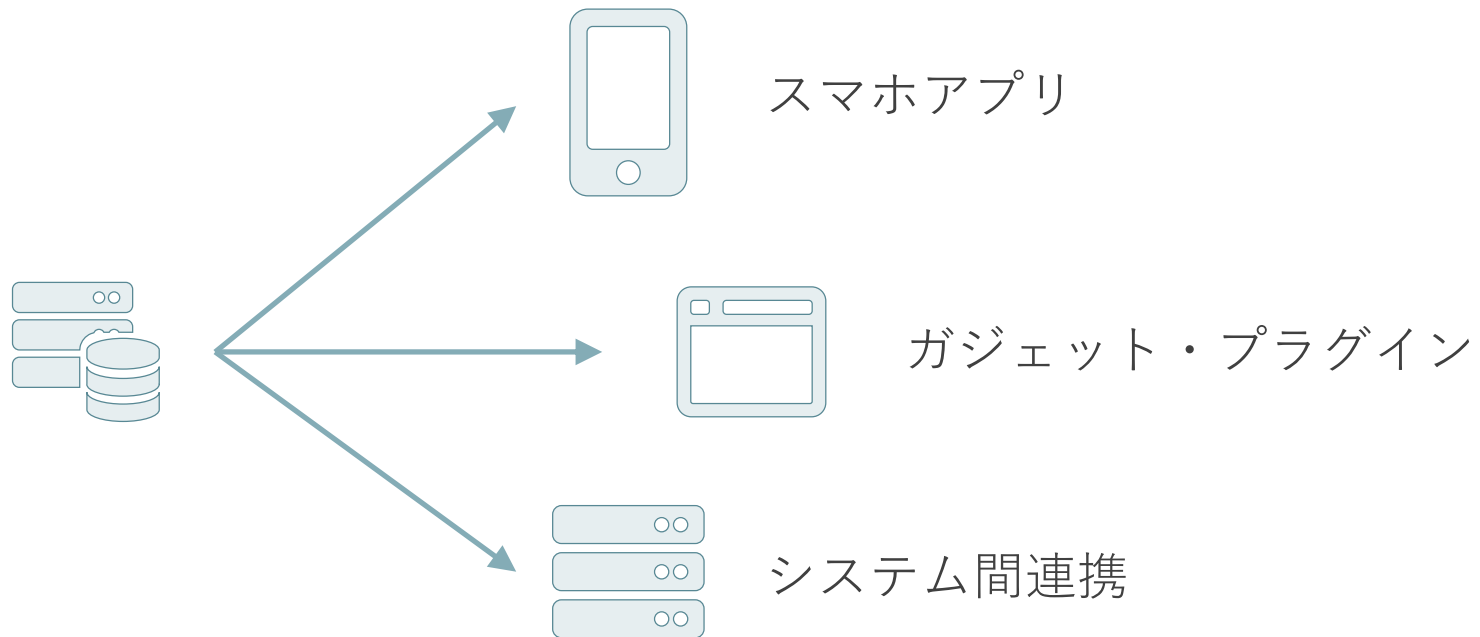
HTTPなどのインターネット関連技術を利用してプログラムが読み書きしやすい形でメッセージ送受信を行えるよう定義した規約，または規約を実装して展開されるサービス。

Webサイト、Webサービス、WebAPI

タイトル	内容
Webサイト	静的コンテンツ。 運営者からの情報提供が目的。ユーザーはサイト内を巡回することで目的を果たす。
Webサービス	動的コンテンツ。 ユーザーが抱える課題を解決することが目的。ユーザーはサービスに対して情報を溜め込む/引き出すといった操作を能動的に行うことで目的を果たす。
WebAPI	Webサービスで提供している機能やデータを外からプログラムが読み取りやすい形で利用できるよう定めた規約またはその実装。

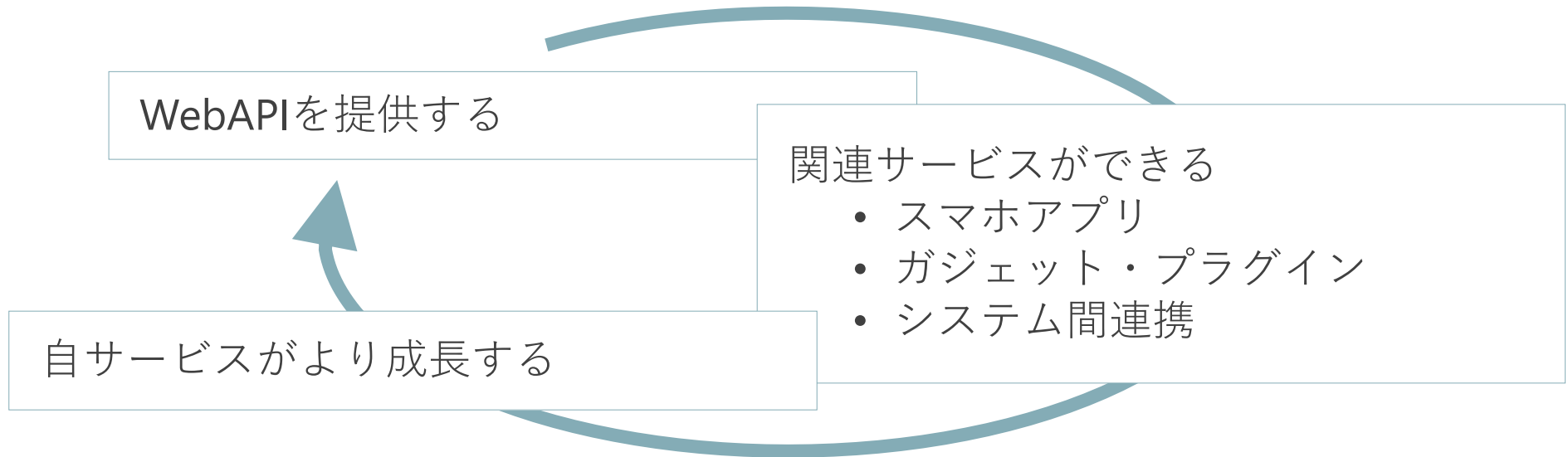
何ができるか

第3者が情報を利活用して新たな機能を開発。



何がうれしいか

“APIエコノミー”の実現による **自サービスの発展**。



何を公開するか

価値あるもの（機能やデータ）はすべて公開！

例）飲食店の口コミサイト

価値あるもの

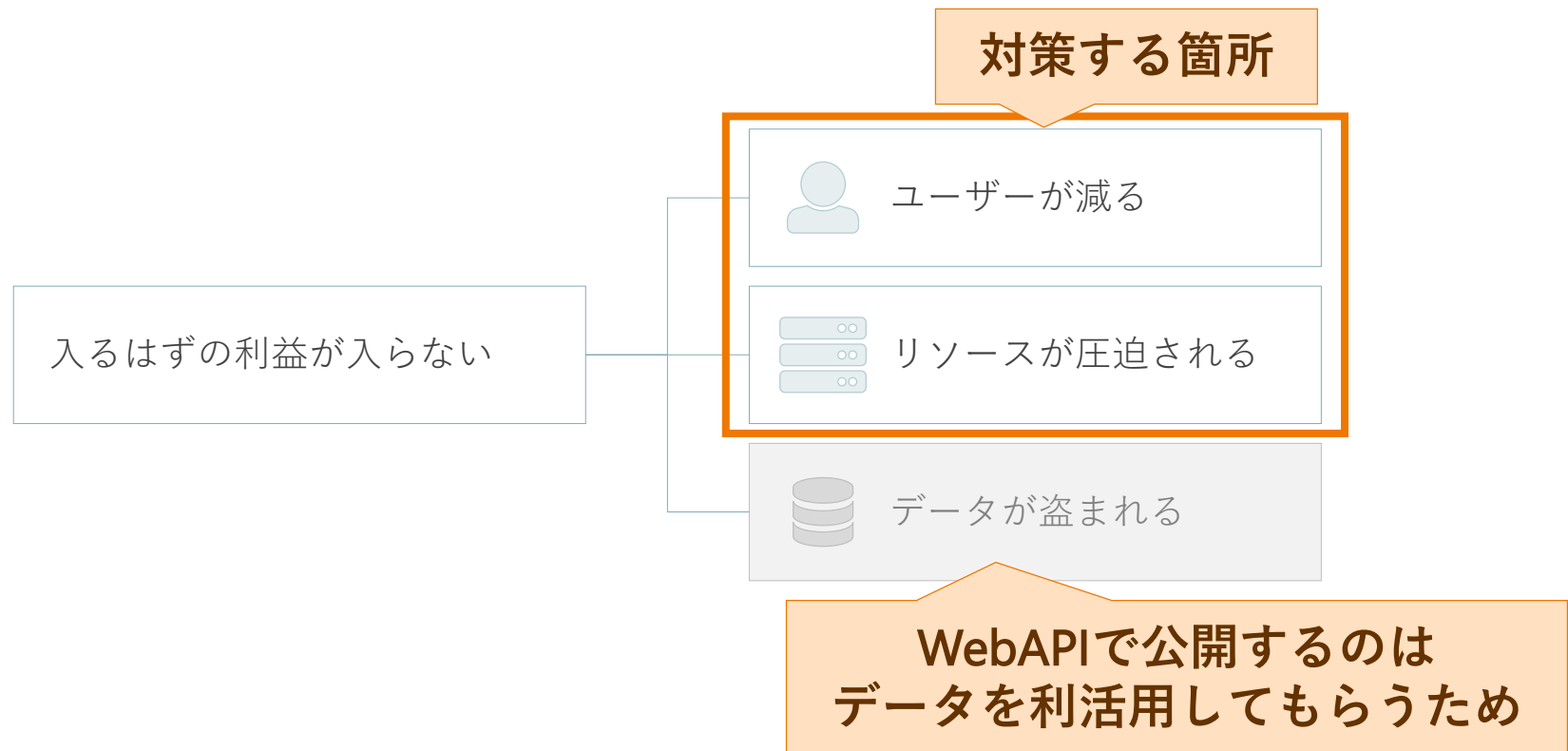
- 飲食店の検索、予約
- 人気店の検索
- 口コミの検索、投稿、修正、削除

価値ないもの

- 郵便番号検索

WebAPI公開によるリスクと対策

一番の問題は「入るはずの利益が入らなくなる」こと。



WebAPI公開によるリスクと対策

リスク	原因	対策
 ユーザーが減る	他サービスが優れている 作りこむ機能の優先順を間違っている	ユーザー獲得している他サービスの機能を取り込む
	他サービスが劣っている 自サービスの評判が落とされている	該当サービスに対して WebAPIの提供を停止する → APIキー停止
 リソースが圧迫される	API化することで機械的に データ取得しやすくなっている	該当サービスに対して WebAPIの利用制限をかける → レートリミット

どちらの対策もAPIキーを使った
利用者に対するアクセス制限

HTTP リクエスト

リクエストは3要素で構成される。

リクエストライン

```
POST https://www.post.japanpost.jp/cgi-zip/zipcode.php HTTP/1.1
```

ヘッダー

```
Host: www.post.japanpost.jp
```

```
Content-Length: 31
```

```
Content-Type: application/x-www-form-urlencoded
```

```
User-Agent: Mozilla/5.0 (...) Chrome/81.0.4044.138 Safari/537.36
```

```
Referer: https://www.post.japanpost.jp/index.html
```

```
Accept-Encoding: gzip, deflate, br
```

```
Accept-Language: ja,en-US;q=0.9,en;q=0.8
```

```
Cookie: ga=GA1.2.1386712802....
```

ボディ

```
pref=13&addr=%E6%B8%AF%E5%8C%BA
```


リクエストライン

リクエストラインは3要素で構成される。

```
POST https://www.post.japanpost.jp/cgi-zip/zipcode.php HTTP/1.1
```

メソッド

リクエストURI

HTTPバージョン

```
Host: www.post.japanpost.jp
Content-Length: 100
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (...) Chrome/81.0.4044.138 Safari/537.36
Referer: https://www.post.japanpost.jp/index.html
Accept-Encoding: gzip, deflate, br
Accept-Language: ja,en-US;q=0.9,en;q=0.8
Cookie: _ga=GA1.2.1386712802....

pref=13&addr=%E6%B8%AF%E5%8C%BA
```

メソッドの種類

メソッドは8種類。それぞれ意味がある。

メソッド	意味
OPTIONS	サーバー側が提供する機能の確認
GET	リソースの取得
HEAD	リソースのヘッダー（メタ情報）取得
POST	従属リソースの作成
PUT	新規リソースの作成、リソースの更新
DELETE	リソースの削除
TRACE	通信経路の確認
CONNECT	プロキシのトンネル接続

メソッドの種類

CRUDに相当する 4 メソッドが重要。

操作		メソッド	
Create	作成	POST	リソース名が未定
		PUT	リソース名が決まっている
Read	読み取り	GET	
Update	更新	PUT	
Delete	削除	DELETE	

ヘッダー

サーバーに対する追加情報を送信。

```
POST https://www.post.japanpost.jp/cgi-zip/zipcode.php HTTP/1.1
```

```
Host:
```

```
Con:
```

```
Content-Type: application/x-www-form-urlencoded
```

```
User-Agent: Mozilla/5.0 (...) Chrome/81.0.4044.138 Safari/537.36
```

```
Referer: https://www.post.japanpost.jp/index.html
```

```
Accept-Encoding: gzip, deflate, br
```

```
Accept-Language: ja,en-US;q=0.9,en;q=0.8
```

```
Cookie: _ga=GA1.2.1386712802....
```

```
pref=13&addr=%E6%B8%AF%E5%8C%BA
```

User-Agent にはクライアントのブラウザ情報が入る

ヘッダー

サーバーに対する追加情報を送信。

POST
Host: ...
Content-Length: 51
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (...) Chrome/81.0.4044.138 Safari/537.36
Referer: https://www.post.japanpost.jp/index.html
Accept-Encoding: gzip, deflate, br
Accept-Language: ja,en-US;q=0.9,en;q=0.8
Cookie: _ga=GA1.2.1386712802....

pref=13&addr=%E6%B8%AF%E5%8C%BA

ヘッダー

サーバーに対する追加情報を送信。

```
POST https://www.post.japanpost.jp/cgi-zip/zipcode.php HTTP/1.1
Host: www.post.japanpost.jp
Content-Length: 31
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (...) Chrome/81.0.4044.138 Safari/537.36
Referer: http://www.post.japanpost.jp/cgi-zip/zipcode.php
Accept-Encoding: gzip, deflate
Accept-Language: ja,en;q=0.8
Cookie: _ga=GA1.2.1386712802....
```

Cookie がよく聞くクッキーの正体

```
pref=13&addr=%E6%B8%AF%E5%8C%BA
```

ボディ

任意のデータを入れる。

```
POST https://www.post.japanpost.jp/cgi-zip/zipcode.php HTTP/1.1
Host: www.post.japanpost.jp
Content-Length: 31
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (...) Chrome/81.0.4044.138 Safari/537.36
Referer: https://www.post.japanpost.jp/index.html
Accept-Encoding: gzip, deflate, br
Accept-Language: ja,en-US;q=0.9,en;q=0.8
Cookie: _ga=GA1.2.1386712802....
```

pref=13&addr=%E6%B8%AF%E5%8C%BA

ボディ

HTTPレスポンス

レスポンスは3要素で構成される。

```
HTTP/1.1 200 OK
```

ステータスライン

```
Content-Type: application/json; charset=utf-8  
Date: Sun, 05 Jan 2020 02:44:28 GMT  
Server: scaffolding on HTTPServer2  
Cache-Control: private  
Content-Length: 1563
```

ヘッダー

```
{  
  "access_token": "ya29.I1-4B_UUTv4gEjys ... ydqtLCdgSMKb",  
  "expires_in": 3600,  
  "scope": "openid email profile",  
  "token_type": "Bearer",  
  "id_token": "eyJhbGciOiJIUzI1Ni5xDVYS ... M7nVlMQv8fD1wK_66TavSOQ"  
}
```

ボディ

ステータスライン

ステータスラインは3要素で構成される。

HTTPバージョン

HTTP/1.1 200 OK

ステータスコード

フレーズ

```
Content-Type: application/json; charset=utf-8
Date: 2020-05-15 12:28 GMT
Server: Server2
Cache-Control: private
Content-Length: 1563

{
  "access_token": "ya29.I1-4B_UUTv4gEjys ... ydqtLCdgSMKbeGujDqHL8w",
  "expires_in": 3600,
  "scope": "openid email profile",
  "token_type": "Bearer",
  "id_token": "eyJhbGciOiJIUzI1Ni5xDVYS ... M7nVlMQv8fD1wK_66TavSOQ"
}
```

ステータスコード

ステータスは大きく5種類。それぞれ意味がある。

ステータス コード	分類	意味
1xx	Informational	リクエストは受け入れられたので処理を継続。
2xx	Success	リクエストが受け入れられて正常処理された。
3xx	Redirection	リクエスト完了のために追加操作が必要。
4xx	Client Error	リクエストに誤りがある。
5xx	Server Error	サーバー処理失敗。

ヘッダー

ステータスラインで表現できない追加情報を返却。

Content-Type には応答データのフォーマットが入る

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json; charset=utf-8
```

```
Date: Sun, 05 Jan 2020 02:44:28 GMT
```

```
Server: scaffolding on HTTPServer2
```

```
Cache-Control: private
```

```
Content-Length: 1563
```

```
{  
  "access_token": "ya29.I1-4B_UUTv4gEjys ... ydqtLCdgSMKbeGujDqHL8w",  
  "expires_in": 3600,  
  "scope": "openid email profile",  
  "token_type": "Bearer",  
  "id_token": "eyJhbGciOiJIUzI1Ni5xDVYS ... M7nVlMQv8fD1wK_66TavSOQ"  
}
```

ヘッダー

ステータスラインで表現できない追加情報を返却。

```
HTTP/1.1 200 OK
```

Cache-Control にはデータをキャッシュしてよいかどうかが入る

```
Server: scott/0.0.1 HTTPServer2
```

```
Cache-Control: private
```

```
Content-Length: 1563
```

```
{  
  "access_token": "ya29.I1-4B_UUTv4gEjys ... ydqtLCdgSMKbeGujDqHL8w",  
  "expires_in": 3600,  
  "scope": "openid email profile",  
  "token_type": "Bearer",  
  "id_token": "eyJhbGciOiJIUzI1Ni5xDVYS ... M7nVlMQv8fD1wK_66TavSOQ"  
}
```

ボディ

任意のデータが入る。

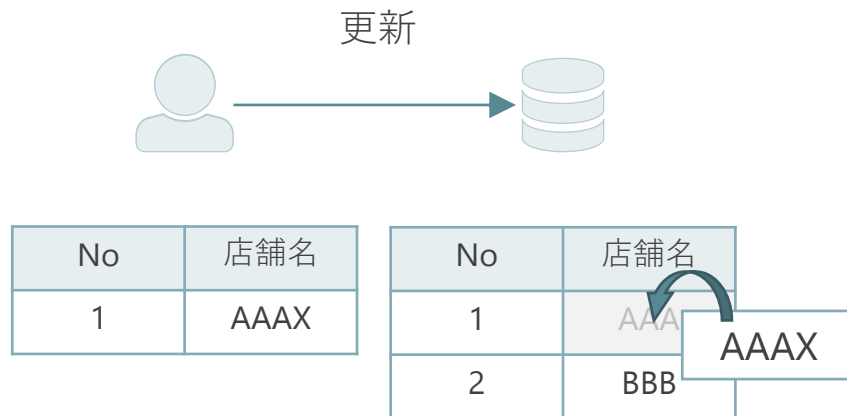
```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sun, 05 Jan 2020 02:44:28 GMT
Server: scaffolding on HTTPServer2
Cache-Control: private
Content-Length: 1563
```

```
{
  "access_token": "ya29.I1-4B_UUTv4gEjys ... ydqtLCdgSMKbeGujDqHL8w",
  "expires_in": 3600,
  "scope": "openid email profile",
  "token_type": "Bearer",
  "id_token": "eyJhbGciOiJIUzI1Ni5xDVYS ... M7nVlMQv8fD1wK_66TavSOQ"
}
```

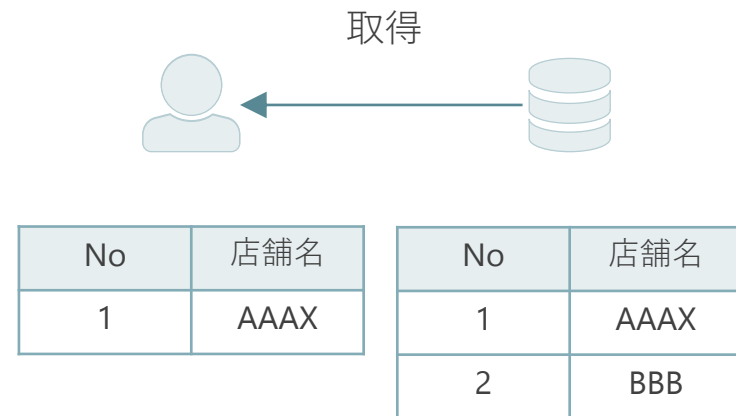
副作用

リソース（データ）が改変されること。

副作用がある



副作用がない



REST 制約

RESTfulとは

RESTful

II

「RESTで求められる原則に従っている」 こと

RESTとは

REpresentational SState TTransfer

RESTとは

「分散型システムにおける設計原則群」

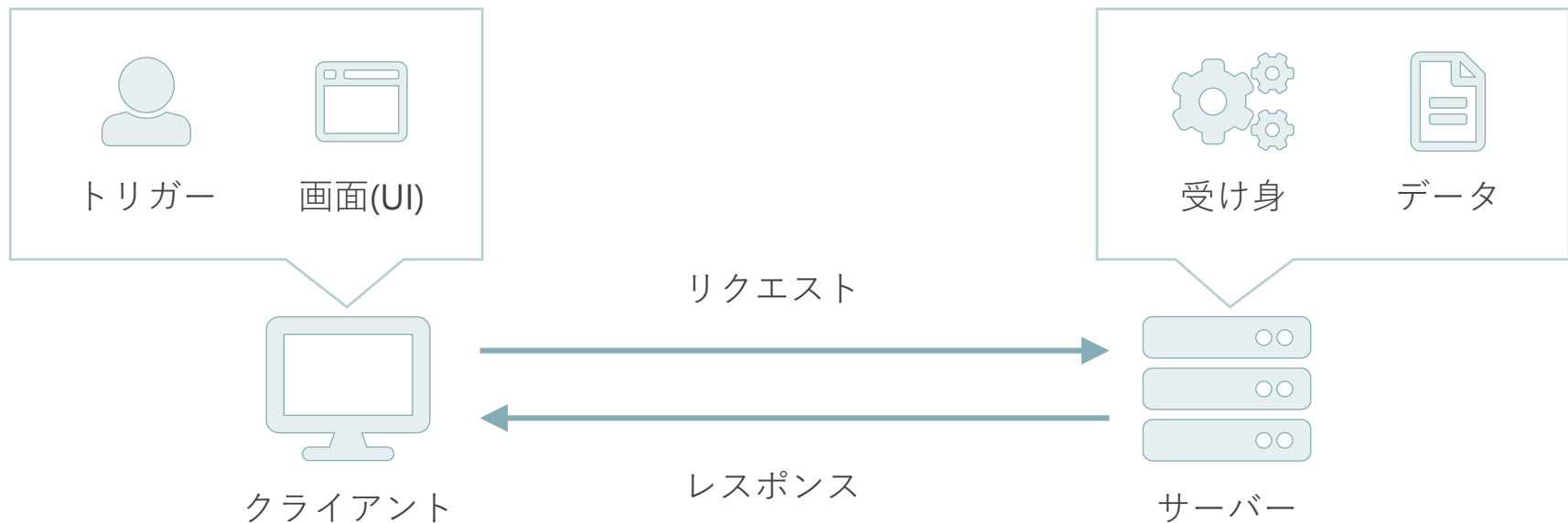
REST原則

6

- クライアント/サーバー
- ステートレス
- キャッシュ制御
- 統一インターフェース
- 階層化システム
- コードオンデマンド

クライアント/サーバー

- ✓ ネットワークベースのアプリケーションではよくある構成。
- ✓ “画面（UI）”と“データ”で関心事を分離。
- ✓ クライアント側がトリガー、サーバー側は受け身。



REST原則

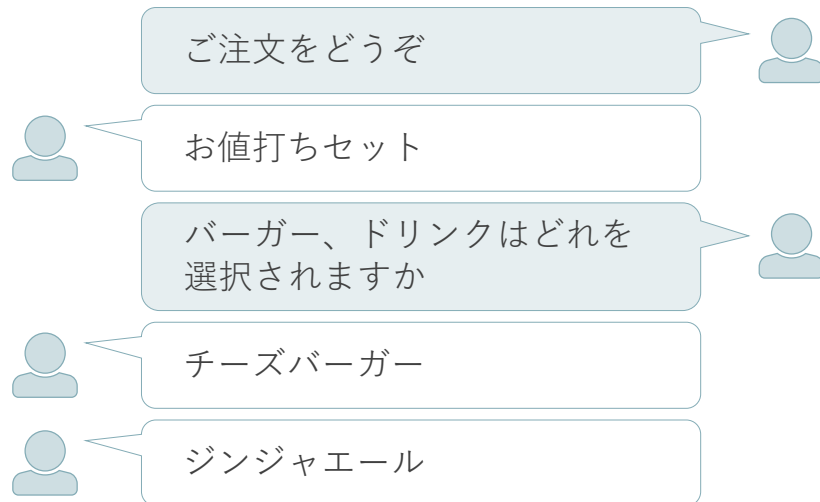
6

- クライアント/サーバー
- ステートレス
- キャッシュ制御
- 統一インターフェース
- 階層化システム
- コードオンデマンド

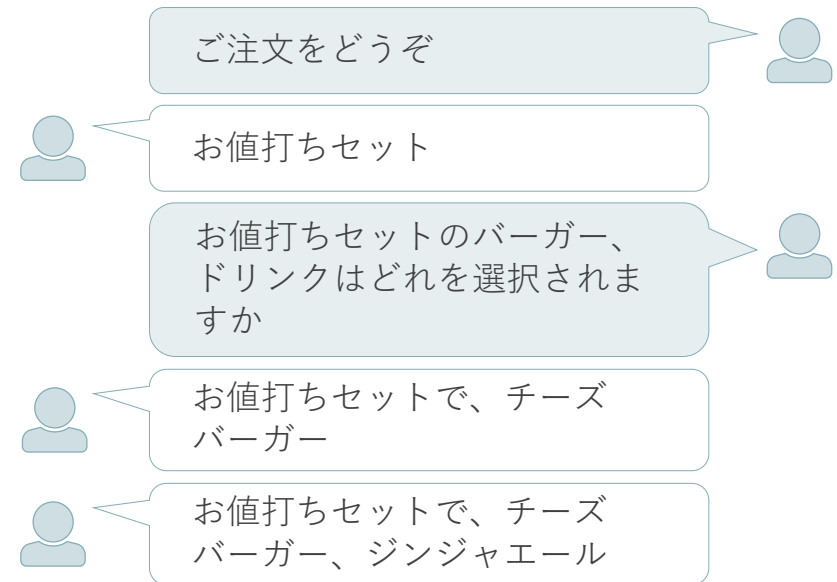
ステートフルとステートレス

某ハンバーガーショップでハンバーガーを買うケース。

ステートフル



ステートレス



**前の状態を保存しない
それぞれの会話が独立して成立**

ステートレス

サーバーはリクエストだけでコンテキストを理解できる。

- サーバーに保存されたコンテキスト情報は使わない（サーバーセッションは使わない）。
- 状態はクライアント上に保存される（リクエストにすべて含める）。

リクエストに状態情報を含むことで
リクエストが独立する



ステートレス

メリット

- 単一のリクエスト以外見る必要がないので、監視が容易。
- 障害発生したリクエストだけ回復すればよいので、障害復旧が容易。
- リクエスト全体でサーバーリソースを共有する必要がないのでスケールが容易。

デメリット

- 単一のリクエストで完結させるため、リクエストデータに重複がある。
- アプリを複数バージョン同時提供し、状態をクライアントに置いておくとアプリ制御が複雑になる。

REST原則

6

- クライアント/サーバー
- ステートレス
- キャッシュ制御
- 統一インターフェース
- 階層化システム
- コードオンデマンド

キャッシュ制御

クライアントはレスポンスをキャッシュできる。

- レスポンスは明示的または暗黙的にキャッシュ可能。
- キャッシュを適切に行うことでクライアント/サーバー間の通信が排除され、ユーザー体験の向上、リソース効率の向上、拡張性の向上が見込める。

キャッシュを適切に行うことで
必要な情報だけリクエストすればよくなる



キャッシュ制御

メリット

- ユーザー体験の向上
- リソース効率の向上
- 拡張性の向上

デメリット

- 古いデータを戻してしまうとシステムに対する信頼性の低下につながる

REST原則

6

- クライアント/サーバー
- ステートレス
- キャッシュ制御
- 統一インターフェース
- 階層化システム
- コードオンデマンド

統一インターフェース

4つの制約。

制約

リソースの識別

表現を用いたリソース操作

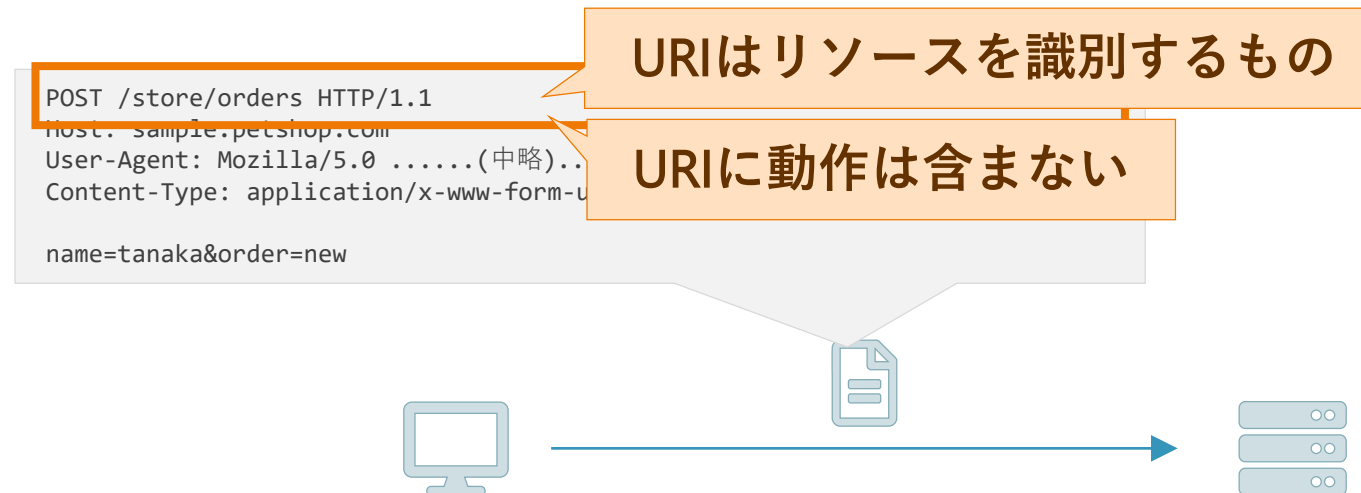
自己記述メッセージ

アプリケーション状態エンジンとしてのハイパーメディア（HATEOAS）

リソースの識別

URIを用いてサーバーに保存されたデータを識別する。

- 名前が付けられるあらゆるものがリソース。
例：ドキュメント、画像、人、情報、サービス、状態
- 抽象的な定義も含む。
 - 「ある断面」（4月1日の天気）、「最新」（今日の天気）



表現を用いたリソース操作

断面情報を利用してサーバー上のデータを操作する。

- リソースのある断面が「表現」。
- クライアントからサーバーへ編集リクエストをする際、認証情報などの追加情報を付与する

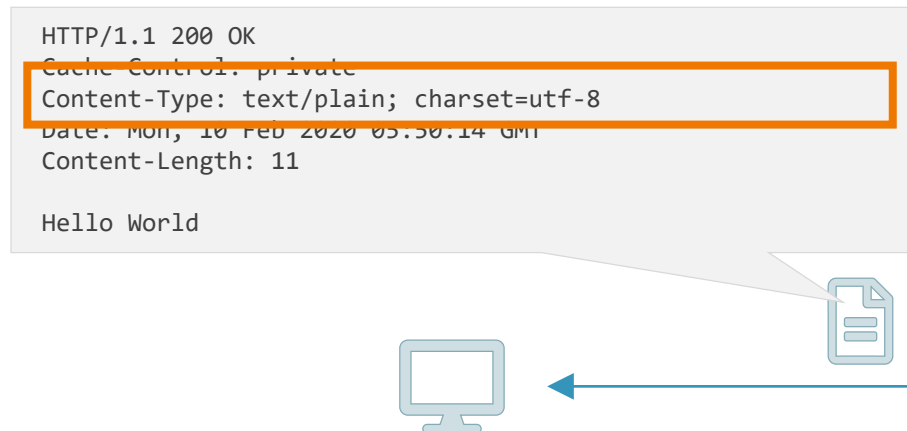
```
GET /yconnect/v1/attribute?schema=openid HTTP/1.1
Host: user-info.yahooapis.jp
Authorization: Bearer 20BRckZjTT5a3aKD.....(中略).....mT1oULUNOuUlw3
User-Agent: Mozilla/5.0 .....(中略)..... Safari/537.36
Content-Type: application/x-www-form-urlencoded
```



自己記述メッセージ

メッセージ内容が何であるか、ヘッダーに記述されている。

- レスポンスに含まれるメタ情報（ヘッダー情報）で内容がどのようなものかわかる。



HATEOAS

Hypermedia as the Engine of Application State

- レスポンスに現在の状態を踏まえて関連するハイパーリンクが含まれている。
例：検索結果ページにおける「次のページ」

```
{  
  message: "Hello World",  
  links: [  
    { name: "next", url: "http://sample.com/message/3" },  
    { name: "prev", url: "http://sample.com/message/5" }  
  ]  
}
```



統一インターフェース

メリット

- システムアーキテクチャ全体が簡素化されてわかりやすくなる。
- 提供するサービスに集中でき、独自の進化ができる。
- 異なるブラウザでも同じような画面を表示できる。

デメリット

- 標準化によって効率が犠牲になる。

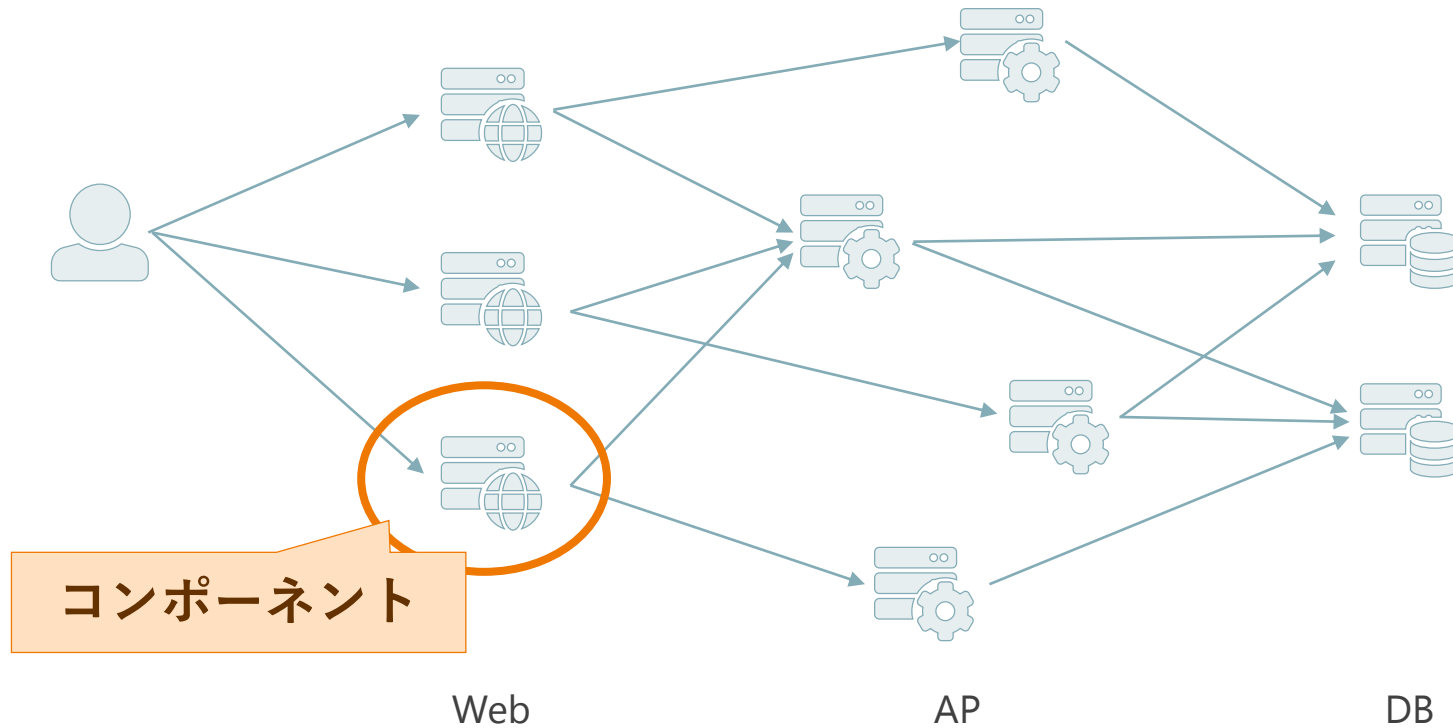
REST原則

6

- クライアント/サーバー
- ステートレス
- キャッシュ制御
- 統一インターフェース
- 階層化システム
- コードオンデマンド

階層化システム

多層アーキテクチャ構成。



階層化システム

メリット

- 各システム（コンポーネント）に役割を決めて独立させることで、進化と再利用が促進できる

デメリット

- データ処理にオーバーヘッドが発生
→ユーザーから見ると応答が悪く見える

REST原則

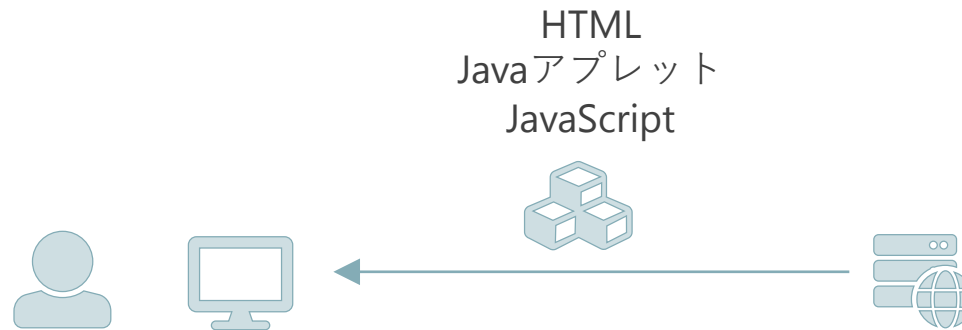
6

- クライアント/サーバー
- ステートレス
- キャッシュ制御
- 統一インターフェース
- 階層化システム
- コードオンデマンド

コードオンデマンド

クライアントコードをダウンロードして実行できる。

- リリース後にクライアントコードを変更できる。
例：HTML自体の更新、Javaアプレットの更新



コードオンデマンド

メリット

- リリース済みのクライアントに対して機能追加ができる。
- サーバーの負荷が下がる（＝クライアントに処理が移譲できるため）。

デメリット

- 評価環境が複雑になる（多数のブラウザとか...）。

REST API 設計レベル

設計レベルは4段階。



LEVEL 0 : HTTPを使っている

REST API の基本レベル。RPCスタイルのXML通信。

- HTTPは単なる通信手段として利用。
- 1 URLですべて完結。
- リクエストボディに処理と引数が含まれる。

```
POST /rpc HTTP 1.0
... (省略) ...

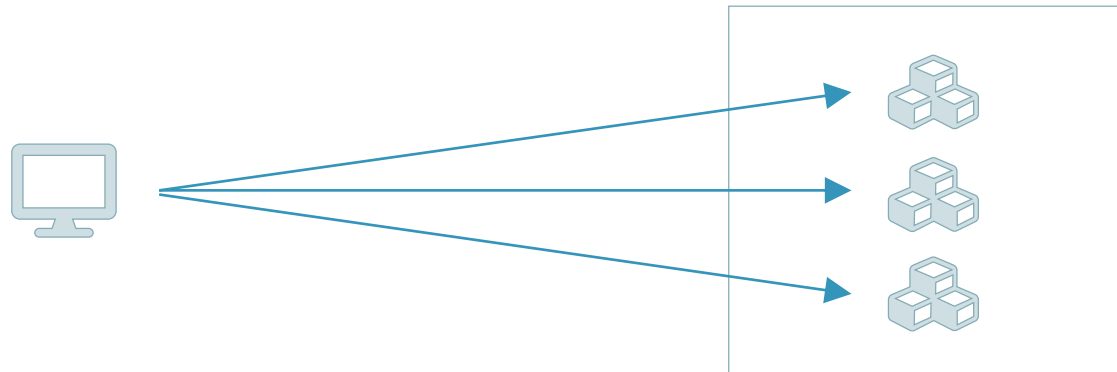
<?xml version="1.0"?>
<methodCall>
  <methodName>circleArea</methodName>
  <params><param><value><double>1.24</double></value></param></params>
</methodCall>
```



LEVEL 1 : リソースの概念を導入

リソースごとにURLを分割。

- リソースごとにURLを分離。
- HTTPメソッドは活用できていないので、GETかPOSTのみで通信。



LEVEL 2 : HTTPの動詞を導入

LEVEL1に加えてHTTPメソッドを活用。

- リソースに対してHTTPメソッドを使ったCRUD操作が行われている。

操作	CRUD	HTTPメソッド
登録	Create	POST, PUT
取得	Read	GET
更新	Update	PUT
削除	Delete	DELETE

LEVEL 3 : HATEOASの概念を導入

LEVEL2に加えてレスポンスにリソース間のつながりが含まれる。

- レスポンスに現在の状態に関連するハイパーリンクが含まれている。
(=HATEOASに相当する情報がレスポンスに含まれている)

```
{  
  message: "Hello World",  
  links: [  
    { name: "next", url: "http://sample.com/message/3" },  
    { name: "prev", url: "http://sample.com/message/5" }  
  ]  
}
```



REST WebAPI サービス設計(基本)

リクエスト設計時のポイント

- URI設計
 - 短く入力しやすい（冗長なパスを含まない）
 - 人間が読んで理解できる（省略しない）
 - 大文字小文字が混在していない（すべて小文字）
 - 単語はハイフンでつなげる
 - 単語は複数形を利用する
 - エンコードを必要とする文字を使わない
 - サーバー側のアーキテクチャが反映されていない
 - 改造しやすい（Hackable）
 - ルールが統一されている
- HTTPメソッドの適用
- クエリパラメータとパスの使い分け

URI設計で考慮すること

短く入力しやすい（冗長なパスを含まない）

⇒シンプルで覚えやすいものにすることで入力ミスを防ぐ



GET `http://api.example.com/service/api/search`



GET `http://api.example.com/search`

URI設計で考慮すること

人間が読んで理解できる（できるだけ省略しない）

⇒その省略はあなたの自己満足ではない？

国や文化が変わっても不変な表記にすることで
誤認識を防ぐ



GET `http://api.example.com/sv/u`



GET `http://api.example.com/users`

URI設計で考慮すること

大文字小文字が混在していない（すべて小文字）

⇒APIをわかりやすく、間違いにくくするためには統一が必要
統一するなら一般的に小文字



GET `http://api.example.com/Users`



GET `http://api.example.com/users`

URI設計で考慮すること

単語はハイフンでつなげる

⇒アンダースコアはタイプライターで下線を引くためのもの
ハイフンは単語をつなぐためのもの



GET `http://api.example.com/popular_users`



GET `http://api.example.com/popular-users`



GET `http://api.example.com/users/popular`

単語の連結をするくらいなら
そもそもURIを見直す

URI設計で考慮すること

単語は複数形を利用する

⇒URIで表現しているのは「リソースの集合」



GET `http://api.example.com/user/12345`



GET `http://api.example.com/users/12345`

URI設計で考慮すること

エンコードを必要とする文字を使わない

⇒URIから意味が理解できない



GET http://api.example.com/ユーザー



GET http://api.example.com/%E3%83%A6%E3%83%BC%E3%82%B6%E3%83%BC



GET http://api.example.com/users

URI設計で考慮すること

サーバー側のアーキテクチャを反映しない

⇒悪意あるユーザーに脆弱性を突かれる危険がある



GET `http://api.example.com/cgi-bin/get_user.php?id=12345`



GET `http://api.example.com/users/12345`

URI設計で考慮すること

改造しやすい (Hackable)

⇒システム依存の設計は意味が理解できない



GET `http://api.example.com/items/alpha/12345`



GET `http://api.example.com/items/beta/23456`



GET `http://api.example.com/items/12345`

URI設計で考慮すること

ルールが統一されている

⇒一定のルールに従って設計することで間違いを防ぐ



友達情報取得

```
GET http://api.example.com/friends?id=12345
```

メッセージ投稿

```
POST http://api.example.com/friends/12345/message
```



友達情報取得

```
GET http://api.example.com/friends/12345
```

メッセージ投稿

```
POST http://api.example.com/friends/12345/messages
```


HTTPメソッドとURI

URI がリソースを示すのに対し

HTTPメソッドはリソースに対する操作を示す



HTTP/1.1 メソッド（主要なもの）

メソッド名	説明
GET	リソースの取得
POST	リソースの新規登録
PUT	既存リソースの更新 / リソースの新規登録
DELETE	リソースの削除

たとえば...

URIは同じでHTTPメソッドを変えることで操作を変える

操作	API 実装 例	
ユーザー情報一覧取得	GET	<code>http://api.example.com/users</code>
ユーザーの新規登録	POST	<code>http://api.example.com/users</code>
特定ユーザーの取得	GET	<code>http://api.example.com/users/12345</code>
ユーザーの更新	PUT	<code>http://api.example.com/users/12345</code>
ユーザーの削除	DELETE	<code>http://api.example.com/users/12345</code>

リソースを特定するパラメータ

絞り込みの方法には2種類ある。

種類	概要
クエリパラメータ	<p>URLの末尾にある"?"に続くキーバリュー。</p> <pre>GET http://api.example.com/users?page=3</pre>
パスパラメータ	<p>URL中に埋め込まれるパラメータ。</p> <pre>GET http://api.example.com/users/123</pre>

クエリとパスの使い分け

クエリパラメータとするかどうかの判断基準

- 一意なリソースを表すのに必要かどうか
⇒パスパラメータを利用
- 省略可能かどうか
⇒クエリパラメータを利用

(例) 検索条件（絞り込み条件）はパスに含めない



```
GET http://api.example.com/users?name=tanaka
```

レスポンス設計時のポイント

- ステータスコード
- データフォーマット
- データの内部構造
 - データの内部構造
 - エンベロープは使わない
 - オブジェクトはできるだけフラットにする
 - ページネーションをサポートする情報を返す
 - プロパティの命名規則はAPI全体で統一する
 - 日付はRFC3339 (W3C-DTF) 形式を使う
 - 大きな数値 (64bit整数) は文字列で返す
- エラー表現
 - エラー詳細はレスポンスボディに入れる
 - エラーの際にHTMLが返らないようにする
 - サービス閉塞時は"503" + "Retry-After"

ステータスコードの分類

処理結果の概要は 5 分類

ステータスコード	意味
100番台	情報
200番台	成功
300番台	リダイレクト
400番台	クライアントサイドに起因するエラー
500番台	サーバーサイドに起因するエラー

1xx : 情報

種類	説明
100 Continue	サーバーがリクエストの最初の部分を受け取り、まだサーバーから拒否されていないことを示す。
101 Switching Protocol	プロトコルの切り替え要求を示す。

2xx：成功

種類	説明
200 OK	リクエストが成功したことを示す。 本文にデータが含まれる。
201 Created	リクエストが成功し、新しいリソースが作成されたことを示す。 ヘッダーの Location に新しいリソースへのURLを含める。
202 Accepted	非同期ジョブを受け付けたことを示す。 実際の処理結果は別途受け取る。
204 No Content	リクエストは成功したが、レスポンスデータがないことを示す。 クライアント側のビューを変更する必要がないことを意味する。

3xx：リダイレクト

種類	説明
301 Moved Permanently	リクエストされたリソースが永久に変更されたことを示す。 レスポンスヘッダーにLocationを設定する。
302 Found	リクエストされたリソースが一時的に変更されたことを示す。 レスポンスヘッダーにLocationを設定する。
303 See Other	リクエストされたリソースを取得するため、別のURLにGETリクエストでアクセスすることを示す。
308 Permanent Redirect	リクエストされたリソースが永久に変更されたことを示す。 リダイレクト時はリクエストと同じメソッドを利用する。
307 Temporary Redirect	リクエストされたリソースが一時的に変更されたことを示す。 リダイレクト時はリクエストと同じメソッドを利用する。
304 Not Modified	前回リクエスト時から変更がないことを示す。 レスポンスボディは空になる。

4xx：クライアントサイドエラー

種類	説明
400 Bad Request	その他のエラー。
401 Unauthorized	認証されていないことを示す。
403 Forbidden	リソースに対するアクセスが許可されていない（認可されていない）ことを示す。
404 Not Found	リクエストされたリソースが存在しないことを示す。
409 Conflict	リソースが競合して処理が完了できなかったことを示す。
429 Too Many Requests	アクセス回数が制限回数を超えたため処理できなかったことを示す。

存在を隠したい場合
404 Not Found にする

「レートリミット」と呼ばれる

5xx：サーバーサイドエラー

種類	説明
500 Internal Server Error	サーバーサイドのアプリケーションエラーが発生したことを示す。
503 Service Unavailable	サービスが一時的に利用できないことを示す。 メンテナンス期間や過負荷で応答できないようなケース。

HTTPメソッドとステータスコード

ステータスコード		GET	POST	PUT	DELETE
2xx	200 OK	<input type="radio"/>	<input type="radio"/> データあり	<input type="radio"/> データあり	<input type="radio"/>
	201 Created	<input type="radio"/>	<input type="radio"/> データなし	<input type="radio"/> 新規作成 データなし	<input type="radio"/>
	202 Accepted	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	204 No Content	<input type="radio"/>	<input type="radio"/>	<input type="radio"/> 更新 データなし	<input type="radio"/>
3xx	304 Not Modified	<input type="radio"/> キャッシュ	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
4xx	400 Bad Request	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	401 Unauthorized	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	403 Forbidden	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	404 Not Found	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	409 Conflict	<input type="radio"/>	<input type="radio"/> データ重複など	<input type="radio"/> ロックなど	<input type="radio"/>
	429 Too Many Requests	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
5xx	500 Internal Server Error	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	503 Service Unavailable	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

レスポンスのデータフォーマット

主要なレスポンスフォーマットは3種類。

フォーマット	サンプル
XML	<pre><user> <name>tanaka</name> </user></pre>
JSON	<pre>{ user: { name: "tanaka" } }</pre>
JOSNP	<pre>callback({user: {name: "tanaka"}})</pre>

XMLフォーマット

サンプル

Content-Type: application/xml

```
<user>
  <name lang="ja">
    <first>tsuyoshi</first>
    <last>tanaka</last>
  </name>
  <dob>1994/03/30</dob>
</user>
```

特徴

- テキスト形式
- タグで記述
- タグは入れ子にできる
- タグに属性が付けられる

JSONフォーマット

サンプル

Content-Type: application/json

```
{  
  name: {  
    first: "tsuyoshi",  
    last: "tanaka"  
  },  
  dob: "1994/03/30"  
}
```

特徴

- テキスト形式
- JavaScriptを元にしたフォーマット
- XMLに比べてデータ量が減らせる（=XMLのタグは末尾にも同じ文字が必要なため冗長）
- オブジェクトは入れ子にできる

JSONPフォーマット

サンプル

Content-Type: application/javascript

```
callback({
  name: {
    first: "tsuyoshi",
    last: "tanaka"
  },
  dob: "1994/03/30"
});
```

特徴

- テキスト形式
- データフォーマットのように見えるが「JavaScriptコード」
- クロスドメインでデータを受け渡すことができる

データフォーマットの指定方法

データフォーマットの指定方法は3種類。

フォーマット	サンプル
クエリパラメータ	<code>http://api.sample.com/v1/users?format=json</code> 実サービスで利用が多い
拡張子	<code>http://api.sample.com/v1/users.json</code>
リクエストヘッダー	<code>GET http://api.sample.com/v1/users</code> <code>Host: api.sample.com</code> <code>Accept: application/json</code> URIがリソースであることを考えると リクエストヘッダーが一番お行儀が良い

データ内部構造の設計で考慮すること

エンベロープは使わない

⇒ヘッダー情報と役割が被るのでエンベロープは使わない



```
HTTP/1.1 200 OK
Content-Type: text/html
... 省略 ...

{
  "header": {
    "status": "success",
    "errorCode": 0,
  },
  "response": {
    name: "Tanaka Tsuyoshi"
  }
}
```



```
HTTP/1.1 200 OK
Content-Type: text/html
... 省略 ...

{
  name: "Tanaka Tsuyoshi"
}
```

データ内部構造の設計で考慮すること

オブジェクトはできるだけフラットにする

⇒レスポンス容量を減らすため



```
{
  "id": "12345",
  "name": "Tsuyoshi Tanaka",
  "profile": {
    "birthday": "3/23",
    "gender": "male"
  }
}
```



```
{
  "id": "12345",
  "name": "Tsuyoshi Tanaka",
  "birthday": "3/23",
  "gender": "male"
}
```

データ内部構造の設計で考慮すること

ページネーションをサポートする情報を返す

⇒情報更新される可能性があるため



```
{
  "users": [
    {
      "id": "12345",
      "name": "Tsuyoshi Tanak"
    },
    ...
  ],
  "nextPage": 1
}
```



```
{
  "users": [
    {
      "id": "12345",
      "name": "Tsuyoshi Tanak"
    },
    ...
  ],
  "hasNext": true,
  "nextPageToken": "FqTt82Cp"
}
```

次をどこから取得するのか
キーとなる情報を返す

"hasNext": true,
"nextPageToken": "FqTt82Cp"

プロパティの命名規則はAPI全体で統一する

プロパティの命名規則はAPI全体で統一する





⇒利用者が混乱するため

種類	サンプル
スネークケース	snake_case
キャメルケース	camelCase
パスカルケース	PascalCase

実例だと
スネークケースかキャメルケース
が同じくらいなのでどちらかでOK

データ内部構造の設計で考慮すること

日付はRFC3339 (W3C-DTF) 形式を使う
⇒インターネットで標準的に用いられるため

種類	サンプル
 RFC822 (RFC1123)	Thu, 29 Mar 2018 08:00:00 GMT
 RFC850	Thursday, 29-Mar-2018 08:00:00 GMT
 Unixタイムスタンプ	1521781500
 RFC3339 (W3C-DTF)	2018-03-29T17:00:00+09:00

データ内部構造の設計で考慮すること

大きな数値（64bit整数）は文字列で返す

⇒通常の整数は32bit整数で64bit整数は処理できないため



```
{  
  "val": 123456789012345678  
}
```



```
{  
  "val": 123456789012345678  
  "val_str": "123456789012345678",  
}
```


エラー表現で考慮すること

エラー詳細はレスポンスボディに入れる

⇒足りない情報はレスポンスボディに追加する



```
HTTP/1.1 400 Bad Request
Server: api.example.com
Date: Sat, 28 Mar 2020 01:57:25 GMT
Content-Type: application/json
Content-Length: 0
```



```
HTTP/1.1 400 Bad Request
Server: api.example.com
Date: Sat, 28 Mar 2020 01:57:25 GMT
Content-Type: application/json
Content-Length: 77
```

```
{
  "code": "1234567890"
  "message": "不正な検索条件です"
}
```

エラー表現で考慮すること

エラーの際にHTMLが返らないようにする

⇒レスポンスフォーマットが変わると
クライアントアプリ側で処理できないケースがある



```
HTTP/1.1 404 Not Found
Server: api.example.com
Date: Sat, 28 Mar 2020 01:57:25 GMT
Content-Type: text/html
Content-Length: 17707
```

```
<!DOCTYPE html>
<html lang="ja">
<head>
...
```



```
HTTP/1.1 404 Not Found
Server: api.example.com
Date: Sat, 28 Mar 2020 01:57:25 GMT
Content-Type: application/json
Content-Length: 74
```

```
{
  "code": "2345678901"
  "message": "リソースが存在しません"
}
```

エラー表現で考慮すること

サービス閉塞時は "503" + "Retry-After"

⇒クライアント側から見ていつから再開してよいかわかる



```
HTTP/1.1 404 Not Found
Server: api.example.com
Date: Sat, 28 Mar 2020 01:57:25 GMT
Content-Type: text/html
Content-Length: 17707
```

```
<!DOCTYPE html>
<html lang="ja">
<head>
...
```



```
HTTP/1.1 503 Service Temporary
Unavailable
Server: api.example.com
Date: Sat, 28 Mar 2020 01:57:25 GMT
Content-Type: application/json
Content-Length: 87
Retry-After: Mon, 6 Apr 2020 01:00:...
```

```
{
  "code": "3456789012"
  "message": "サービス利用できません"
}
```

REST WebAPI サービス設計(応用)

APIにバージョンを含めるか？


APIにバージョンを含める場合の観点について考えてみましょう

メリット

特定バージョン指定でアクセスできるので、クライアント側で突然エラーにはならない

デメリット

複数バージョンを並列稼働させるため、ソースコードやデータベースの管理が複雑になる



広く世間一般に公開するようなサービスを展開するのであれば
利用者の利便性を考慮してAPIバージョンを含めたURLの設計を行う

バージョンを入れる場所

バージョンを入れる場所は3種類

どのパターンも実例としては存在するが
「パス」のケースが多い

パス	<code>http://api.example.com/v1/users/</code>
クエリ	<code>http://api.example.com/users?version=1</code>
ヘッダー	<code>GET http://api.example.com/users</code> <code>X-API-Version: 1</code>

2012年6月以降は“X-接頭辞”は非推奨
サービス固有の接頭辞をつける
(例: GData-Version)

バージョンの付け方

「セマンティックバージョニング」がよく知られている

サンプル

バージョン：	1	.	2	.	3
	メジャー		マイナー		パッチ

バージョンを上げるルール

位置	ルール
メジャー	後方互換しない修正
マイナー	後方互換する機能追加
パッチ	後方互換するバグ修正

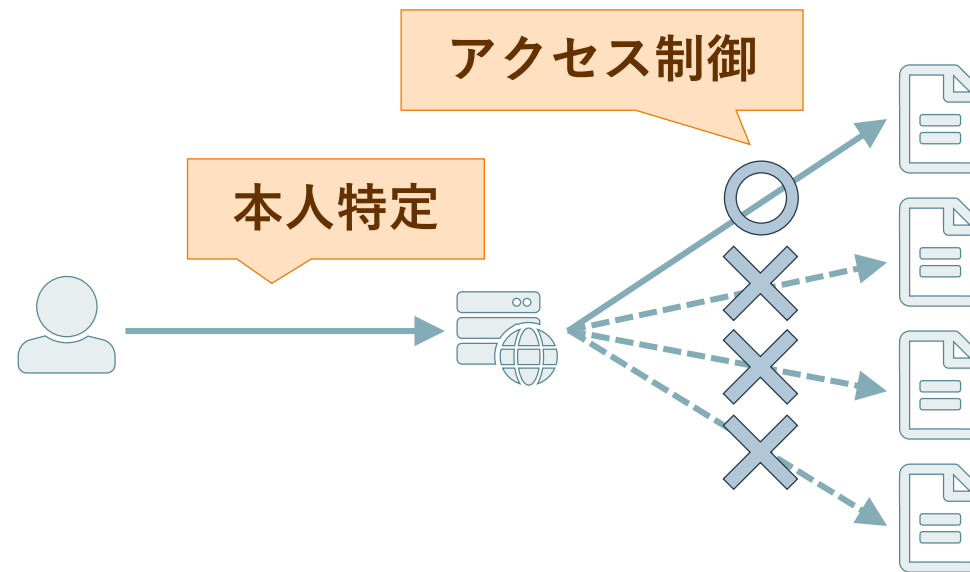
バージョンの付け方

APIは後方互換しなくなったタイミングで付けるのがおススメ
(=メジャーバージョンのみ利用)

- ▶ マイナーやパッチで付けているとその分管理が多くなってしまう
最低限守るのは後方互換しないメジャーの変更

認証と認可

認証は「本人特定」、認可は「アクセス制御」



OAuth と OpenID Connect の違い

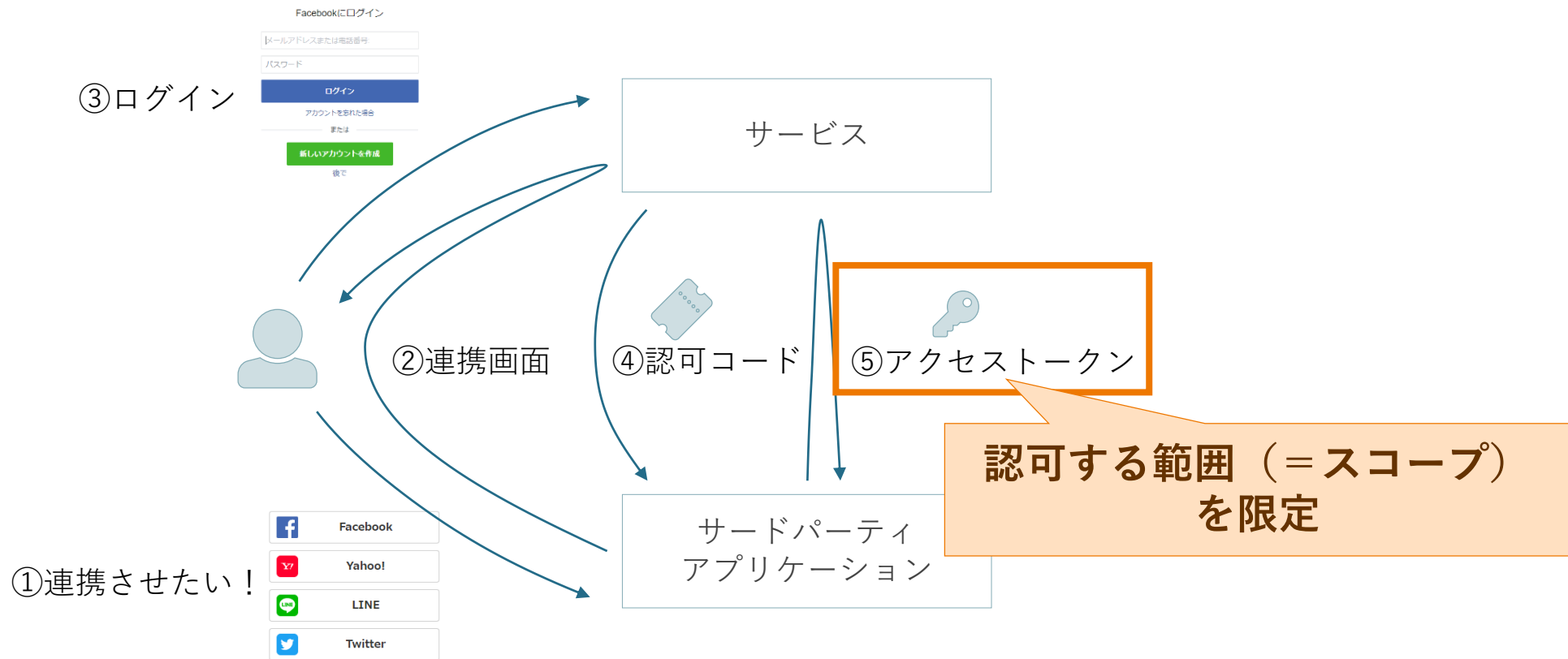
OAuthもOpenID Connectも認可の仕組み

OpenID ConnectはOAuthに本人情報取得を加えた仕組み



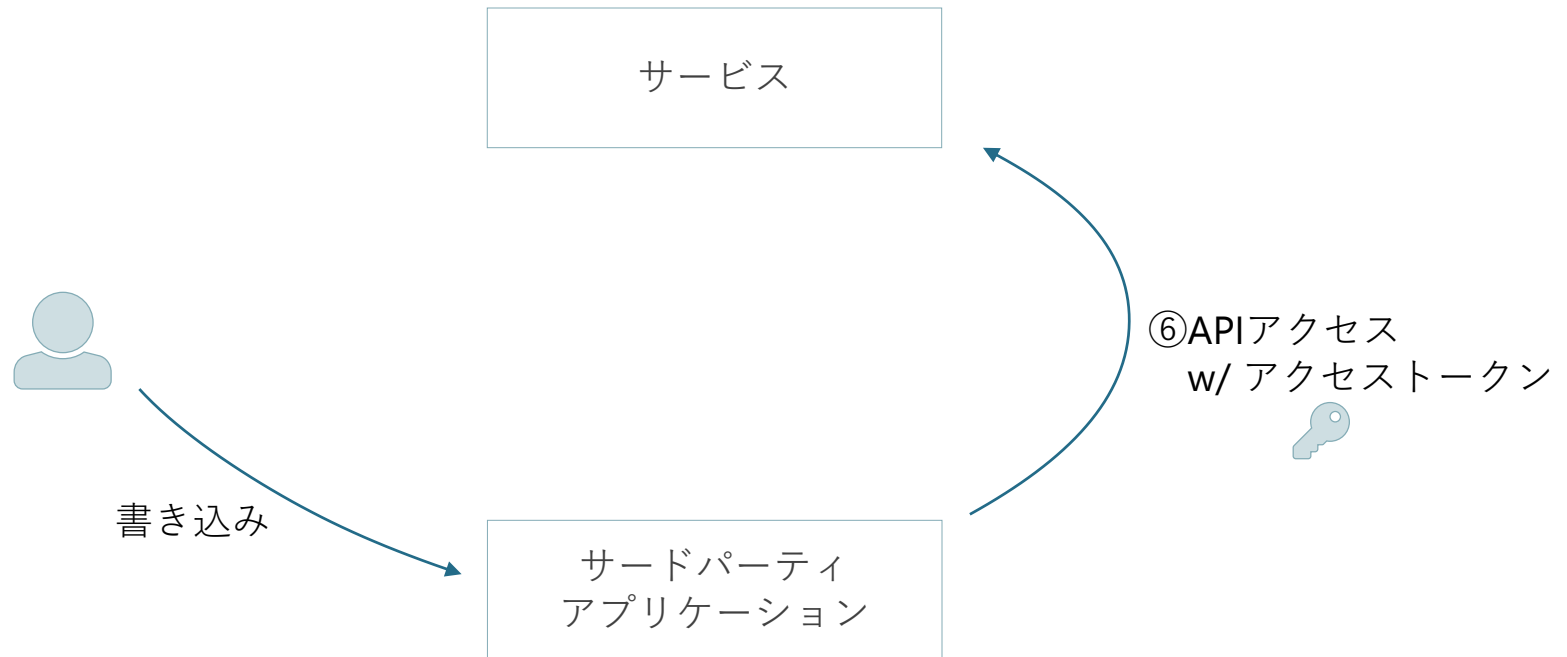
OAuth 概要

Authorization Code フローの場合



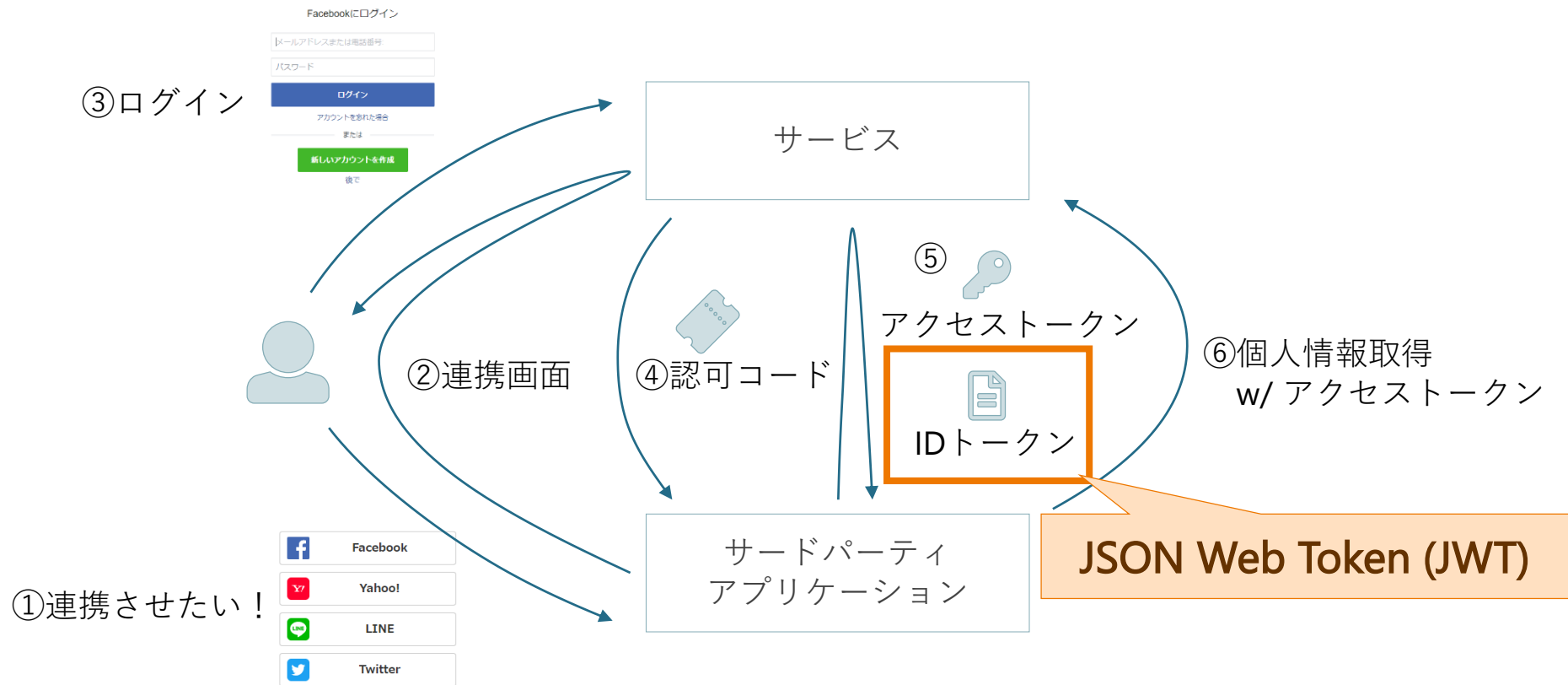
OAuth 概要

Authorization Code フローの場合



OpenID Connect 概要

codeフローの場合



JSON Web Token (JWT) とは

読み方	“ジョット”
仕様	RFC 7519 – JSON Web Token(JWT) で標準化
特徴	<ul style="list-style-type: none">署名による改ざんチェックURL-safeなデータデータの中身はJSON形式
用途	<ul style="list-style-type: none">認証結果をサーバーサイドで保存せずクライアントサイドで保持 (ステートレスな通信の実現)

基本構造

サンプル

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9  
.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ  
.  
Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```



```
base64UrlEncode( ヘッダー )  
+ "." +  
base64UrlEncode( ペイロード )  
+ "." +  
base64UrlEncode( 署名 )
```

ヘッダー

署名で利用するアルゴリズムなどを定義

```
{  
  "typ": "JWT",  
  "alg": "ES256"  
}
```

項目	名称	概要
"typ"	Type	"JWT"固定
"alg"	Algorithm	署名に利用するアルゴリズム
		HS256HMAC using SHA-256
		RS256RSASSA-PKCS1-v1_5 using SHA-256
		ES256ECDSA using P-256 and SHA-256
		none暗号なし

ペイロード

保存したいデータの実態

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

項目	名称	概要
"iss"	Issuer	JWTを発行しているサービス、システムの識別子
"sub"	Subject	同一Issuer内での識別子
"aud"	Audience	JWTを利用しているサービス、システムの識別子
"exp"	Expiration Time	JWTの有効期限
"jti"	JWT ID	JWTの再利用を防ぐために利用する一意識別子

署名

改ざんされていないか確認するための署名

```
ALGORITHM(  
  base64UrlEncode( ヘッダー )  
  + "." +  
  base64UrlEncode( ペイロード ),  
  SECRET  
)
```

項目	概要
ALGORITHM	ヘッダーの "alg" に指定したアルゴリズム
SECRET	アルゴリズムにあわせた鍵（秘密鍵 or 共通鍵）

レートリミットとは

WebアプリをAPI化することで発生する問題は何でしょう？

問題点

API化により、簡単に大量アクセスするプログラムが書ける
意図しない、プログラマの不注意で大量アクセスが発生する

対応策

時間あたりのアクセス制限をかける

レートリミット

レートリミットで考慮すること

観点	設定例
誰に対して	APIキー、ユーザーID...
何に対して	単一機能、機能群、API全体...
制限回数	10回、100回、1000回...
単位時間	10分、1時間、1日...

**制限をリセットするタイミング
Windowとも呼ばれる**

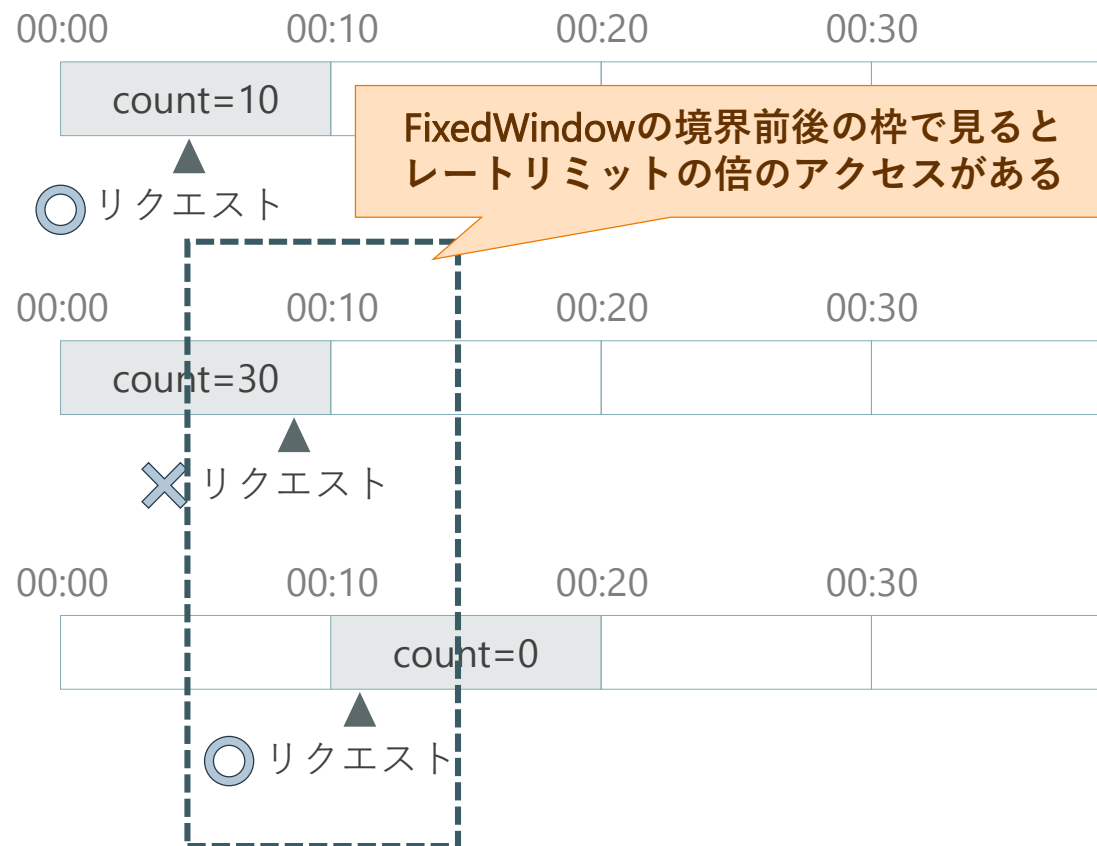
レートリミットアルゴリズム

代表的なレートリミット実現方法は以下の3種類

- Fixed Window
- Sliding Log
- Sliding Window

Fixed Window

30回/10分のレートリミットの場合



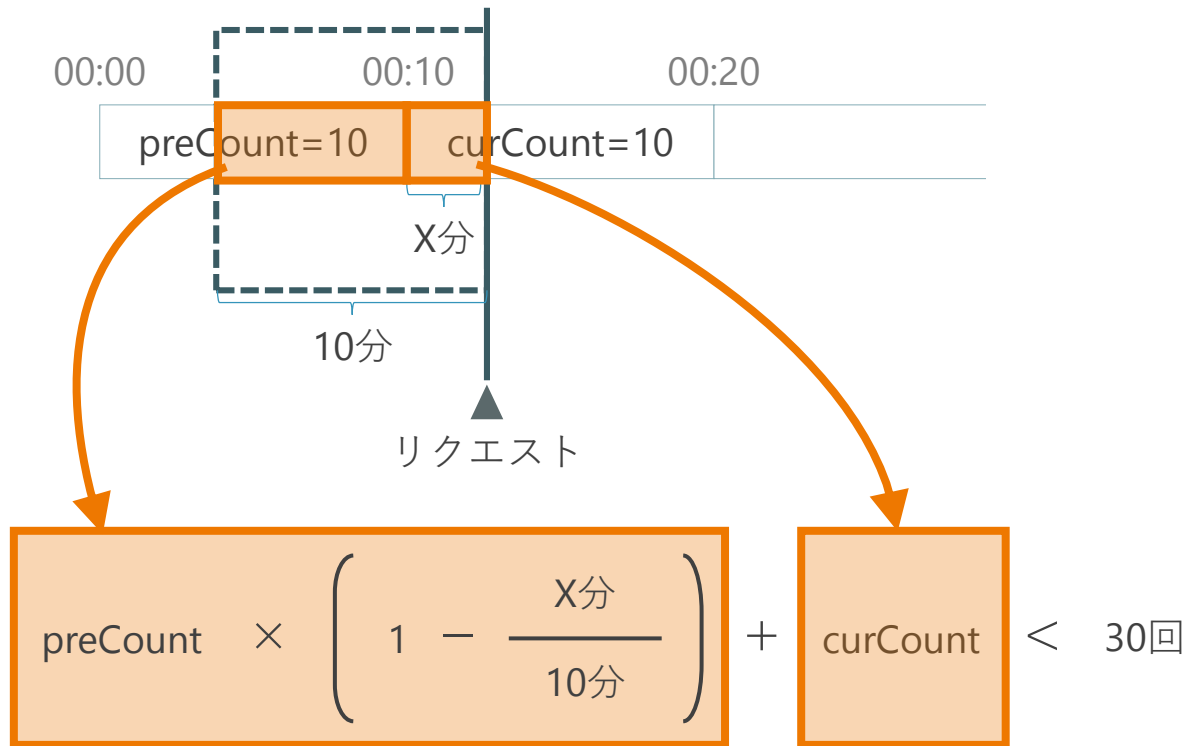
Sliding Log

30回/10分のレートリミットの場合



Sliding Window

30回/10分のレートリミットの場合



アクセス制限の緩和

接続元によっては特別な緩和措置が必要なケースがある

例えば...

- サービス利用が多く、自社にとって優良顧客である場合
- キャンペーンなど、一時的に負荷増大がある場合



アクセス元ごとに一時的に設定変更できる仕組みを考慮する

キャッシュさせる方法

キャッシュ制御に利用するヘッダーは2分類3パターン

分類	ヘッダー
有効期限による制御	Expires
	Cache-Control + Date
検証による制御	Last-Modified + ETag

Expires

Expires: Sun, 03 May 2020 12:30:00 GMT

- キャッシュとしていつまで利用可能かの期限を指定
- 過去日を指定すると「リソースが有効期限切れ」であることを意味する
- Cache-Control が同時指定されている場合、Expiresは無視

Cache-Control + Date

Cache-Control: public, max-age=604800

Date: Sun, 03 May 2020 12:30:00 GMT

- Cache-Control でキャッシュの「可否」「期限」を指定

キャッシュ可否	public	通信経路上のどこでも保存できる
	private	クライアント端末のみ保存できる
	no-cache	保存できるが必ず有効性の確認が必要
	no-store	保存不可
キャッシュ期限	max-age=<秒>	新しいとみなせる時間（秒）

Last-Modified + ETag

```
Last-Modified: Sun, 03 May 2020 12:30:00 GMT  
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

- Last-Modified にリソースの最終更新日時を指定
- ETag に特定バージョンを示す文字列を指定

ETag に指定する文字列例

- コンテンツのハッシュ
- バージョン番号
- 最終更新日時のハッシュ ...など

キャッシュさせる単位

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 212
Cache-Control: public
Content-Language: ja
Date: Sun, 03 May 2020 13:29:56 GMT
Server: api.sample.com
Vary: Content-Language
```

例えば...
応答言語によってキャッシュを分けたい

キャッシュ判断に利用する
ヘッダーを指定

...(省略)...

APIはどこから呼ばれるのか

呼び出し元

- スマホアプリ
- Webページ
 - scriptタグ
 - JavaScript
- 外部システム（バッチ）

悪意あるユーザーが
紛れてくる

APIにもWebサービスと同じセキュリティ対策が必要

代表的な脆弱性対策

今回はWebAPIで代表的な脆弱性対策について学習します

- XSS
- CSRF
- HTTP
- JSON Web Token (JWT)

XSS

脆弱性

悪意あるユーザーが正規のサイトに不正なスクリプトを挿入することで、正規ユーザーの情報を不正に引き出したり操作できてしまう問題

対策

- レスポンスヘッダーの追加

X-XSS-Protection

"1"でXSSフィルタリング有効化。

X-Frame-Options

"DENY"でframeタグ呼び出しを拒否。

X-Content-Type-Options

"nosniff"でIE脆弱性対応。

CSRF

脆弱性

本来拒否しなければならないアクセス元（許可しないアクセス元）からくるリクエストを処理してしまう問題

対策

- 許可しないアクセス元からのリクエスト

X-API-Key (*)

Amazonの場合

システム単位で実行可否判断。

Authentication

ユーザー単位で実行可否判断。

- 攻撃者に推測されにくいトークンの発行/照合処理を実装

X-CSRF-TOKEN (*)

トークンを使って実行可否判断。

SpringSecurityの場合

(*) 独自ヘッダー。標準仕様はないので

HTTP

脆弱性

通信経路が暗号化されないので盗聴されやすい

対策

- 常時HTTPSを利用した通信にする

SSL / TLS / HTTPS の違い

SSL	安全に通信を行うためのプロトコル。2015年に使用禁止。
TLS	安全に通信を行うためのプロトコル。SSLの後継。
HTTPS	HTTP + SSL/TLS。Webで安全に通信するプロトコル。

盗聴、改ざん、なりすましを防ぐ

JSON Web Token (JWT)

脆弱性

クライアント側で内容の確認/編集が簡単にできるため、サーバー側の検証が不十分だと改ざんされた情報を正規として受け入れてしまう

対策

- ヘッダーの alg に"none"以外を指定して署名を暗号化する

```
{  
  "typ": "JWT",  
  "alg": "ES256"  
}
```

- ペイロードの aud に想定する利用者を指定して受信時に検証する

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "aud": "https://api.example.com/"  
}
```

OpenAPI & Swagger基礎

OpenAPI とは

用語の定義

OpenAPI

WSDL や XML と比較されるような**“フォーマット”**を意味する。
このフォーマットを使うと「機械可読なREST API仕様」が記述できる。
JSON または YAML で記述する。

OpenAPI Specification

OpenAPIを記述するための**“書式ルール”**のこと。

Swagger とは

▶ 用語の定義

Swagger

OpenAPI を作成、表示、利用するツール群。



Swagger Editor

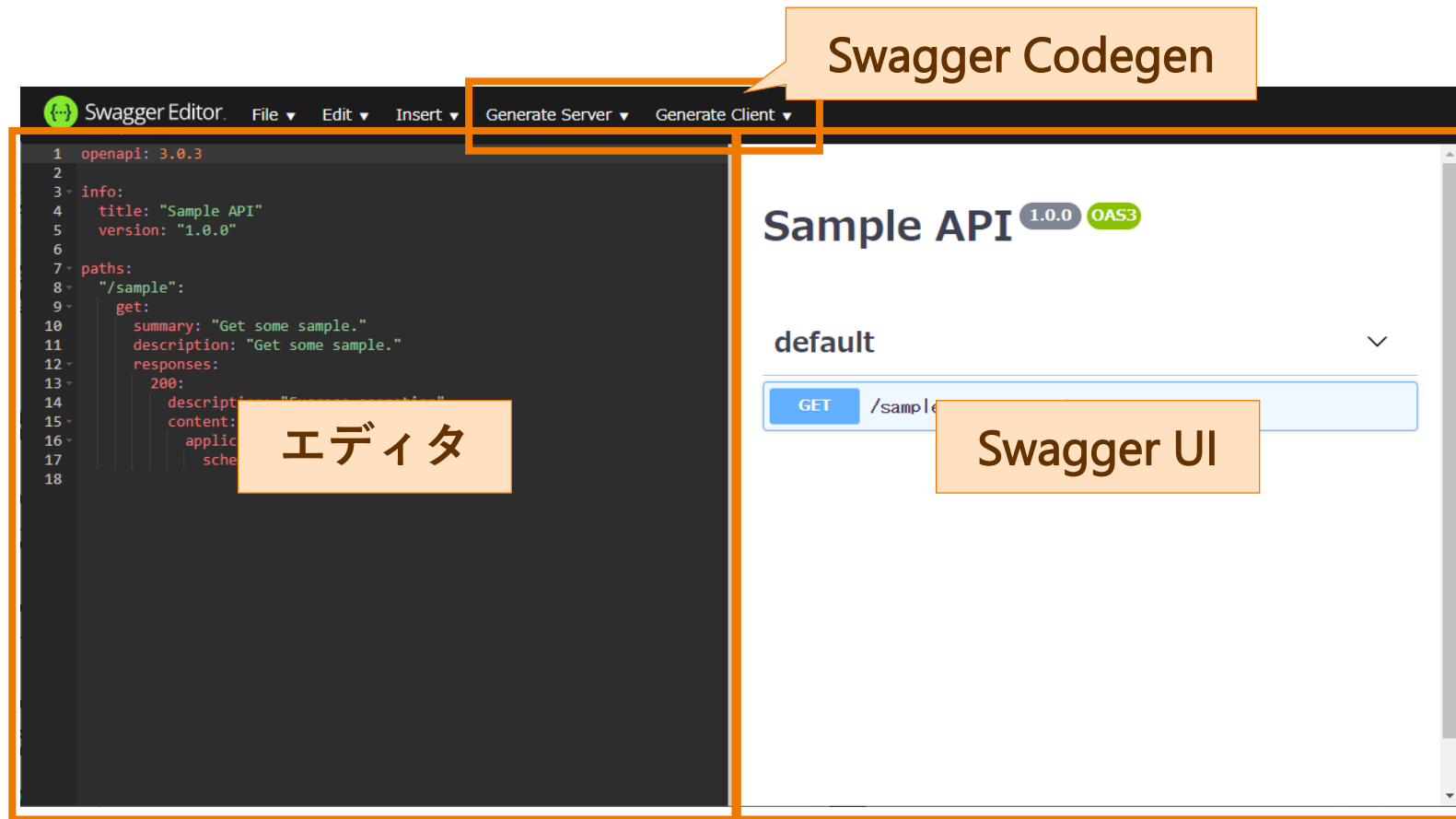


Swagger Codegen



Swagger UI

Swagger Editor の画面構成



Swagger Editor の起動方法

簡単に起動する方法は以下の2種類

- Gitリポジトリのソースコードをクローン
- Dockerイメージを利用

Git リポジトリのソースを使った起動

0. 前提

(特になし)

1. リポジトリのクローン

```
git clone https://github.com/swagger-api/swagger-editor.git
```

2. ブラウザで開く

```
/index.html
```

Docker を使った起動

0. 前提

- Dockerがインストール済みの環境が必要です。
もしなければ Docker Playground でも試せます。

1. コンテナ起動

```
docker run -d -p 80:8080 --name editor swaggerapi/swagger-editor:v3.8.2
```

2. アクセス

```
http://localhost:80
```

基本となるデータ型

▶ 主要な「型（type）」は6種類

type	説明
integer	整数
number	浮動小数
string	文字列
boolean	真偽値
object	オブジェクト
array	配列

▶ Schema オブジェクトに定義する

フォーマット

type	format	説明
integer	int32	符号付き32ビット整数
	int64	符号付き64ビット整数
number	float	浮動小数
	double	倍精度浮動小数
string	—	文字列
	byte	Base64エンコードされた文字列
	binary	バイナリ
	⋮	

フォーマット

type	format	説明
string	date	日付(YYYY-MM-DD形式)の文字列 [RFC3339]
	date-time	日時(YYYY-MM-DDThh:mm:ssTZD形式)の文字列 [RFC3339]
	email	メールアドレスを示す文字列 [RFC5322]
	hostname	ホスト名を示す文字列 [RFC1123]
	ipv4	"." (ピリオド) で区切られた IPv4 アドレス文字列 [RFC2673]
	uri	URI フォーマットに従った文字列 [RFC3986]
	uuid	UUID 文字列 [RFC4122]

Schemaオブジェクト基本

データ型はSchemaオブジェクトで定義する

```
components:
  schemas:
    SampleString:
      type: string
      format: email
```

Schemaオブジェクト

ルートオブジェクト

主要なオブジェクトは 7 種類。必須は 3 種類。

```
openapi: "3.0.3"

info:
  ...
servers:
  ...
tags:
  ...
paths:
  ...
security:
  ...
components:
  ...
```


infoオブジェクト

APIのメタ情報を定義する

```
info:  
  title: "Sample API"  
  description: |  
    # Features  
    - Get users.  
    - Create user.  
  termsOfService: "https://sample.com/terms/"  
  contact:  
    name: "Customer Support"  
    url: "https://sample.com/support/"  
    email: "support@sample.com"  
  license:  
    name: "MIT License"  
    url: "https://opensource.org/licenses/MIT"  
  version: "1.0.0"
```

必須
APIのタイトルを指定する。

必須
APIドキュメントのバージョン情報

Serversオブジェクト

接続可能なサーバーを定義する

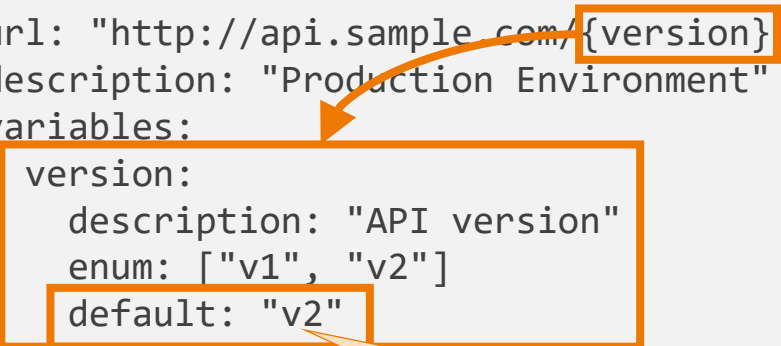
```
servers:  
  - url: "http://api.sample.com/{version}"  
    description: "Production Environment"  
    variables:  
      version:  
        description: "API version"  
        enum: ["v1", "v2"]  
        default: "v2"
```

必須
接続先URLを指定します

Serversオブジェクト

接続可能なサーバーを定義する

```
servers:  
  - url: "http://api.sample.com/{version}"  
    description: "Production Environment"  
    variables:  
      version:  
        description: "API version"  
        enum: ["v1", "v2"]  
        default: "v2"
```



デフォルト値は必須

パス全体像

```
paths:
  "/users/{userId}/message":
    post:
      summary: "Send new message."
      description: "Send new message."
      tags: ["users"]
      deprecated: false
      parameters:
        - name: "userId"
          in: "path"
          required: true
          schema: { type: string }
      requestBody:
        content:
          application/json: {}
      responses:
        "201":
          description: "Success response"
      security:
        - sample_oauth2_auth: ["create_review"]
```

メタデータ

リクエストパラメーター

リクエストボディー

レスポンス

セキュリティ

リクエストパラメーター

```
paths:
  "/users/{userId}/message":
    post:
      summary: "Send new message."
      parameters:
        - name: "userId"
          in: "path"
          description: "User identifier"
          required: true
          schema: { type: string }
          example: "m4v5bjhq"
      responses:
        "201":
          description: "Success response"
```

リクエストパラメーター

```
paths:
  "/users/{userId}/message":
    post:
      summary: "Send new message."
      parameters:
        - name: "userId"
          in: "path"
          description: "User identifier"
          required: true
          schema: { type: string }
          example: "m4v5bjhq"
      responses:
        "201":
          description: "Success response"
```

パラメーターの場所

値	説明
query	クエリパラメーター
header	リクエストヘッダー
path	パスパラメーター
cookie	クッキー

リクエストパラメーター

```
paths:
  "/users/{userId}/message":
    post:
      summary: "Send new message."
      parameters:
        - name: "userId"
          in: "path"
          description: "User identifier"
          required: true
          schema: { type: string }
          example: "m4v5bjhq"
      responses:
        "201":
          description: "Success response"
```

必須パラメーターかどうか
デフォルト false
in: path の場合だけ必ず true を指定する

リクエストボディ

```
paths:
  "/users/{userId}/message":
    post:
      summary: "Send new message."
      parameters:
        - name: "userId"
          in: "path"
          description: "User identifier"
          required: true
          schema: { type: string }
      requestBody:
        description: "Message body"
        required: true
        content:
          application/json:
            schema: { type: string }
            example: "Hello World"
      responses:
        "201":
          description: "Success response"
```

リクエストボディ

レスポンス

```
paths:
  "/users/{userId}/message":
    put:
      summary: "Modify message"
      parameters:
        - name: userId
          in: "path"
          description: "User identifier"
          required: true
          schema: { type: string }
      responses:
        "200":
          description: "Success operation"
          headers:
            x-rate-limit-remaining:
              description: "Number of remaining requests"
              schema: { type: integer }
          content:
            application/json:
              schema:
                type: object
                properties:
                  score: { type: integer }
                  comment: { type: string }
                  created: { type: string, format: date-time }
```

レスポンス

レスポンス

```
paths:
  "/users/{userId}/message":
    put:
      summary: "Modify message"
      parameters:
        - name: user
          in: "path"
          description: "user id"
          required: true
          schema: { type: "string" }
      responses:
        "200":
          description: "message modified"
          headers:
            x-rate-limit-remaining:
              description: "rate limit remaining"
              schema: { type: "integer" }
          content:
            application/json:
              schema:
                type: object
                properties:
                  score: { type: integer }
                  comment: { type: string }
                  created: { type: string, format: date-time }
```

ステータスコードごとに定義

JSON⇔YAML互換性のため"(ダブルクォート)で囲む

4XX, 5XX のように"X"を使ってまとめることも可能

成功ステータスは最低限入れておくのが推奨

レスポンス

```
paths:
  "/users/{userId}/message":
    put:
      summary: "Modify message"
      parameters:
        - name: userId
          in: "path"
          description: "User identifier"
          required: true
          schema: { type: string }
      responses:
        "200":
          description: "Success operation"
          headers:
            x-rate-limit-remaining:
              description: "Number of remaining requests"
              schema: { type: integer }
          content:
            application/json:
              schema:
                type: object
                properties:
                  score: { type: integer }
                  comment: { type: string }
                  created: { type: string, format: date-time }
```

レスポンスの説明

レスポンスヘッダー

レスポンスボディ

Schemaオブジェクト表現

共通プロパティ

integer, number

string

boolean

object

array

enum

Schemaオブジェクト共通プロパティ

```
components:
  schemas:
    SampleString:
      type: string
      format: email
      description: "str" # 共通：データ型の説明
      default: "hoge" # 共通：デフォルト値
      nullable: true # 共通：null許容するかどうか
      example: "abc" # 共通：サンプル
      deprecated: false # 共通：廃止かどうか
```

integer, number

```
components:
  schemas:
    SampleInt:
      type: integer
      format: int32
      multipleOf: 10          # 指定された数の倍数になっているかどうか
      maximum: 100           # 最大値
      exclusiveMaximum: false # 最大値を含まないかどうか
                             # true: x<100, false: x<=100
      minimum: 0             # 最小値
      exclusiveMinimum: false # 最小値を含まないかどうか
                             # true  0<x, false: 0<=x
```

string

```
components:
  schemas:
    SampleString:
      type: string
      format: email
      minLength: 0
      maxLength: 100
```

最小文字数。指定された文字以上であること。
最大文字数。指定された文字数以下であること。

boolean

```
components:  
  schemas:  
    SampleBoolean:  
      type: boolean
```

真偽型で追加定義できる
特別な項目はない

object

```
components:
  schemas:
    SampleObject:
      type: object
      properties:                # プロパティ定義
        name: { type: string }
        dob: { type: string, format: date }
      additionalProperties: true # スキーマ以外のプロパティを許すかどうか
      required:                # 必須プロパティの定義
        - name
      minProperties: 2          # 最小プロパティ数
      maxProperties: 2          # 最大プロパティ数
```

array

```
components:
  schemas:
    SampleArray:
      type: array
      items: { type: string } # 配列内に入れられるスキーマを指定
      minItems: 0             # 最小個数
      maxItems: 5             # 最大個数
      uniqueItems: true       # 配列内で値の重複を許すかどうか
```

enum

```
components:
  schemas:
    SampleEnum:
      type: string
      enum: ["red", "blue", "yellow"] # 選択可能な値を設定
```

タグ

タグはAPIを分類する仕組み

```
...
tags:
- name: "users"
  description: "User operation"

paths:
  "/users":
    get:
      summary: "Get user list"
      description: "Get user list"
      tags: ["users"]
      deprecated: false
      responses:
        "200":
          description: "Success operation"
          content: {}
...

```

pathsの中で利用

複数指定した場合
複数個所にAPIが表示される

コンポーネント化できる要素

パス中で利用する 5 要素 + セキュリティ 1 要素

```
components:
```

```
  schemas:
```

```
    ...
```

```
  parameters:
```

```
    ...
```

```
  requestBodies:
```

```
    ...
```

```
  responses:
```

```
    ...
```

```
  headers:
```

```
    ...
```

```
  securitySchemes:
```

```
    ...
```

パス中で利用する 5 要素

セキュリティ

コンポーネントの定義

分類ごとに名前を付けて定義する

定義するコンポーネントの分類

- schemas
- parameters
- requestBodies
- responses
- headers

```
schema: { type: string }  
responses:  
  "200":  
    description: "Success operation"  
    content:  
      application/json:  
        schema:  
          $ref: "#/components/schemas/Book"
```

components:

schemas:

```
Book:  
  type: object  
  properties:  
    title: { type: string }  
    author: { type: string }  
    published: { type: string }
```

コンポーネントの定義

分類ごとに名前を付けて定義する

```
paths:
```

コンポーネント名
利用可能文字： a~z A~Z 0~9 . _ -

```
  name: id
  in: path
  required: true
  schema: { type: string }
  responses:
    "200":
      description: "Success operation"
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/Book"
```

```
components:
```

```
  schemas:
```

Book:

```
    type: object
    properties:
      title: { type: string }
      author: { type: string }
      published: { type: string }
```

コンポーネントの定義

分類ごとに名前を付けて定義する

```
paths:
  "/books/{id}":
    get:
      summary:
      parameter
      - name: id
        in: path
        required: true
        schema: { type: string }
      responses:
        "200":
          description: "Success operation"
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Book"
```

コンポーネントの定義

```
components:
  schemas:
    Book:
      type: object
      properties:
        title: { type: string }
        author: { type: string }
        published: { type: string }
```


コンポーネントの利用

URLのように定義する

```
paths:
  "/books/{id}":
    get:
      summary: "Get specified book"
      parameters:
        - name: id
          in: path
          required: true
          schema: { type: string }
      responses:
        "200":
          description:
            content:
              application/json:
                schema:
                  $ref: "#/components/schemas/Book"
```

参照先の指定

`[{FILE_PATH}]#/components/{TYPE}/{NAME}`

`"#/components/schemas/Book"`

セキュリティ定義概要

利用するスキームの定義

```
components:
  securitySchemes:
    sample_jwt_auth:
      type: http
      description: "JWT Auth."
      scheme: bearer
      bearerFormat: JWT
```

利用するスキームは components に定義

APIに適用

```
paths:
  "/samples":
    get:
      summary: "Get all sample data"
      responses:
        "200":
          description: "Success"
      security:
        - sample_jwt_auth: []
```

APIへ適用する方法は2種類

- ・ 個別に適用
- ・ 全体に適用して個別に解除

利用するスキームの定義

Basic認証

```
components:
  securitySchemes:
    sample_basic_auth:
      description: "Basic authentication"
      type: http
      scheme: basic
```

利用するスキームの定義

JWT

```
components:
  securitySchemes:
    sample_jwt_auth:
      description: "JWT authentication"
      type: http
      scheme: bearer
      bearerFormat: JWT
```

利用するスキームの定義

API Key

```
components:
  securitySchemes:
    sample_apikey_auth:
      description: "API-Key authentication."
      type: apiKey
      in: header
      name: X-API-Key
```

利用するスキームの定義

ログインセッション

```
components:
  securitySchemes:
    sample_cookie_auth:
      description: "Login Session authentication."
      type: apiKey
      in: cookie
      name: JSESSIONID
```

利用するスキームの定義

OAuth 2.0

```
components:
  securitySchemes:
    sample_oauth2_auth:
      description: "OAuth2"
      type: oauth2
      flows:
        authorizationCode:
          authorizationUrl: "https://oauth.sample.com/auth"
          tokenUrl: "https://oauth.sample.com/token"
          scopes:
            "create_review": "Post new review."
```

適用方法 1 : 個別APIに適用

```
paths:
  "/samples":
    get:
      summary: "Get all sample data"
```

scopeがない場合（Basic認証, JWT など）は空配列
scopeが必要な場合（OAuth, OpenID Connect）は必要なスコープ名

```
security:
  - sample_jwt_auth: []
components:
  securitySchemes:
    sample_jwt_auth:
      type: http
      description: "JWT Auth."
      scheme: bearer
      bearerFormat: JWT
```

指定するものは
securitySchemesに定義したものと
同じ名称を指定する

適用方法 2 : 全体に適用して個別に解除

```
paths:
  "/samples":
    get:
      summary: "Get all sample data"
      responses:
        "200":
          description:
```

解除したいAPIには空配列を指定

```
      security: []
```

```
security:
- sample_jwt_auth: []
```

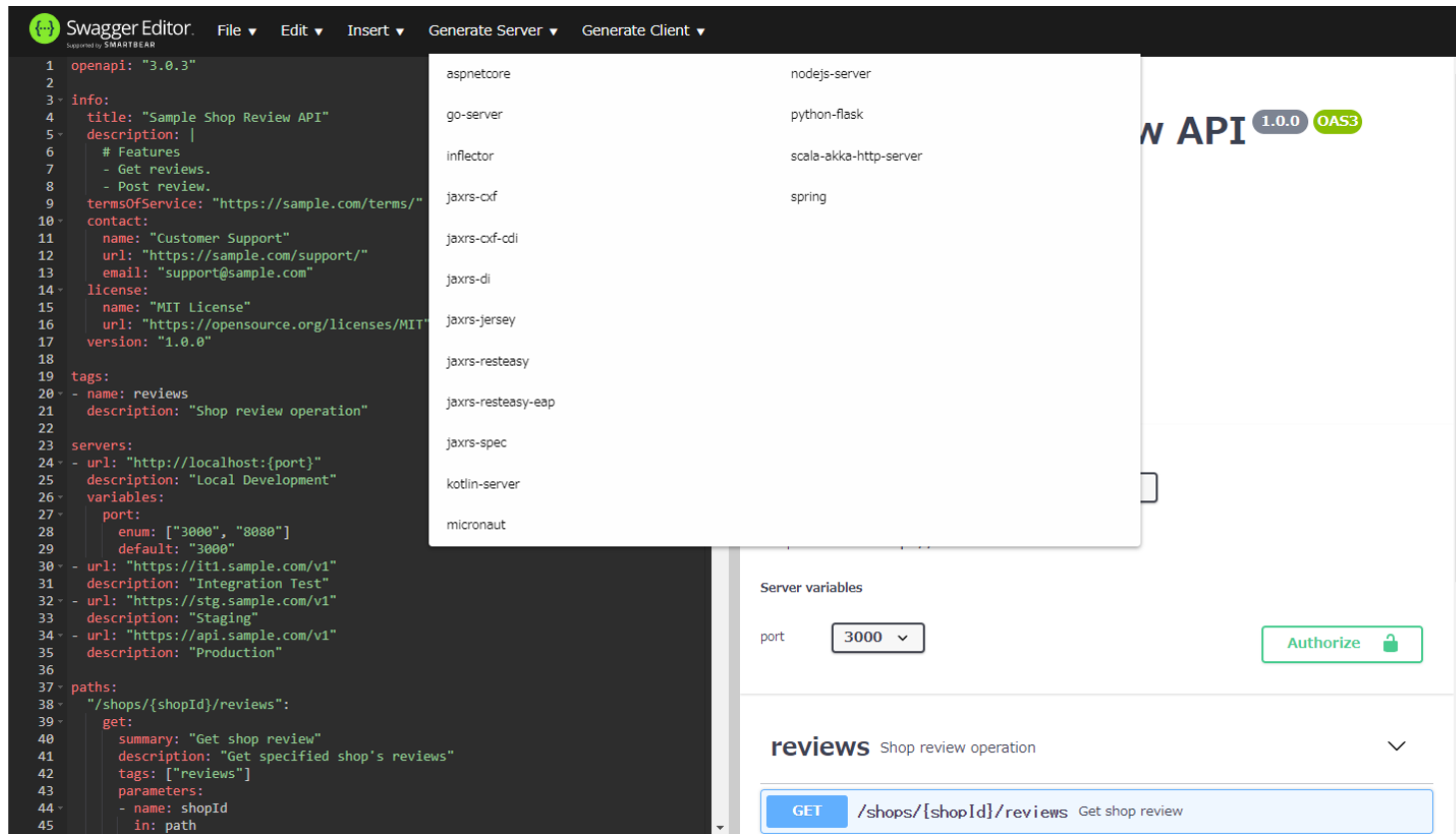
ルートドキュメントの security に
必要な認証を指定

```
components:
  securitySchemes:
    sample_jwt_auth:
      type: http
      description: "Sample
      scheme: bearer
      bearerFormat: JWT
```

指定するものは
securitySchemesに定義したものと
同じ名称を指定する

スタブコードの生成

Swagger Editor 上からダウンロード



スタブの実行

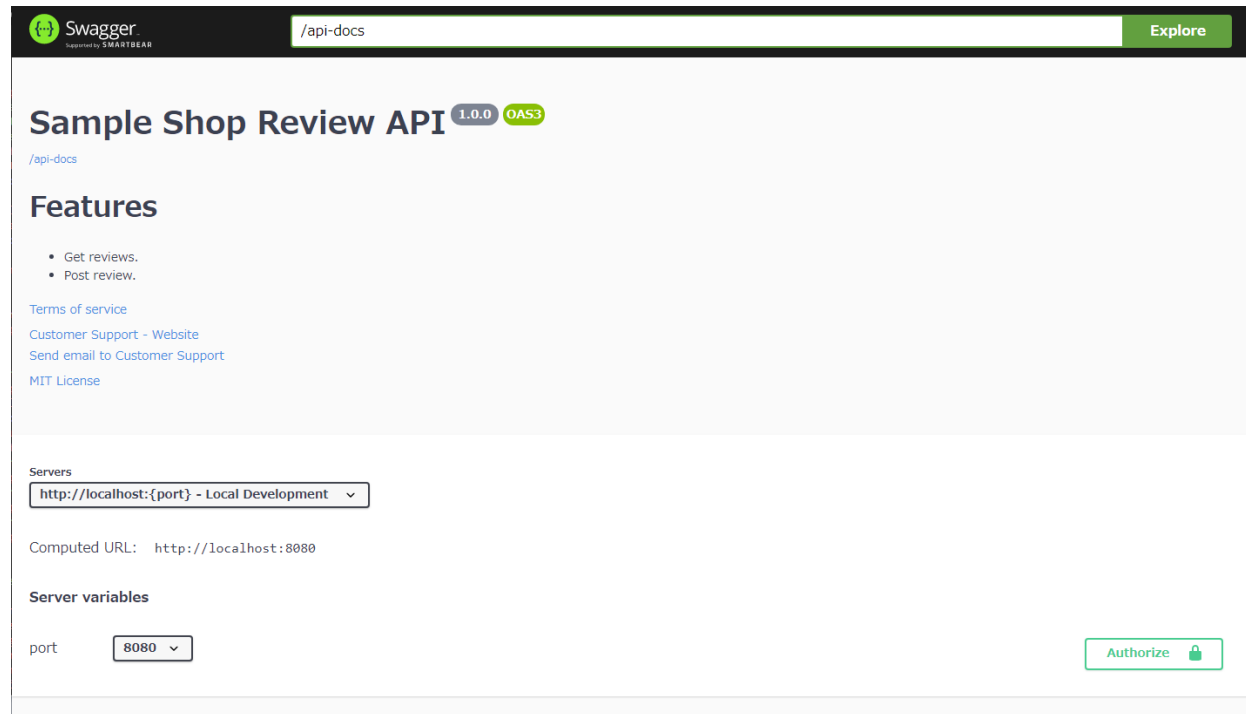
Node.js の場合、解凍したフォルダで以下のコマンドを実行

```
npm start
```

スタブの動作確認

ブラウザで以下のアドレスへアクセス

`http://localhost:8080/docs`



スタブの動作確認

ブラウザで以下のアドレスへアクセス

`http://localhost:8080/docs`

Servers

`http://localhost:{port} - Local Development`

Computed URL: `http://localhost:8080`

Server variables

port `8080`

Authorize

reviews Shop review operation

GET `/shops/{shopId}/reviews` Get shop review

Get specified shop's reviews

Parameters

Name	Description
shopId * required string (path)	<input type="text" value="hoge"/>
order string (query)	Sort order Available values : asc, desc

Try it out

Try it out から実行

おわりに

これから

本講座で触れられなかったこと

“

HTTPなどのインターネット関連技術を利用してプログラム
が読み書きしやすい形でメッセージ送受信を行えるよう定義
した規約，または規約を実装して展開されるサービス。

規約の作り方

サービスの実装

講座紹介



Packerで実現するInfrastructure as Code

手作業でAWSマシンイメージを作成することに限界を感じた方向け、IaCの実践講座。
コードでAWSマシンイメージを作成するだけでなく、マシンイメージのテストを行う方法についても学べます。



AWSで作るWebアプリケーション実践講座

はじめてAWSを触る方へ向けた本格的Webアプリケーション構築講座。
アカウント作成から各種設定に始まり、順にアプリケーションの構築を学べます。



Docker + Kubernetes で構築する Webアプリケーション 実践講座

DockerおよびKubernetesを使ったWebアプリケーション構築の実践講座。
Docker、Kubernetesを学習するための学習環境作成から基礎、応用を学べます。



REST WebAPI サービス 設計

今までありそうでなかった「REST WebAPI 設計」の"いろは"が学べる講座。
なんとなく作っていたAPIに対して、あるべき姿がどのようなものかを学べます。



MongoDB 入門 一演習しながら学ぶクエリ操作一

主にアプリ開発を行っている方でNoSQLに興味がある方に向け、MongoDBを使ったクエリ実行を学べる講座。
MongoDBの学習環境構築からサンプルデータを使って実際にクエリの組み立て、実行を行いながら学習できます。



Node.js + Express で作る Webアプリケーション 実践講座

本格的Webアプリ開発をやったことのない方向け、Node.js + MongoDB を使ったWebアプリケーション開発の実践講座。
実開発で必要となるツール類の基本、開発における基本を学び、最終的には実開発を意識したWebアプリケーション構築ができます。



Node.js 入門 一演習しながら学ぶ基本クラスの使い方

JavaScriptを学んだ方向け、Node.js の基本が学べる講座。
Node.jsを使った開発をするうえで基礎となるコアモジュールの仕組みや使い方をサンプルコードを実装しながら学べます。

End of Page