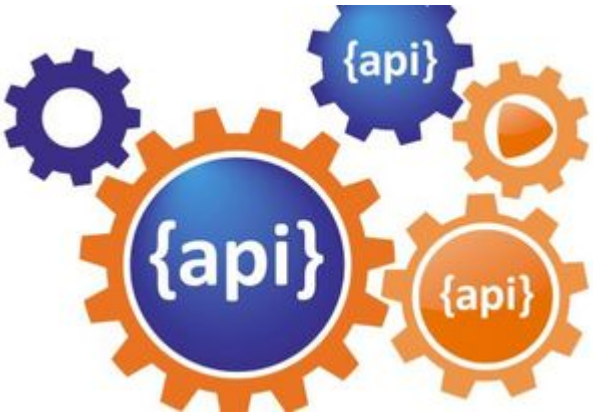
Zalando RESTful API と イベントスキーマのガイドライン

Zalando SE (翻訳: kawasima) Original: https://opensource.zalando.com/restful-api-guidelines/



別の形式: PDF, EPUB3

Table of Contents

1. はじめに

Zalandoのソフトウェアアーキテクチャは、疎結合なマイクロサービスを中心としており、 それらはJSONペイロードをもつRESTful API群によって、機能が提供されています。 小さなエンジニアのチームは、自分たちでAWSアカウントにこれらのマイクロサービスを デプロイしたり運用したりしています。 私たちのAPIは、その多くが私たちのシステムが何をするのかを完全に表現しており、 それゆえに貴重なビジネス資産となっています。 Zalandoがとあるオンラインショップから価値あるファッションプラットフォームへと変貌を とげるために、私たちは新しいオープンプラットフォーム戦略の展開をはじめました。 なので、高品質で長持ちするAPIの設計は、私たちにとってよりクリティカルなものになってきているのです。 私たちのビジネスパートナーがサードパーティのアプリケーションから使える公開APIをたくさん 開発することは、私たちの戦略の肝なのです。

私たちは"APIファースト"を、主要なエンジニアリングの原則の1つとして採用しています。 マイクロサービス開発はコードの外にAPIを定義することから始まり、ピアレビューのフィードバックを 十分に取り込みつつ、高品質のAPIへと発展させていきます。 APIファーストは品質に直結する標準を網羅しつつ、軽量なレビュー手順を含んだピアレビューの文化を 育んでいきます。

- APIは理解しやすく習得が簡単である。
- APIは特定の実装やユースケースから汎用的であり抽象化されている。
- APIは堅牢で使うのが簡単である。
- APIは共通の見た目と操作性をもっている。
- APIは一貫したRESTfulなスタイルと文法にしたがう。
- APIは他のチームのAPI群や私たちのグローバルなアーキテクチャとも一貫性をもつ。

ガイドラインで使われる規約

本文書では要求レベルのキーワードとして、 "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", "OPTIONAL" が 使われます。 これは RFC 2119の解釈と同様です。

Zalando固有の情報

私たちの「RESTful API ガイドライン」の目的は、「一貫したAPIのルックアンドフィール」の品質水準を定めることにあります。チームはAPIの開発においては、このガイドラインを守っていく責務がありますし、またプルリクエストを投げてガイドラインを進化させていくことが求められます。

これらのガイドラインは私たちの仕事が発展していく限り、ずっと「作成中」のステータス のままでしょう。しかしチームはこれらにしたがい信頼することが、きっと可能なはずです。

変化を続けるガイドラインに対して、次のルールを適用します。

- 既存のAPIは変更する必要がなくても、私たちはそれを推し進めます
- 既存のAPIのクライアントは、古くなったルールにしたがったAPIに対処しなければならない
- 新しいAPIは現在のガイドラインを尊重しなければならない

さらには、いったんAPIが外部に公開されたら、全体の一貫性の維持のために、 最新のガイドラインにしたがって、再レビューし変化させていかなければ ならないことを、肝に銘じておかなければなりません。

2. 原則

API設計の原則

SOAPインタフェースをもつSOA WebサービスとRESTを比較すると、前者はユースケースに特化した 操作を中心に据える傾向があります。一方RESTはビジネス(データ)のエンティティを、URIで識別されるリソースとして どう見せるか、標準化されたCRUDのようなメソッドで、異なる表現やハイパーメディアを使ってどう操作されうるかを中心に据えます。 RESTful APIは、固有のユースケースからは切り離され、クライアント/サーバの結合を疎にする傾向にあります。 さまざまな新しい業務サービスを構築するために、APIのプラットフォームを提供している(コア)サービスのエコシステムに、 より適したものとなっていきます。 私たちは、その機能がインターネットで提供されるのか、イントラネットで提供されるのかに かかわらず、すべての種類のアプリケーション、(マイクロ)サービスコンポーネントに、RESTfulウェブサービスの原則を適用します。

- 私たちはJSONペイロードをもったRESTベースのAPIを好む
- 私たちはシステムが、真にRESTfulになることを好む [1]

API設計と利用の重要な原則がポステルの原則です。 ロバストネスの原則 としても知られています。(RFC 1122 も参照ください)

- 送信するものに関しては厳密に、受信するものに関しては寛容に

読みましょう: RESTful APIの設計スタイルとサービスアーキテクチャについての読み物がいくつかあります。

- Book: Irresistable APIs: Designing web APIs that developers will love
- Book: REST in Practice: Hypermedia and Systems Architecture
- Book: Build APIs You Won't Hate
- InfoQ eBook: Web APIs: From Start to Finish
- Lessons-learned blog: Thoughts on RESTful API Design
- Fielding Dissertation: Architectural Styles and the Design of Network-Based Software Architectures

製品としてのAPI

前述のとおり、Zalandoは単なるオンラインショップから、価値あるファッションプラットフォームへと 転換しており、ビジネスパートナーのためにSaaP(プラットフォームとしてのソフトウェア)モデル に続く、製品の豊富な集合を提供しています。 企業として私たちは、これらの製品を(内外問わず)顧客に届けたいのです。

プラットフォームとしての製品とは、(公開)APIで提供される機能群です。 したがってAPIの設計は、製品としての原則に基づくべきなのです。

- APIを製品として扱い、製品のオーナーのように振る舞う
- 顧客の立場に身を置き、彼らのニーズの支持者となる
- 顧客エンジニアにとってAPIがとても魅力的なものになるために、APIのシンプルさ、理解しやすさ、使いやすさを際立たせる
- 長期にわたって、APIをアクティブに改善し、一貫性を維持していく
- 顧客のフィードバックを受け付け、サービスと同レベルのサポートを提供する。

「製品としてのAPI」を掲げることで、サービスのエコシステムはより進化がしやすくなり、 コア機能を組み合わせることによって、新しいビジネスアイデアを早く試すことができるようになります。

APIプラットフォーム上に構築されたイノベーティブな製品サービスと、 オマケとしてAPIが提供されている通常の企業情報システムとでは、 システム統合をサポートしたり、ローカルに最適化されたサーバサイドを実現するところで違いが出てきます。

顧客の具体的なユースケースを理解し、製品として考えAPIの設計のズレとのトレードオフをチェックしましょう。 クライアント側に不要な負担を強いるような、実装の早すぎる最適化を避け、APIの品質とクライアント開発者の使い勝手に最大限の関心をもちましょう。

製品としてのAPIは、(次章で示す)APIファーストの原則と非常に近い関係にあります。 APIファーストは高品質なAPIを、どうエンジニアリングしていくかによりフォーカスしたものです。

APIファースト

APIファーストは、私たちの エンジニアリングとアーキテクチャ原則 の1つです。

手短にいうと、APIファーストは2つの観点を要求します。

- 実装する前に標準仕様言語を使って、まずAPIを定義する
- 同僚やクライアント開発者から、いち早くレビューのフィードバックを得る

コードの外にAPIを定義することで、私たちは早いレビューフィードバックを促し、 サービスインタフェースの設計が、以下に焦点を当てれるような開発の規律にしたいのです。

- ドメインと要求機能の深い理解
- 一般化された業務エンティティ / リソース (つまり、あるユースケース固有のAPIを避ける)

- WHATとHOWの関心の分離。すなわち、実装の観点から「抽象」をを切り離す。APIはたとえ技術スタックが完全に置き換わったとしても、安定しているべきである。

さらには、標準化された仕様のフォーマットによるAPI定義は、また、

- API仕様の拠り所。それはサービスプロバイダとクライアントのユーザの間の重要な契約の1つとなります。
- APIディスカバリ、API GUI、APIドキュメント、API品質の自動チェックなどのAPI基盤ツール。

APIファーストの要素はまた、このAPIガイドラインと同僚やクライアント開発者から、 早期にレビューフィードバックを得るための、APIガイドラインと標準化されたAPIレビュープロセスです。 APIの品質を高め、アーキテクチャと設計の調整を可能にし、サービスプロバイダの開発サイクルに クライアントアプリケーションの開発が引きずられないようにするために、私たちにとってピアレビューは重要です。

APIファーストが、私たちの愛する **アジャイル開発の原則と反するものではない** ことは重要なことです。サービスアプリケーションは徐々に進化していくように、APIもまた進化していきます。 もちろんAPI仕様は異なるサイクルで、繰り返し進化を遂げていくでしょうし、そうあるべきですが、 それらはみな、ドラフトの状態と、 *早期の* チームレビューおよびピアレビューフィードバックから始まるのです。 APIが変更されると、実装上の懸念と自動化テストのフィードバックからメリットを得ることもあります。 開発のライフサイクルにおいて、生産的な機能がまだない間も、クライアントと変更の調整を続ける限り 破壊的変更を伴いながらAPIを進化させていくことができます。

したがってAPIファーストは、要求とドメインについて100パーセント理解し、完全なAPIを 定義し、ピアレビューでその確信が得られるまで、コードを書いてはいけない、ということを 意味するもの *ではありません* 。一方、APIファーストがサービス統合または本番運用開始の後に、API定義の公開したりピアレビューしたりする バッドプラクティスと衝突があることも明らかです。

早く、できるだけ早くフィートバックを得ることは重要ですが、その前にAPI変更が次の進化のステップへ の足がかりであることを認識し、既にチーム内レビューで確立された(APIガイドライン遵守を含む)一定水準の品質を、 保つこともまた重要です。

3. 全般にわたるガイドライン

タイトルには関連するラベルがついています。: **MUST**, **SHOULD**, **MAY**

MUST APIファーストの原則にしたがう

あなたはAPIファーストの原則にしたがわなければなりません。

- 実装を始める前に、仕様記述言語としてOpenAPIを使って、まずAPI定義を書かなければならない。
- このガイドラインに沿って一貫性のあるAPIを設計しなければならない。
- 同僚やクライアント開発者からのレビューフィードバックを早めに受け取るようにしなければならない。

MUST OpenAPIを使ってAPIの仕様を提供する

私たちは Open API (以前はSwaggerと呼ばれていたもの)を、API定義の標準として使っています。APIの設計者はAPI仕様ファイルを、可読性向上のため **YAML** を使って書きます。**Open API 3.0** バージョンを使うことを推奨しますが、**Open API 2.0** (別名 Swagger 2)もまだサポートした方がよいでしょう。

API仕様はソースコード管理システムを使って、バージョン管理するべきです。一番良いのはAPIの実装コードを同じやり方にしておくことです。

API実装のデプロイと同じタイミングで、API仕様もデプロイするようにします。そうすることでAPIポータルから探せるようになります。

ヒント: Open API 3.0/2.0 を調査するには、 Open API specification mind map を使って探ってみたり、私たちの Swagger Plugin for IntelliJ IDEA を使って最初のAPIを作ったりすることがよい方法です。既存のAPIを調査したり、検証/評価するには、 Swagger Editor が良い出発点になるでしょう。

ヒント: 私たちは GraphQL のガイドラインはまだ提供していません。私たちの技術評価軸であるZalando Tech Radarでは、汎用目的のピアトゥピアなマイクロサービス間のやり取りには、リソース指向のHTTP/REST APIのスタイル(と関連するツールやインフラサポート)に注力した方が良いと評価しています。RESTと比較してGraphQLには大きなメリットが無いが、いくつかの欠点があると私たちは考えます。ですが、GraphQLは特定ターゲットドメインの問題、特にフロントエンド(BFF)およびモバイルクライアントのバックエンドに多くの価値を提供できます。私たちは既にDXインタフェースフレームワークのAPIテクノロジーとしてGraphQLを既に利用しています。

MUST 永続的で不変であるリモート参照のみを使う

通常、API仕様のファイルは自己記述的です。つまりファイルは `../fragment.yaml#/element` や `$ref: 'https://github.com/zalando/zally/blob/master/server/src/main/resources/api/zally-api.yaml#/schemas/LintingRequest'`. のようなローカルおよびリモートのコンテンツへの参照を含まないようにすべきです。その理由は参照コンテンツが *一般的には* 、**永続的ではない** し、**不変でもない** からです。 結果としてAPIのセマンティクスが予期せぬ形で変わる可能性があります。

ですが、次のURLで示されるリソースへのリモート参照は使ってもよいこととします。

- `https://opensource.zalando.com/problem/` (see **MUST** Problem JSONを使う)
- `https://zalando.github.io/problem/` (deprecated alias for **MUST** Problem JSONを使う)

これらのURLは私たちが管理していて、**永続的** で **不変** であることを保証するからです。**MUST** 成功とエラーレスポンスを規定する で提案するように、このソースを使ってAPI仕様を定義することができます。

SHOULD APIのユーザマニュアルを提供する

API仕様に加えて、APIのユーザマニュアルも提供することは、クライアント開発者(とくにそのAPIを使った開発経験があまりない人)にとってとてもありがたいことです。APIユーザマニュアルは、以下のような観点を記述するとよいでしょう。

- APIのスコープ、目的、ユースケース
- 具体的な使用例
- 境界値、エラー時の詳細、修正のヒント
- アーキテクチャと主要な依存関係 (図やシーケンスがあるとよい)

ユーザマニュアルはオンラインで公開されなければなりません。API仕様中の `#/externalDocs/url` プロパティで書かれたリンクを、APIユーザマニュアルに含めるのを忘れないようにしましょう。

MUST APIはUS英語で書く

4. メタ情報

MUST APIメタ情報を含める

API仕様はAPI管理のための、次のOpenAPIメタ情報を含まなければならない。

- `#/info/title` : APIを(一意に)識別でき、その機能を表す名前
- `#/info/version` : API仕様のバージョン。 セマンティックルール にしたがう。
- `#/info/description` : APIの説明
- `#/info/contact/{name,url,email}` : 担当のチームの情報

Open APIの拡張プロパティにしたがい、以下も **定義しなければなりません** 。

- `#/info/x-api-id` : APIの一意の識別子 (215 ルール参照)
- `#/info/x-audience` : intended target audience of the API (219 ルール参照)

MUST セマンティックバージョニングを使う

Open APIはAPI仕様のバージョンを、 `#/info/version` で指定します。 バージョン情報の共通の意味を共有するために、私たちはAPI設計者に セマンティックバージョン 2.0 の ルール 1 から 8 までと 11 に準拠することを期待します。 それは<MAJOR>.<MINOR>.<PATCH>の形式で、次のような意味を与えます。

- 非互換なAPIの変更をしたら、 **MAJOR** バージョンをあげる。
- 後方互換性を保ちつつ、新規機能を追加したら、 **MINOR** バージョンをあげる。
- 機能に影響のない後方互換性を保ったバグフィクスや表記上の修正をしたら **PATCH** バージョンをあげる。

追加の注意:

- **プレリリース** バージョン(rule 9) と **ビルドメタデータ** (rule 10) は、 APIバージョン情報に使ってはいけない

- パッチバージョンはtypoの修正になどに役立つけれども、API設計者が、 それでバージョンをあげるかどうかは自由である
- API設計者はAPIバージョン `0.y.z` は、初期API設計のために使うべきである (rule 4)

例:

```
openapi: '3.0.1'
info:
  title: Parcel Service API
  description: API for <...>
  version: 1.3.7
  <...>
```

MUST API識別子を提供する

それぞれのAPIは一意でイミュータブルなAPI識別子が与えられます。 API識別子は、Open API仕様の `info`-ブロックで定義します。 そしてこれは、次の仕様に準拠しなければなりません。

```
/info/x-api-id:
  type: string
  format: urn
  pattern: ^[a-z0-9][a-z0-9-:.]{6,62}[a-z0-9]$
  description: |
    全体で一意で、イミュータブルなAPI識別子。API IDによって、
    API仕様の進化と履歴を一連のバージョンとしてトラッキングできる。
```

API仕様は発展し、Open API仕様の観点も変わっていくでしょう。 私たちがAPI識別子を要求するのは、API利用者や提供者に 変更のトレースや、履歴、自動互換性チェックのような APIのライフサイクル管理機能をサポートしたいためです。 イミュータブルなAPI識別子によって、APIの発展にともなうすべてのAPI仕様のバージョンを 識別可能になります。 APIセマンティックバージョン情報 や API公開日 を順序性の基準として使うことで、 **バージョン** や **公開履歴** を一連のAPI仕様として 取得できることでしょう。

注意: 自分で管理するURNを使って、可読性のあるAPI識別子を使うのはよいことではありますが、 APIの進化の過程で、API設計者がAPI識別子を変更したくなる衝動がきっと出てくるので、 UUIDを使っておくことをおすすめします。

例:

```yaml
openapi: '3.0.1'
info:
  x-api-id: d0184f38-b98d-11e7-9c56-68f728c1ba70
  title: Parcel Service API
  description: API for <...>
  version: 1.5.8
  <...>
```

MUST APIオーディエンスを提供する

それぞれのAPIはAPIの利用が想定されている対象オーディエンスで分類されなければなりません。 それごとに見つけやすさ、変わりやすさ、設計とドキュメントの品質、権限付与などAPIの異なる標準を 容易にするためです。 私たちは、次に示すようなAPIオーディエンスグループを使って、組織や法的な境界で分類しています。

component-internal

> このオーディエンスのAPI利用者は、同一 **機能コンポーネント** のアプリケーションに制限される。 機能コンポーネント/プロダクトのすべてのサービスは、専門のオーナーとエンジニアチームが管理する。 component-internal APIの典型例には、内部のヘルパーやワーカーサービスに利用されるもの、 サービス運用を支援するものがある。

business-unit-internal

> このオーディエンスのAPI利用者は、同一の業務ユニットが持っている 固有のプロダクトポートフォリオのアプリケーションに制限される。

company-internal

> このオーディエンスのAPI利用者は、同一企業の業務ユニット (例えば Zalando SE, Zalando Payments SE & Co. KG. など) が持つアプリケーションに制限される。

external-partner

> このオーディエンスのAPI利用者は、APIを所有している企業の提携企業と、その企業自身の アプリケーションに制限される。

external-public

> このオーディエンスのAPIは、インターネットにアクセスできる誰でも利用できる。

注意: より小さなオーディエンスグループは、より大きなグループに含まれることを意味します。 したがってオーディエンスグループを追加で宣言する必要はありません。

APIオーディエンスは、Open API仕様の `info`- ブロックにAPIメタデータとして含めます。 そして、次の仕様に準拠しなければなりません。

```
#/info/x-audience:
  type: string
  x-extensible-enum:
    - component-internal
    - business-unit-internal
    - company-internal
    - external-partner
    - external-public
  description: |
    対象とするAPIのオーディエンス。設計とドキュメント、レビュー、探しやすさ、
    変更しやすさ、権限付与などの質に標準に影響する。
```

注意: API仕様につき、オーディエンスは正確に **1つだけ** です。その理由は、小さなオーディエンスグループは、大きなオーディエンスグループに含まれるからです。もしAPIの一部が異なる対象オーディエンスを持つのであれば、 API仕様を分割することをおすすめします。たとえ冗長だとしてもです。

例:

```
swagger: '3.0.1'
info:
  x-audience: company-internal
  title: Parcel Helper Service API
  description: API for <...>
  version: 1.2.4
  <...>
```

5. セキュリティ

MUST OAuth 2.0でエンドポイントをセキュアにする

すべてのAPIエンドポイントはOAuth 2.0を使ってセキュアにする必要があります。 API仕様におけるセキュリティ定義のやり方は、 公式のOpenAPI仕様 を参照してください。 次に例も示しておきます。

```
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

次のコードスニペットは、このセキュリティスキームをすべてのAPIエンドポイントに適用するものです。クライアントのbearerトークンは、scope_1とscope_2のスコープを追加でもたなければなりません。

```
security:
  - BearerAuth: [ scope_1, scope_2 ]
```

MUST 権限を定義し割り当てる (スコープ)

APIはリソースを保護するために権限を定義しなければなりません。 少なくとも1つの権限が、それぞれのエンドポイントに割り当てられなければなりません。 権限は 前節 で示したように定義されます。

権限のスキーマの命名は、 ホスト名 と イベント型名 の命名に対応しています。 権限名の設計には **MUST** 権限(スコープ)の命名規約にしたがう を参照ください。

権限の種類が多く細かくなり過ぎて、複雑なガバナンスを強いられることがないよう、 リソース拡張なしで、コンポーネント固有の権限を使うことにこだわりましょう。 大概のユースケースでは、(read と write違いで)特定のAPIへのアクセスを制限することは、荷主か小売か、カスタマか運用スタッフか、といったクライアントの種類によってアクセスを制御するのに十分なものです。 ただ、APIが異なるオーナーには異なるリソースを返すような状況下では、 リソース固有のスコープは意味があるかもしれません。

標準とリソース固有の権限の例を以下に示します。

Application ID	Resource ID	Access Type	Example
order-management	sales_order	read	order-management.sales_order.read
order-management	shipment_order	read	order-management.shipment_order.read
fulfillment-order		write	fulfillment-order.write
business-partner-service		read	business-partner-service.read

権限名を定義し、権限をAPI仕様の先頭でセキュリティ定義として宣言したら、 `セキュリティ要求 を記述して、以下のように各API操作に割り当てます。

```
paths:
  /business-partners/{partner-id}:
    get:
      summary: ビジネスパートナーについての情報を取得する
      security:
        - BearerAuth: [ business-partner-service.read ]
```

非常にレアケースですが、API全体または、その内いくつかのエンドポイントが、特定のアクセス制御を必要としない ことがあります。しかし、この場合も uid の疑似アクセス権スコープを明示的に割り当てるようにすべきです。 これはユーザIDで、OAuth2のデフォルトスコープとして常に利用できます。

```
paths:
  /public-information:
    get:
      summary: Provides public information about ...
               Accessible by any user; no access rights needed.
      security:
        - BearerAuth: [ uid ]
```

ヒント: "Authorization" ヘッダを明示的に定義する必要はありません。 セキュリティセクションが定義されていれば、暗黙的にそれは標準ヘッダとなるからです。

MUST 権限(スコープ)の命名規約にしたがう

Functional naming が権限にもサポートされない限り、 APIの権限名は次の命名パターンに準拠しなければなりません。

```
<permission> ::= <standard-permission> |  --
大部分のユースケースでこれを使うべき
                 <resource-permission> |  --
異なるユースケースへの特別なセキュリティアクセスのため
                 <pseudo-permission>      --
アクセスが制限されないことを明示的に指し示すのに使う

<standard-permission> ::= <application-id>.<access-mode>
<resource-permission> ::= <application-id>.<resource-name>.<access-mode>
<pseudo-permission>   ::= uid

<application-id>      ::= [a-z][a-z0-9-]*  -- アプリケーション識別子
<resource-name>       ::= [a-z][a-z0-9-]*  -- 自由なリソース識別子
<access-mode>         ::= read | write    -- 将来拡張されるかもしれない
```

このパターンは、以前の定義とも互換性があります。

6. 互換性

MUST 後方互換性を崩してはならない

APIの変更は、すべての利用者の動作に影響がないようにおこなわなければなりません。 通常、利用者はAPIのリリースサイクルからは独立していて、安定性に注力しており、 付加価値をうまない変更は避けようとします。 APIはサービスプロバイダとサービスコンシューマの間の契約であり、一方の都合だけで それを破棄することはできません。

それを実現するための2つの方法があります。

- 互換性拡張のルールにしたがう
- APIに新しいバージョンを導入し、古いバージョンもサポートする

私たちは互換性を維持したAPIの拡張を強く推奨し、バージョニングは最後の手段とします。(以下に示すルールをみてください **SHOULD** バージョニングを避ける,**MUST** メディアタイプバージョニングを使う) サービスプロバイダが従うガイドライン(**SHOULD** 互換性を維持した拡張をやろう) と コンシューマが従うガイドライン(**MUST** 互換性維持のAPI拡張を受容できるクライアントを用意する) で、バージョニングなしに(ポステルの法則を胸に)互換性をたもった変更ができるようになります。

注意: 非互換性とは破壊的変更には違いがあります。非互換は変更とは、以下の互換性ルールを満たしていない変更です。破壊的変更は非互換な変更を本番環境へデプロイすることです。したがって今動いているAPIコンシューマをも破壊することを意味します。通常は、非互換の変更は本番環境にデプロイされたとき破壊的変更になりますが、どのAPIコンシューマにも影響を与えないのであれば、破壊的変更をせずに本番環境に非互換な変更をデプロイ可能です。(廃止予定 ガイドライン参照)

ヒント: 互換性の保証は"オンザワイヤー"形式に対して行われます。API定義から生成されたバイナリまたはソースコードの互換性は、このルールの範疇ではありません。もしクライアントの実装がAPI定義の新しいバージョンへ追従するために更新されるならば、コードの変更が必要だと予想されます。

SHOULD 互換性を維持した拡張をやろう

APIの設計者は後方互換性を維持しつつ、RESTful APIを進化させていくために、次のルールにしたがうべきです。

- 任意のフィールドのみ追加して、必須のフィールドは追加してはならない。
- フィールドの意味は決して変えてはならない (例えばcustomer-numberをcustomer-idに変更することは、両者はカスタマの一意キーとしての意味は異なるのでNG)
- サーバサイドのビジネスロジックでバリデーションしなきゃいけないような(複雑な)制約をもつ入力フィールド。バリデーションロジックは、より厳しくなる方向には変更してはいけません。すべての制約はdescriptionに明示します。
- 入力パラメータとして使われる列挙型の要素は、サーバが古い値も受け付けて正しくハンドリングできる場合のみ減らすことができる。出力パラメータとして使われる列挙型はいつでも減らすことはできる。
- 出力パラメータとして使われる列挙型は、クライアントがハンドリングできないかもしれないので追加してはならない。入力パラメータとして使われる列挙型はいつでも追加できる。
- 出力パラメータととして使用され将来の拡張も考えておきたい場合は、`x-extensible-enum`を使う。 明示的に値を上限なしリストと定義し、クライアントは新しい値には依存しない設計をしなければならない。
- URLを変更するときはリダイレクションをサポートする 301 (Moved Permanently).

MUST 互換性維持のAPI拡張を受容できるクライアントを用意する

サービスクライアントにはロバストネスの原則を適用すべきです。

- APIリクエストと入力として渡すデータは保守的に。例えば最大長が定義されていないからといって、数メガバイトの文字列を渡すようなことは避けよう。
- APIレスポンスのデータの処理や読み込みについては特に寛容に。

サービスクライアントはサービスプロバイダの互換性あるAPI拡張に対して、準備しておかなければなりません。

- ペイロードの未知のフィールドに対して寛容でなければならない。続くPUTリクエストで必要とされるならば、ペイロードから削除せず新しいフィールドは無視する。 (ファウラーの "TolerantReader" の記事もみてください)
- 未知の値について予測できなくても、デフォルトの振る舞いが与えられていても、`x-extensible-enum`の戻りパラメータは、新しい値を含んでいると思って設計する
- エンドポイントの定義に明記されてないHTTPステータスコードがきても、ハンドリングできるようにしておく。またステータスコードは拡張可能なことにも注意しよう。デフォルトのハンドリングは、関連する2xxコードをどう扱うかである。(RFC7231 Section 6 も見てください)
- サーバがHTTPステータス301(Moved Permanently)を返したら、リダイレクトを追従しよう。

SHOULD APIを保守的に設計する

サービスプロバイダAPIの設計者は、クライアントから受け付けるものについて保守的で正確であるべきです。

- ペイロードやURL中の未知のフィールドは無視すべきでない。サーバは400のレスポンスコードを返して、 クライアントにエラーである旨を通知すべきである。
- 入力データの制約(フォーマット、範囲、長さなど)の定義には正確にしたがい、入力チェックして違反があれば、専用のエラーを返す。
- (機能要求に準拠している限りでは) 例えば文字列の長さの範囲を定義するなどして、より限定的で制限の強い方を選択する。そうすることで、互換性ある拡張として進化の自由を与えつつ、実装を単純化できるかもしれません。

未知の入力フィールドを無視しないというのは、ポステルの法則 (The Robustness Principle Reconsidered) から逸脱していますが、これを強く推奨します。 サーバは次のような問題に気付き、サポートするものを明示すべきです。

- 続くGETレスポンスとの対称性が失われるので、未知の入力フィールドを無視することは、PUTにとって任意ではないことになります。HTTPではPUTの *置換* セマンティクスとデフォルトの期待するラウンドトリップは明確になっています(RFC 7231 Section 4.3.4参照)。未知の入力フィールドを受け入れない(即ち無視しない)のと、続くGETレスポンスでそれを返すのとでは、異なるシチュエーションであり、PUTセマンティクスに準拠したものであることに注意してください。

- あるクライアントエラーはサーバには認識できない。例えば、属性名にタイプミスがあれば、 サーバエラーなしには無視されてしまう。クライアントが任意の入力フィールドを与えたつもり だったとしても、サーバはクライアントの意図した追加のフィールドなのか、フィールド名を間違って送ったのか区別できないのである。
- 入力データ構造の将来の拡張は、すでに無視されているフィールドと競合するかもしれない。 そうなると、互換性は無くなるだろう。つまり、このフィールドを別の型として既に使ってるクライアントを破壊することになる。

特定の状況では、(既知の)入力フィールドがどこからも必要とされていなければ、「not used anymore」の記述をAPI定義に書いておくか、サーバがこの特定のパラメータを無視する限りAPI定義から削除するかしましょう。

MUST 常にトップレベルのデータ構造としてJSONオブジェクトを返す

レスポンスボディには、常に将来の拡張を考慮して、常にトップレベルのデータ構造として(例えばArrayではなく)JSONオブジェクトを返さなければなりません。JSONオブジェクトは属性を追加することによって、互換性を維持した拡張ができます。これがレスポンスの拡張が簡単になる理由であり、例えば後からページネーションを追加したりということが、後方互換性を崩すことなく可能になります。

Map(**SHOULD** Mapは `additionalProperties` を使って定義する参照)は互換性のある将来の拡張をサポートしないので、トップレベルのデータ構造としても禁止されています。

MUST Open APIの定義をデフォルトで拡張に対して開かれているものとして扱う

Open API 2.0仕様では、オブジェクトのデフォルト拡張についてはあまり仕様化されておらず、 拡張に関しては `additionalProperties` のように、JSONスキーマキーワードを再定義したものになっています。 私たちの互換性ガイドライン全般にしたがうと、Open APIオブジェクト定義は、JSONスキーマの Section 5.18 "additionalProperties" のようにデフォルトで拡張に対して開かれているとみなすことができます。

Open API 2.0に関していえば、これは `additionalProperties` 宣言が、オブジェクト定義を拡張可能にする必要がないことを意味します。

- データを受け取るAPIクライアントが、 `additionalProperties` 宣言が無いからといって、 拡張がされないものと仮定してはならないし、サーバから送られてきた処理できないフィールドは無視しなくてはならない。そうすることで、APIサーバはデータフォーマットを拡張していけるようになる。
- APIサーバが予期しないデータを受け取るときは、ちょっと事情が異なる。フィールドを無視する代わりにクライアントにこれらのフィールドが保存されなかったことを通知するために、サーバは定義されていないフィールドを含むリクエストを拒否 __してもよい__ 。API設計者はPUT,POST,PATCHリクエストについて、予期しないフィールドをどう扱うか、ドキュメントに明記しなければならない。

APIフォーマットは `additionalProperties` をfalseと宣言してはなりません。将来的にオブジェクトが拡張できなくなるからです。

このガイドラインはデフォルトの拡張可能性に焦点を当てているのであって、 ある状況下では単なる値とし

て additionalProperties を使うことを否定はしていません。 例えば、 **SHOULD** Mapは additionalProperties を使って定義する を参照。

SHOULD 列挙型の代わりに、上限なしの値リスト(x-extenible-enum)を使う

列挙型は値の閉集合であり、完全性が仮定されていて拡張は意図されていません。この列挙型のクローズドな原則は、これを拡張しなきゃいけなくなったときに互換性の問題となってあらわれます。これの問題を回避するために、列挙型の代わりに、上限のない値リストを使うことを強く推奨します。

例外として以下の場合は列挙型を使用してもかまいません。

1. 例えば値のリストが外部のツールやインタフェースに依存しないなど、APIが列挙型の値を完全に制御できる
2. 将来の機能を考慮可能、不可能に関わらず完全な値リストである

上限なしの値リストを特定するために、次のように x-extensible-enumのマーカーを使います。

```
deliver_methods:
  type: string
  x-extensible-enum:
    - parcel
    - letter
    - email
```

注意: x-extensible-enum は、JSONスキーマに準拠していませんが、大抵のツールには無視されます。

SHOULD バージョニングを避ける

RESTful APIを変更するときは、互換性をたもつ方法でおこない、APIのバージョンが新たに作られてしまうことを避けましょう。複数のバージョンはシステムを理解するのも、テストするのも、保守するのも、進化させるのも、運用するのも、リリースするのも全部を複雑化してしまいます。 (こちらも参照ください)

互換性を維持する方法でAPIを変更出来ないのであれば、以下の3つのどれかを選択してください。

- 古いリソースのバリアントに追加する形で、新しいリソース(バリアント)を作る。
- 新たにエンドポイントを作る。 つまり、新しいAPIをもった(新しいドメイン名で)新しいアプリケーションを作るということです。
- 同じマイクロサービスで古いAPIもサポートしつつ、新しいバージョンのAPIを作る。

さまざまなデメリットがあるので、バージョニングは何としても避けたいところで、私たちは最初の2つのアプローチのみを使うことを強く推奨しています。

MUST メディアタイプバージョニングを使う

APIバージョニングを避けられないのであれば、(URIバージョニングの代わりに、以下に示すように)メディアタイプバージョニングを利用したマルチバージョンRESTful APIを設計しなければなりません。メディアタイプバージョニングは、コンテントネゴシエーションをサポートするので、密結合度合いは緩和されます。したがってリリース管理の複雑さも減少することでしょう。

メディアタイプバージョニング: バージョン情報とメディアタイプは、Content-TypeのHTTPヘッダで与えられます。例えば application/x.zalando.cart+json;version=2 のように。非互換な変更があるときは、リソースに新しいメディアタイプバージョンがふられます。新しいバージョンを生成するために、コンシューマとプロデューサはContent-TypeとAcceptのHTTPヘッダを使ってコンテントネゴシエーションできるのです。注意: このバージョニングはURIやメソッドには適用できません。リクエストおよびレスポンスのコンテントスキーマにのみ適用可能です。

この例では、クライアントはレスポンスの新しいバージョンのみをリクエストします。

```
Accept: application/x.zalando.cart+json;version=2
```

クライアントと同様に、サーバもContent-Typeヘッダに新しいバージョンを送る宣言をして レスポンスします。

```
Content-Type: application/x.zalando.cart+json;version=2
```

ヘッダバージョニングを使うべきなのは、以下の点にあります。

- リクエストとレスポンスのヘッダにバージョンを含めることで可視性が増す
- バージョンごとのプロキシキャッシュを有効にするために、Content-TypeをVaryヘッダに含めることができる

ヒント: 非互換の変更が必要になるまでは、通常の `application/json` メディアタイプのままにしておきましょう。

さらに: APIバージョニングに「正解」はない では、自説にこだわることなく破壊的変更をどう扱うかを、異なるバージョニングのアプローチで全体感を述べています。

MUST URIバージョニングを使わない

URIバージョニングとは、/v1/customers のように、パスに(メジャー)バージョン番号を含ませる方法です。API利用者は、APIプロバイダがデプロイされリリースされるまで待たなくてはなりません。もしコンシューマもまた、ワークフローを追従できるよう(HATEOAS)ハイパーメディアリンクをサポートするのであれば、これはたちまち複雑化します。特にハイパーリンクで結ばれたサービス依存関係のあるところで、URLバージョンニングを使うと、バージョンアップの調整もまた困難です。この密結合で複雑なリリース管理になるのを避けるためには、URIバージョニングは避けたほうがよいでしょう。代わりに(上で示したような)メディアタイプバージョニングとコンテントネゴシエーションを使いましょう。

7. 廃止予定

APIエンドポイント(または、そのバージョン)を、廃止する必要が出てくることがあります。例えば、もはやサポートされていないフィールドや、業務機能ごと停止したいエンドポイントなど、様々な理由があることでしょう。これらのエンドポイントは、利用者に使われている限りは、破壊的変更は許されません。利用者にとって必要な変更を整理し、廃止予定のエンドポイントがAPIの変更がデプロイされる前に使われないようにするため、「廃止予定ルール」を適用します。

MUST クライアントの承認を得る

API(またはAPIのバージョン)を停止する前に、すべてのクライアントに、そのエンドポイントを停止してもよいという同意をとらなければなりません。(移行マニュアルを提供するなどして) 新しいエンドポイントへ移行の手助けをしてください。 すべてのクライアントが移行が完了したら、廃止予定のAPIを停止できます。

MUST 外部パートナーは廃止までの猶予期間に同意をしなければならない

もし外部パートナーによってAPIが使われていたら、廃止予定をアナウンスしたあとAPIの現実的な移行猶予期間を定義しなければなりません。外部パートナーはAPIを使い始める前に、この最小の移行猶予期間に同意しなければなりません。

MUST API定義に廃止予定を反映する

APIの廃止予定は、OpenAPI定義に含まれなくてはなりません。とあるメソッド、パス全体、(複数のパス含む)APIエンドポイントまるごと、いずれにしてもそれらを廃止予定とするならば、メソッド/パスエレメントそれぞれに `deprecated: true` を設定しなければなりません。 もし廃止予定がより詳細なレベルで必要であれば、影響する要素に `deprecated=true` を設定したうえで、`description` に説明書きを加えます。

`deprecated` に `true` が設定されたら、クライアントが代わりに使うべきものやAPIがいつ廃止されるのかを、API定義の `description` に記述しなければなりません。

MUST 廃止予定APIとAPIエンドポイントの利用状況をモニタリングする

本番環境で使われるAPIのオーナーは、APIが廃止予定を調整し、コントロールできない 破壊的影響を避けるため、APIが廃止されるまで、廃止予定APIの利用状況をモニタリングしなくてはなりません。 **SHOULD** API利用状況をモニタリングするも参照ください。

SHOULD レスポンスにDeprecationヘッダを付ける

廃止予定フェーズの間、 `Deprecation` ヘッダを付けましょう。 (draft: RFC Deprecation HTTP Headerをみてください)。 廃止予定日(HTTP Date/Time形式)を値として付けます。

```
Deprecation: Sun, 11 Nov 2018 23:59:59 GMT
```

`Deprecation` ヘッダを付けても、APIを廃止することをクライアントの合意を取り付けたことにはならないことに注意しましょう。

ヒント: ガイドラインの初期バージョンでは、`Warning` ヘッダを使うようにしていました。しかし、`Warning` ヘッダはあまりその意味がハッキリしないので draft: RFC HTTP Caching では廃止になるようです。私たちの構文も、 RFC 7234 — Warning header に合っていませんでした。

SHOULD Deprecationヘッダのモニタリングを追加する

クライアントはHTTPレスポンスの `Deprecation` ヘッダをモニタリングし、APIが将来廃止されることがあるかどうかを注視してください。

ヒント: ガイドラインの初期バージョンでは、`Warning` ヘッダを使っていました。**SHOULD** レスポンスにDeprecationヘッダを付けるを参照してください。

MUST 廃止予定APIは新規に利用し始めてはならない。

クライアントは、廃止予定のものを利用し始めてはなりません。

8. JSONガイドライン

ZalandoにおいてJSONデータを定義するのに推奨されるガイドラインです。 JSONとは、ここでは RFC 7159 (RFC 4627 のアップデート)を指します。 "application/json"のメディアタイプとAPIで定義されたカスタムのJSONメディアタイプをもちます。 このガイドラインでは、Zalandoの用語やサービスの用例をもつJSONデータを使って、具体的なケースを示します。

最初のいくつかはプロパティ名についてのガイドラインであり、 後半は値についてのガイドラインになります。

MUST プロパティ名はASCIIスネークケースでなければならない (キャメルケースは使わない): `[a-z_][a-z_0-9]*$`

プロパティ名は、ASCII文字列という制限があります。最初の一文字はアルファベットまたはアンダースコアで、 それに続く文字は、アルファベットまたはアンダースコア、数字のいずれかでなくてはなりません。

(*`links` のようなキーワードのみ、* から始まるプロパティ名とすることを推奨します)

理念: 確立された標準は存在しませんが、多くの有名インターネット企業は、スネークケースを好みます。GitHub, Stack Exchange, Twitterなど。一方でGoogleやAmazonは、- だけでなくキャメルケースも 使っています。同じところからくるJSONが一貫したルック・アンド・フィールとなるように設計するのは 必要不可欠なことです。

MUST enumの値は、UPPER_SNAKE_CASE形式で宣言する

Enumの値 (`enum` または `x-extensible-enum`で使われる) は、大文字のスネークケース形式を一貫して使う必要があります。 `VALUE` や `YET_ANOTHER_VALUE` のように。このすると、プロパティや他の要素との区別がハッキリつくためです。

SHOULD Mapは `additionalProperties` を使って定義する

ここで「map」は、文字列のキーから他の型へのマッピングを意味します。 JSONにおいてこれはオブジェクトとして表現されます。キーと値のペアはプロパティ名とプロパティの値によって 表現されます。 OpenAPIスキーマにおいては(JSONスキーマにおいても同様)、それらはadditionalPropertiesを使って 表現すべきとされます。そのようなオフジェクトは他に定義されたプロパティは持ちません。

mapのキーは、命名ルール 118の意味ではプロパティ名とみなしませんので、ドメイン固有のフォーマットにしたがうようにします。 ドキュメントにはmapオブジェクトのスキーマの詳細に、これを記述するようにしてください。 これはそのようなmapの例です。(`transactions` プロパティがそれにあたります)

```
components:
  schemas:
    Message:
      description:
        いくつかの言語に翻訳したメッセージ
      type: object
      properties:
        message_key:
          type: string
          description: メッセージのキー
        translations:
          description:
            このメッセージをいくつかの言語に翻訳したもの。
            キーは https://tools.ietf.org/html/bcp47[BCP-47 言語タグ] である。
          type: object
          additionalProperties:
            type: string
            description:
              キーによって識別された言語に、このメッセージを翻訳したもの
```

実際のJSONオブジェクトは次のようなものです。

```
{ "message_key": "color",
  "translations": {
    "de": "Farbe",
    "en-US": "color",
    "en-GB": "colour",
    "eo": "koloro",
    "nl": "kleur"
  }
}
```

SHOULD Arrayの名前は複数形にする

複数の値をもつArrayのプロパティ名は複数形にします。これはオブジェクトの名前は単数形にすべきということも暗に示しています。

MUST Booleanのプロパティに `null` を使わない。

booleanとして設計されたJSONプロパティは、スキーマ上nullであってはなりません。 booleanはtrueとfalseの2つの値をもった列挙型です。もしnull値をもちたいことがあれば、 booleanの代わりに列挙型を使うことを強く奨めます。 例えばaccepted_terms_and_conditionsがtrueまたはfalseをもつとき、 これはyes/no/unknownの値をもったterms_and_conditionsに置き換えることができます。

MUST `null` とプロパティ自体が無いことは同一セマンティクスとして使う

Open API 3.x では、プロパティが無い(`{}`)かもしれないことと、`null` (`{"example":null}`)を表すのに、`required` と `nullable` を付けることができます。 もし、あるプロパティが `required` でなく `nullable` だと定義されていたら(下表の2行目のように)、このルールは、双方を正確に同じ方法で扱えるようにしなければなりません。

次の表は、すべての組み合わせとそれが有効かどうかを示します。

required	nullable	{}	{"example":null}
`true`	`true`	☒ No	☒ Yes
`false`	`true`	☒ Yes	☒ Yes
`true`	`false`	☒ No	☒ No
`false`	`false`	☒ Yes	☒ No

APIの設計者や実装者は、どちらの場合も異なる意味を割り当てたくなるかもしれませんが、表現力が高いことよりも、クライアントがその微妙な違いを理解せず誤って実装してしまうリスクの方がはるかに高いと考えられるので、私たちはこの選択肢には反対です。

例えばミーティングのように、異なるユーザがスケジュールを調整できるようなAPIでは、各ユーザが 選択しなければならない選択肢をリソースとできます。 *undecided* と *decided* の違いは、それぞれ *absent* と `null` としてモデル化できる。 `null` の場合は、 Null object (例えば、`{"id":"42"}` に対しての`{}` のように)として表すのが安全です。

さらに多くの主要なライブラリは、 `null` / absent パターンをほとんどサポートしていないか、まったくサポートしてないかという状態です。(Gson, Moshi, Jackson, JSON-B) 特に、強い型付けを持つ言語では、第3の状態を表現するためには新しい複合型が必要なため、この問題に頭を悩ます。Nullable `Option` / `Optional` / `Maybe` 型を使うことができますが、これらの型のNullableな参照を持つことは、その型の目的と完全に矛盾してしまいます。

このルールの唯一の例外はJSON Merge Patch RFC 7396)で、これは `null` を使ってプロパティの削除を明示的に示しますが、存在しないプロパティは無視されます。つまり変更されません。

SHOULD 空のArray値はnullにはしない

Arrayが空であることは `[ ]` として曖昧さなく表現できます。

SHOULD 列挙型はStringとして表現する

Stringは列挙型で設計された値を表現するには妥当な型です。

SHOULD 日付/日時のプロパティには `_at` をサフィックスとして付ける

日付と日時のプロパティは `_at` で終わるようにすべきです。よく似た名前のbooleanプロパティと区別できるようになります。

- `created_at` rather than `created`,
- `modified_at` rather than `modified`,
- `occurred_at` rather than `occurred`, and
- `returned_at` rather than `returned`.

注意: `created` と `modified` はガイドラインの以前のバージョンで言及されていました。したがって、このルールより前のAPIでも引き続き受容されます。

SHOULD 日付型のプロパティ値はRFC 3339に準拠する

RFC 3339 で定義された日付と時刻のフォーマットを使いましょう。

- "date"には `年 "-" 月 "-" 日` を使う。例: `2015-05-28`
- "date-time"には `年-月-日 "T" 時:分:秒` を使う, 例: `2015-05-28T14:07:17Z`

Open API フォーマット の"date-time"はRFCの"date-time"に相当し、`2015-05-28`のして表されるOpen

APIフォーマットの "date"は、RFCの"full-date"に相当します。 どちらもspecific profilesで、国際標準 ISO 8601 のサブセットです。

(リクエストとレスポンスの両方で) ゾーンオフセットが使われる可能性があります。 これも標準で定義されているものです。 しかし、私たちは日付に関しては、オフセットなしのUTCを使うよう制限を設けることを推奨しています。`2015-05-28T14:07:17+00:00` ではなく、`2015-05-28T14:07:17Z` のように。 これはゾーンオフセットは理解が難しく、正しく扱えないことがよくあることを経験上学んだので、そうしています。 ゾーンオフセットはサマータイムを含むローカルタイムとは異なることに注意してください。 日時のローカライズは、必要ならユーザインタフェースを提供するサービスによってなされるべきです。 保存するときは、すべての日時データはゾーンオフセットなしのUTCで保存します。

時々、数値タイムスタンプで日時を表すデータを見かけますが、 これは精度に関しての解釈の問題を引き起こします。 例えば1460062925というタイムスタンプの表現は、1460062925000 なのか 1460062925.000 なのか 判別できません。日時文字列は冗長でパースが必要ですが、この曖昧さを避けるために必要なことなのです。

MAY 期間(duration)と時間間隔(interval)はISO8601に準拠する

期間と時間間隔の設計は、 ISO 8601で推奨されている形式の文字列を使います。 (期間については 付録A RFC 3339に文法が含まれます)

9. APIの命名

MUST/SHOULD 機能本位の命名体系を使う

機能本位の命名は、あるアプリケーション群の中で、*ホスト* や *権限*、 *イベント名* のような各所から利用されるリソースに、整合性をもたせる強力で簡単な方法です。 そうすることで、コンポーネントについての意味のあるコンテキスト情報を 読み手に提供しつつ、名前の一意性も保証できます。 さらにもっとも重要なのは、技術的、組織的な変化の中でAPIを安定した状態が保たれるという点です。

拡大していく オーディエンス とともにAPIがこの利点を享受できるよう、 ホスト名 や 権限名、 イベント名 に関して、 次のような機能本位の命名体系にしたがうことを強く推奨します。

Functional Naming	オーディエンス
must	external-public, external-partner
should	company-internal, business-unit-internal
may	component-internal

機能本位の命名体系に導くために、 一意な `functional-name` が各機能コンポーネントに割り当てられます。 それはコンポーネントが属する機能グループのドメイン名からなり、 機能コンポーネントを一意に識別できる短い名前です。

```
<functional-name>       ::= <functional-domain>-<functional-component>
<functional-domain>     ::= [a-z][a-z0-9]*  --
管理されたコンポーネントの機能グループ
<functional-component>  ::= [a-z][a-z0-9-]* -- 機能コンポーネント自身の名前
```

機能の命名パターンの詳細ルールについては、以下のものも参照ください。

- **MUST** ホスト名の命名規約にしたがう
- **MUST** イベント型の名前は命名規約にしたがう

MUST ホスト名の命名規約にしたがう

APIにおけるホスト名は、以下に示すようにそれぞれ オーディエンス ごとに 定められた機能の命名ルールに準拠すべきです。 (詳細は **MUST**/**SHOULD** 機能本位の命名体系を使う の `<functional-name>` 定義を参照ください)

```
<hostname>              ::= <functional-hostname> | <application-hostname>

<functional-hostname>   ::= <functional-name>.zalandoapis.com
```

次に示すアプリケーション固有のレガシーな規約は、 component-internal APIのホスト名に **だけ** 適用できます。

```
<application-hostname> ::= <application-id>.<organization-unit>.zalan.do
<application-id>       ::= [a-z][a-z0-9-]*  -- アプリケーション識別子
<organization-id>      ::= [a-z][a-z0-9-]*  --
組織単位の識別子。例えば、チームID
```

MUST パスセグメントはハイフンで区切られた小文字を使う

例:

```
/shipment-orders/{shipment-order-id}
```

このルールは具体化されたパスセグメントに適用され、パスパラメータの名前は この限りではありません。例えば`{shipment_order_id}`は、パスパラメータとしてはOKです。

MUST クエリパラメータは、スネークケースを使う(決してキャメルケースにしない)

例:

```
customer_number, order_id, billing_address
```

SHOULD HTTPヘッダはハイフン区切りのパスカルケースにする

これは一貫性のためのルールです(他のほとんどのヘッダがこの規約にしたがっているためです)。 (ハイフンなしの)キャメルケースにするのは避けましょう。例外は「ID」のような共通の略語です。

例:

```
Accept-Encoding
Apply-To-Redirect-Ref
Disposition-Notification-Options
Original-Message-ID
```

参考: HTTPヘッダは大文字・小文字を区別しない (RFC 7230)

共通のヘッダ と 独自ヘッダ の章に、HTTPヘッダに関する ガイダンスがもう少しあるので参照ください。

MUST リソース名は複数形にする

ふつうリソースインスタンスのコレクションが提供されます(すくなくともAPIは用意されるべきです)。 リソースがシングルトンである特別な場合が、カーディナリティ1のコレクションと考えます。

SHOULD ベースパスとして /api を付けない

たいていの場合、サービスによって提供されるすべてのリーソースは、公開APIの一部です。 それゆえにベースパスであるルート"/"は利用可能にしておくべきです。

もし非公開の内部APIもサポートする必要があるならば(例えばサービスが特定の運用サポート機能のために)、2つの異なるAPI仕様をメンテナンスし、API audienceを提供することを推奨します。 双方のAPIとも `/api` をベースパスとして使うべきではありません。

私たちはAPIのベースパスを、デプロイ時の可変な設定の一部として考えています。したがって、この情報は、 server object に定義されなければなりません。

MUST 末尾のスラッシュを避ける

末尾スラッシュに特定の意味をもたせてはなりません。リソースパスは 末尾にスラッシュがあろうがなかろうが、同じ結果を返さなければなりません。

MUST クエリストリングの規約を使う

もしソートやページネーション、フィルタ関数または他のアクションをサポートしたクエリを提供することになったら、 次に示す標準の命名規約にしたがってください。

- `q`: デフォルトのクエリパラメータ (つまりブラウザのタブ補完で使われる); skuのようにエンティティ特有の別名を持つべきである。
- `sort`: ソートに使うフィールドをカンマで繋いだリスト(**SHOULD** ヘッダとクエリパラメータのコレクションフォーマットを定義するで定義される)。ソートの方向を指示するために、フィールドには + (昇順) または - (降順) のプレフィクスが付くことがある。例: /sales-orders?sort=+id
- `fields`: フィールドのサブセットのみを取得するため。**SHOULD** フィルタリングによって部分的なレスポンスをサポートする。 参照。
- `embed`: サブエンティティの中身を展開したり、組み込んだりするためのフィールド名表現。例えば記事エンティティの中に、シルエットコードをシルエットオブジェクトに拡張するのに使われる。 `embed`を正しく実装するのは難しいので、注意してやる必要がある。
- `offset`: 数値のオフセットによるページのスタート地点。ページネーション セクション参照。
- `cursor`: ページへのOpaqueポインタで、クライアントが検査したり構築したりすることない。通常は(暗号化)ページ位置、つまり最初または最後のページ要素の識別子、ページネーションの方向、およびコレクションを再作成するために適用されたクエリフィルタをエンコードする。ページネーション セクション参照。
- `limit`: クライアントから1ページのエンティティ数を制限する数を与える。ページネーション セクション参照。

10. リソース

MUST アクションを避ける — リソースについて考える

RESTはリソースにまつわるものが全てです。したがって、Webサービスとドメインエンティティがどうやり取りするか、標準のHTTPメソッドを使ってAPIをどうモデル化するか、が関心事となります。 例えば、記事の編集するアプリケーションで、同時に1人のユーザしか編集できないように明示的にロックをしたいとします。「ロックする」というアクションの代わりに、「記事のロック」を`PUT`または`POST`で生成するようにします。

リクエスト:

```
PUT /article-locks/{article-id}
```

これは記事のロックを閲覧したり、フィルタリングしたりするサービスがすでに存在していると、追加のメリットとなります。

SHOULD 完全な業務プロセスをモデル化する

APIはプロセスを表現したすべてのリソースを含んだ、完全な業務プロセスを含めるべきです。 そうすることによって、クライアントが業務プロセスを理解し、業務プロセスの一貫した設計を推進し、ドキュメントと実装の観点から相乗効果が得られるようになり、API間の暗黙的で見えにくい依存関係を消すことができます。

おまけに、業務ロジックをクライアントサイドにシフトしてしまう「データベースの薄いラッパーAPI」を避ける効果もあります。

SHOULD *有用な* リソースを定義する

リソースはすべてのクライアントのユースケースの90%をカバーするようにしよう、というのが経験則としてあります。 *有用な* リソースは情報を必要なだけ多く含むと同時に、できるだけ小さくあるべきです。 残りの10%をサポートするよい方法は、クライアントがその必要性に応じてフィルタリングしたりembeddingできるようにすることです。

MUST URLに動詞を入れない

APIはリソースを記述します。HTTPメソッドの中にのみ、振る舞いが現れます。 したがって、URLは名詞だけを使うようにしてください。 振る舞い(動詞)を考える代わりに、郵便ポストにメッセージを投函することを考えるようにします。 例えば、URLに *キャンセル* という動詞をもたせる代わりに、「注文をキャンセルする」というメッセージを、 サーバの *キャンセル* 郵便受けに届ける、と考えるのです。

MUST ドメイン固有のリソース名を付ける

APIリソースはアプリケーションのドメインモデルの要素を表現するものです。 リソース名にドメイン固有の命名法を使うことは、開発者がリソースのもつ機能や基本的な意味を理解するのに役立ちますし、 API定義の以外にドキュメントをたくさん書かなきゃいけない必要性を軽減できます。 例えば「sales-order-items」は単に「order-items」とするよりも、その対象をはっきりと指し示しているのでより良いものといえます。 同様に「items」とするのは、一般的過ぎます。

MUST URLフレンドリなりソース識別子を使う: [a-zA-Z0-9:._-]*

URLにおけるリソースIDのエンコードを単純にするため、それらにはアルファベット、数字、アンダースコア、マイナス、コロンとピリオドのみのASCII文字列で構成します。

MUST パスセグメントによってリソースとサブリソースを識別できるようにする

いくつかのAPIリソースは、サブリソースを含んだり参照したりするかもしれません。 トップレベルのリソースではないEmbeddedサブリソースは、より高次のリソースの一部であり、 そのスコープの外からは使われないものです。 サブリソースはパスセグメントに含まれた名前と識別子によって参照されます。

使い勝手を向上させるため、パスセグメントそれぞれがリソースやリソースの集合を正しく指すような、 直感的に理解できるURLを目指すべきです。 例えば `/customers/12ev123bv12v/addresses/DE_100100101` はAPIのパスとしてあったとき、 `/customers/12ev123bv12v/addresses`, `/customers/12ev123bv12v` や `/customers` も、 原則的には妥当なパスでなくてはなりません。

基本形のURL構造:

```
/{resources}/[resource-id]/{sub-resources}/[sub-resource-id]
/{resources}/[partial-id-1][separator][partial-id-2]
```

例:

```
/carts/1681e6b88ec1/items
/carts/1681e6b88ec1/items/1
/customers/12ev123bv12v/addresses/DE_100100101
/content/images/9cacb4d8
```

SHOULD 必要なときだけUUIDを使う

IDの生成はハイトラフィックでリアルタイム性の要求されるようなユースケースでは、 スケールの点で問題を引き起こすことがあります。 UUIDは分散非協調な方法で競合することなく、かつ他にサーバ通信の必要もなく生成可能なので、この問題の解となりえます。

しかし、UUIDにはいくつかのデメリットがあります。

- 意味のない人工的なキーである。せっかく命名規約を用意しているのに、使わないのは実用的な理由から良くない。例えば、UUIDの代わりに製品属性に付けられた名前を使おう。
- 使いづらい
- 人間には覚えられないし、それを使ってコミュニケーションできない
- デバッグやログ解析に使いづらい
- かなり長い: 読めるキャラクタ形式にすると36文字にもなり、メモリや帯域の圧迫の原因となる。
- 生成順に並べることができない。
- レガシーなIDの後方互換サポートと競合するかもしれない

UUIDはID生成がボトルネックとなるようなときまで避けるべきです。 代わりに、例えばIDリソースへのPOSTしてID生成してから、エンティティリソースへの冪等なPUTをするようにできます。 特にカーディナリティが低いけれども、いろんな機能で利用されるbrand-idやattribute-idのようなものに、 マスタデータや設定データの主キーとしてUUIDを使うことは控えましょう。

また連番識別子は、発注量のような業務上の機密情報を、権限をもたない顧客にまで 漏らしてしまう可能性があることに気をつけてください。

どんな場合も、IDには数値型よりも文字列型を常に使うべきです。 これはIDの体系が進化していくときに、自由度が高くなるからです。 したがって、UUIDはformatプロパティで修飾してはいけません。

ヒント: よくランダムUUIDが使われます。 RFC 4122 のUUID バージョン4をみてください。 UUID バージョン1は、タイムスタンプを元に作るけれども、生成順にソートはできない仕様です。 ULID (Universally Unique Lexicographically Sortable Identifier) はこの欠点をなくすように作られています。 生成時間でソートするページネーションのユースケースなどでは、UUIDの代わりにULIDが使えるでしょう。

MAY ネストURLを使う/使わないはよく考える

もしサブリソースがその親リソースにアクセス可能で、親リソースなしでは存在しえないものであったら、ネストURL構造を検討しましょう。

例えば、以下のようなものです。

```
/carts/1681e6b88ec1/cart-items/1
```

しかし、リソースがそのユニークなIDによって直接アクセスされうるとしたら、 APIはトップレベルのリソースとして用意するべきです。

例えばカスタマは複数の販売注文をもちますが、販売注文にはユニークなIDがふってあって、 いくつかのサービスからは直接注文にアクセスするかもしれない場合です。

そのようなケースでは以下のようにします。

```
/customers/1681e6b88ec1
/sales-orders/5273gh3k525a
```

SHOULD リソースの型の上限を定める

サービスの開発・メンテナンスを続けていくためには、「機能分割」や「関心の分離」の設計原則にしたがい、 同一のAPI定義に異なる業務機能群を混ぜ込まないようにするべきです。 実際にAPIをつうじて機能提供されるリソースの種類は、その数に上限をもうけたほうがよいでしょう。

リソースの型はコレクションのような関連するリソース、そのメンバ、サブリソースの集合として定義されます。 例えば、下記のリソース群はカスタマ、住所、カスタマの住所の3つのリソース型として数えられます。

```
/customers
/customers/{id}
/customers/{id}/preferences
/customers/{id}/addresses
/customers/{id}/addresses/{addr}
/addresses
/addresses/{addr}
```

注意:

- `/customers/id/preferences` は、追加の識別子なしでカスタマと1対1の関係をもつので、`/customers` リソースの一部としてみなします。
- `/customers` と `/customers/id/addresses` とは、`/customers/id/addresses/{addr}` が存在し住所の識別子を追加でもつので、別々のリソース型とみなします。
- `/addresses` と `/customers/id/addresses` は、 それらが同一のものであると確信もって言えるすべがないので、別々のリソース型とみなします。

この定義にしたがうと、経験的にリソースのタイプは4〜8より多くなることはないと思います。 より多くのリソースを必要とする複雑な業務ドメインでは例外があるかもしれませんが、 その際はAPIを分類することによって、サブドメインに分割できないかをまず検討するべきです。

そうはいっても1つのAPIは、利用者が業務フローを理解できるように完全な業務プロセスをモデル化し、 必要なリソースすべてを揃えたものであるべきなのは、お忘れなく。

SHOULD サブリソースのレベルの深さを制限する

(ルートからのURLパスをもつ)メインリソースと(非ルートのURLで表される)サブリソースが 存在します。対象のリソースのライフサイクルが、メインリソースと(疎に)結びついていれば、 サブリソースを使います。つまりメインリソースは、サブリソースエンティティの コレクションリソースの役割を担います。 サブリソースの(ネストした)レベルは3以下にすべきです。 それ以上になるとAPIの複雑性は増し、URLパスも長くなりすぎてしまうからです。 (ふつうのWebブラウザは2000文字以上のURLをサポートしないことを忘れずに)

11. HTTPリクエスト

MUST HTTPメソッドを正しく使う

以下に示すように、標準のHTTPメソッドの意味に沿うようにしましょう。

GET

`GET`リクエストは、単一のリソースの読み込み、またはリソースの集合のクエリのために使用されます。

- 個々の`GET`リクエストは、リソースが存在しなければ通常404となる。

- リソースのコレクションへのGETのリクエストは、(リストが空であれば) 200を、 (リスト自体が存在しなければ) 404が返る。
- GETリクエストはボディのペイロードをもってはいけない。(GET With Body参照)

注意: リソースのコレクションへのGETリクエストは、フィルタやページネーションの機能を十分に提供するべきです。

"ボディ付きのGET"

GETでの構造をもつリクエストは、クライアントやロードバランサ、サーバのサイズ制限に引っかかることがあり、APIでもときどきこの問題に直面します。 私たちはAPIが標準に準拠する(すなわちサーバサイドでGETのボディは無視されなくてはならない)よう要求するので、API設計者は次の2つのうち何れかを選択しなければなりません。

1. URLエンコードされたクエリパラメータ付きのGET: クライアント、ゲートウェイ、サーバの通常のサイズ制限を守りつつ クエリパラメータにリクエスト情報をエンコードできるのであれば、 これが第1の選択肢です。リクエスト情報は、複数のクエリパラメータ分散してもたせてもよいし、 単一のパラメータにURLエンコードしてもたせてもかまいません。
2. ボディコンテンツ付きのPOST: URLエンコードされたクエリパラメータ付きのGETが どうしても制限に引っかかる場合は、ボディコンテンツ付きのPOSTを使わねばなりません。 この場合、エンドポイントは`GET With Body`ヒントを必ずドキュメントに付けて、 GETの意味での呼び出しであることを伝えなければなりません。

注意: ヘッダに構造化されたリクエスト情報をエンコードすることは選択肢にはなりません。 コンセプト上の観点から、常にリソース名とクエリパラメータ(つまりURLになるもの)で 操作の意味を表さなくてはなりません。 リクエストヘッダは、例えばFlowIDのような、一般的な文脈情報のために予約されています。 おまけにクエリパラメータとヘッダのサイズ上限には、これで決まりというものはなく、 クライアント、ゲートウェイ、サーバの設定に依存したものです。 だから、ヘッダに切り替えたからといって何も問題は解決しないのです。

ヒント: GET With Bodyは拡張クエリパラメータが使われるので、cursorはもはやカーソルベースページネーションの場合に、クエリフィルタをエンコードするのには使われません。結果として、クエリフィルタはボディに入れて送るのがベストプラクティスになります。cursorに適用されたクエリフィルタを全体のハッシュ値を含ませるようにすると、ページネーションの順番を保つことができます(**SHOULD** 適用可能なところではページネーションリンクを使う参照)。

PUT

PUTのリクエストは、リソース **全体** の **更新** (稀に作成)に使われます。 単一のリソース、リソースのコレクション両方が対象です。PUTリクエストは、 *"URLが表すリソースを、このオブジェクトで既存のリソースと置き換えてください"* という意味になります。

- PUTリクエストは通常はコレクションでなく単一リソースに適用されるものです。 コレクションに対するPUTは、その全体を置き換えることを暗に意味するからです。
- PUTリクエストは更新前に、暗黙的にリソースの作成をおこなうことによって、 存在しないリソースに対しても問題を発生しないようにできます。

- PUTリクエストが成功したら、URLによって表現されるリソース **全体** が更新されます (後続の読み取りにも同じペイロードが返される)。
- PUTリクエストが成功したら、(更新オブジェクトを返すならば) 200を (何も返さないならば) 204を、 (リソースが新規に生成された場合は) 201をステータスコードとして返す。

重要: (少なくともトップレベルの)リソースの生成については、PUTよりもPOSTを使うのがベストプラクティスです。そうすることで、リソースIDを残すことができるし、次に示すようなPUTを使った更新セマンティクスに集中できるようになります。

注意: PUTがリソースの生成に使われるのは稀なケースなので、リソースIDはクライアントが保持し、URLパスセグメントで受け渡しします。 同一のリソースに2度PUTしても、冪等である必要があり同じ結果を返さなくてはなりません(**MUST** メソッド毎に共通の性質を満たす参照)。

ヒント: PUTを使うときに意図せず同時更新してしまうことを防ぐため、`ETag` と `If-(None-)Match`ヘッダの 組み合わせで、コンフリクトを表明し変更を失わないようにするために、サーバに厳密な要求を送るようにしましょう。 RESTful APIにおける楽観ロックセクションでもこのアプローチの代替案を記述しています。

POST

POSTは慣例的には、リソースのコレクションのエンドポイントに、単一のリソースを作成するのに使われますが、単一リソースエンドポイントでも他のものと同じように使えます。コレクションのエンドポイントにとっての意味は"*URLによって識別されるリソースのコレクションにオブジェクトを追加してください*"というものになります。

- POSTリクエストが成功すると、サーバは1つまたは複数の新しいリクエストを生成し、レスポンスにそれらのURI/URLを含みます。
- 成功したPOSTリクエストは、(もしリソースが更新されたら) 200を、(もしリソースが生成されたのであれば) 201を、(もしリクエストが受け付けられたがまだ終了してないならば) 202を、例外的に(もし本当のリソースを返さないならば) `Location`ヘッダを付けた204を返すのが通常です。

単一のリソースエンドポイントに対しての意味は、 *"URLで識別されるリソースに与えられたリクエストを実行してください"* というものになります。

より一般的に: POSTは、他のHTTPメソッドだと十分でないシナリオのためにも使われるべきです。そのような場合には、POSTがワークアラウンドとして使われる事実をドキュメント化するようにしましょう。(`GET With Body`参照)

注意: POSTリクエストと関連したリソースIDは、サーバで作成・管理され、レスポンスのペイロードでクライアントに返されます。

ヒント: 同じリソースを2回POSTする際、冪等は必要では **ありません** (**MUST** メソッド毎に共通の性質を満たすをチェックしてみよう)。でも、これを防ぐために**SHOULD** POST と PATCH の冪等設計を検討するしましょう。

PATCH

PATCHリクエストは、単一リソースの部分更新にのみ使われます。つまりリソースフィールドの 特定のサブ

セットのみが置き換わります。 そのリクエストは、 "
この変更リクエストに対応するURLで特定されるリソースを変更してください" という意味になります。 変更リクエストの意味は、HTTP標準では定義されていないので、適したメディアタイプを使い API仕様に記述しなければなりません。

- PATCHリクエストは、通常単一リソースに適用される。
- PATCHリクエストはリソースインスタンスが存在しないものに対しては、通常安定的ではない。
- PATCHリクエストが成功したら、ペイロード中の変更リクエストに定義されているとおりに、 サーバはURLによって指し示されたリソースを更新するだろう。
- PATCHリクエストが成功したら通常、(更新されたコンテンツを含むならば) 200 を (何も返さないならば) 204のステータスコードを返します。

注意: PATCHを正しく実装するのは些かトリッキーなので、 後方互換性ある変更 がされる限り、 私たちはエンドポイントにつき次のパターンのどれか1つを選択するよう強く推奨します。 好ましい順に並べると:

1. リソースの更新にはオブジェクトまるごと全体を渡すPUTを使う (つまりPATCHを一切使わない)
2. リソースの一部を更新するためだけに、部分的なオブジェクトでPATCHを使う (これは JSON Merge Patch であり、部分的なリソース表現であることを示すために `application/merge-patch+json` のメディアタイプを使う)
3. JSON Patch で規定されたPATCHを使う。 専用のメディアタイプ `application/json-patch+json` は、リソース変更の方法を 指示していることを表す。
4. メディアタイプで定義された手段で、リクエストがリソースを変更しない場合は、 PATCHの代わりに (何が起きたかの正しい記述がされた) POSTを使う。

特に JSON Merge Patch は、 特に(リソースの一部として) 巨大なコレクションの中の1つのオブジェクトを更新しようとすると、 すぐに限界を感じることでしょう。 この場合、 JSON Patch が可読性のあるPATCHリクエストである限りは有効な手段です。 (JSON patch vs. merge をみてください)。

注意: 同じリソースに対して2回パッチすることは、冪等である必要は **ありません** (**MUST** メソッド毎に共通の性質を満たすをチェックしてみましょう)。でも、これを防ぐために**SHOULD** POST と PATCH の冪等設計を検討するしましょう。

ヒント: {PACTH}を使うとき、気付かずに同時更新してしまうのを防ぐために、 **MAY** If-Match/If-None-Match ヘッダともにEtagのサポートを検討しようで、サーバがコンフリクトを避け、変更内容がロストしないようできるます。RESTful APIにおける楽観ロックと**SHOULD** POST と PATCH の冪等設計を検討するにより詳細と選択肢があります。

DELETE

DELETEリクエストはリソースの削除に使われ、 "*URLによって特定されるリソースを削除してください*" ということを意味します。

- DELETEリクエストは、通常単一リソースに適用される。コレクションリソースに対するDELETEは、 コレクションまるごと削除することを暗に示しているので、あまり使われない。

- `DELETE`リクエストが成功したら通常、(削除されたリソースを返すならば) 200を、(何も返さないならば) 204のステータスコードを使う。
- DELETEリクエストが失敗したら通常、(リソースが見つからない場合は) 404を、(リソースが既に削除済みならば) 410のステータスコードを使う。

重要: `DELETE`でリソースを削除した後のそのリソースに対する`GET`リクエストは、削除後にリソースがどう表現されるかによって、404 (not found)と410 (gone)のどちらかを返すことが期待されます。この操作の後、リソースがそのエンドポイントでアクセス可能である必要はありません。

HEAD

`HEAD`リクエストは、単一のリソースまたはリソースのコレクションについてのヘッダ情報だけを取得するのに使われます。

- `HEAD`は`GET`と正確に同じ意味を持ちますが、ボディは返されず、ヘッダのみが返されます。

ヒント: `HEAD`は特に、`ETag`ヘッダとともに、大きなリソースやコレクションリソースが更新されたかどうかを効率的に確認するのに使われます。

OPTIONS

`OPTIONS`リクエストは、与えられたエンドポイントの利用可能な操作(HTTPメソッド)が何かを調べるのに使われます。

- `OPTIONS`レスポンスは通常、利用可能なメソッドをカンマ繋ぎにしたものを(`Allow:`-ヘッダで)返すか、リンクテンプレートのリストとして返されます。

注意: `OPTIONS`を実装することはあまりありませんが、リソースの全機能を示すのに使われます。

MUST メソッド毎に共通の性質を満たす

RESTfulサービスにおけるリクエストメソッドは…

- safe - リードオンリーで定義された操作は、 *意図的な副作用* を持ってはならない。すなわちサーバの状態を変更してはならない。
- idempotent - その操作が一回のみの実行でも、複数回の実行でも、サーバの状態に同じ *意図した効果* しかもたらさない。 **注意:** これは操作が同じレスポンスまたはステータスコードを返すことまでは要求しない。
- cacheable - レスポンスを将来の再利用のために保管できることを示す。一般に安全なメソッドへの要求は、サーバからの現在のレスポンスまたは権限のレスポンスを必要としない場合、キャッシュ可能となる。

注意: 上記の定義で_意図された(副)作用_ により、サーバはロギング、アカウンティング、プリフェッチなどの追加の状態変更する振る舞いを提供します。ただし、これらの実際の作用と状態変更が、その操作によって意図されたものであってはなりません。

メソッド実装は、 RFC 7231 にしたがい、次の基本的な性質を満たさなければなりません。

メソッド	安全性	冪等性	キャッシュ可能性
GET	☒ Yes	☒ Yes	☒ Yes
HEAD	☒ Yes	☒ Yes	☒ Yes
POST	☒ No	⚠️ NoだがSHOULD <code>POST</code> と <code>PATCH</code> の冪等設計を検討すると実現可能	⚠️ もし特定の<code>POST</code>エンドポイントが安全であれば可能かもしれない。 Hint: 大抵のキャッシュではサポートされない。
PUT	☒ No	☒ Yes	☒ No
PATCH	☒ No	⚠️ NoだがSHOULD <code>POST</code> と <code>PATCH</code> の冪等設計を検討すると実現可能	☒ No
DELETE	☒ No	☒ Yes	☒ No
OPTIONS	☒ Yes	☒ Yes	☒ No
TRACE	☒ Yes	☒ Yes	☒ No

注意: **MUST** キャッシュ可能な GET, HEAD, POST エンドポイントをドキュメント化する

SHOULD POST と PATCH の冪等設計を検討する

多くの場合で、例えば同じリソースが並行して作成または変更されたり、複数回にわたって変更されたりする可能性がある場合など、クライアントが競合を明らかにし、リソースの重複(いわゆるゾンビリソース)や、更新内容の消失を防ぐために、POSTとPATCHの冪等性を設計することが役立ちます。冪等 APIエンドポイントを設計するには、次の3つのパターンのいずれかを適用することを検討する必要があります。

- リクエストにIf-Match ヘッダを介してリソース固有の **条件付きキー** を与える。キーは一般にリソースのメタ情報です。たとえば、*hash* や *バージョン番号* 等で、よく一緒に保存されます。冪等な挙動を保証するために、同時に発生する生成と更新を検出できるようになります。(**MAY** If-Match/If-None-MatchヘッダともにEtagのサポートを検討しよう参照)

- リソース固有の **セカンダリキー** は、リクエストボディ中のリソースプロパティとして提供されます。 *セカンダリキー* はリソースに永続的に保存され、異なるクライアントから複数のリソース生成要求が発生する場合に、一意なセカンダリキーを探すことによって、冪等 な振る舞いを保証できるようになります。(**SHOULD** 冪等な POST 設計のためにセカンダリキーを使う参照)
- クライアント固有の **冪等キー** は、リクエストの`Idempotency-Key`ヘッダを介して与えられます。キーはリソースの一部ではありませんが、リクエストをリトライする際の冪等な挙動を保証するために、元のレスポンスを指し示すために一時的に保存されます。(**MAY** `Idempotency-Key` ヘッダのサポートを検討しよう参照)

注意: 条件付きキー と **セカンダリキー** は、同時リクエストを扱うのに注力していて、 **冪等キー** は上で定義された冪等性よりも *より強い* 要求となる正確に同じレスポンスを返すということに注力しています。したがって他の2つと組み合わせることができます。

あなたのユースケースにどのパターンが適しているかを決めるために、各パターンの主要な判断軸を示した以下の表をよく見てください。

	条件付きキー	セカンダリキー	冪等キー
Applicable with	PATCH	POST	POST/PATCH
HTTP Standard	☒ Yes	☒ No	☒ No
Prevents duplicate (zombie) resources	☒ Yes	☒ Yes	☒ No
Prevents concurrent lost updates	☒ Yes	☒ No	☒ No
Supports safe retries	☒ Yes	☒ Yes	☒ Yes
Supports exact same response	☒ No	☒ No	☒ Yes
Can be inspected (by intermediaries)	☒ Yes	☒ No	☒ Yes
Usable without previous GET	☒ No	☒ Yes	☒ Yes

注意: PATCHに適用可能なパターンは、同じプロパティを提供するPUTおよびDELETEに同じ方法で適用できます。

安全なリトライをサポートすることを主目的するならば、条件付きキーとセカンダリキー パターンを冪等キーパターンの前に適用します。

SHOULD 冪等な POST 設計のためにセカンダリキーを使う

生成時のPOST 冪等を設計するための最も重要なパターンは、重複リソースの問題(いわゆるゾンビリソース)を無くすために、リクエストボディにリソース固有の **セカンダリキー** を導入します。

セカンダリキーは、リソースに *代替キー* または(もし複数のプロパティからなるのであれば) *複合キー* として永続的に保存され、サーバサイドで実行される一意制約によって上記問題からガードします。 最良かつ自然に存在する候補は、新しく生成されたリソースと *1対1* の関連をもつ別のリソース(親プロセス識別子など)を指す *一意の外部キー* です。

セカンダリーキーの例として良い例は、注文リソースにおけるショッピングカートIDです。

注意: `Idempotency-Key`無しにセカンダリキーパターンを使うときは、全ての一連のリトライはコード409 (conflict)で失敗すべきです。リソースが明確に定義された振る舞いを実装する元のリソースであるという確信がないのであれば、200を使うのは避けたほうがよいでしょう。コンテンツなしで204を使うことも、同様に適切に定義された選択肢です。

SHOULD ヘッダとクエリパラメータのコレクションフォーマットを定義する

カンマで区切られた値のリストか、パラメータを複数回繰り返すかのどちらかで、ヘッダとクエリパラメータで値の集合を渡すことができます。

Parameter Type	Comma-separated Values	Multiple Parameters	Standard
Header	`Header: value1,value2`	`Header: value1, Header: value2`	RFC 7230 Section 3.2.2
Query	`?param=value1,value2`	`?param=value1&param=value2`	RFC 6570 Section 3.2.8

Open APIでは一度に両方のスキーマをサポートできないので、API仕様ではどちらかを明示的に定義しなければなりません。

Parameter Type	Comma-separated Values	Multiple Parameters
Header	`style: simple, explode: false`	not allowed (see RFC 7230 Section 3.2.2)
Query	`style: form, explode: false`	`style: form, explode: true`

コレクションフォーマットを選択する際には、ツールのサポート、特殊文字のエスケープ、URLの最大長を超えないかに注意してください。

SHOULD クエリパラメータを使ったシンプルなクエリ言語を設計する

クエリパラメータを使って、大部分のAPIのリソース固有のクエリ言語を記述することを推奨します。これはクエリパラメータがHTTPネイティブであり、拡張が容易で、HTTPクライアントおよびWebフレームワークで優れた実装サポートがあるためです。

クエリパラメータは、次の観点をもつべきです。

- 対応するプロパティの参照 (存在する場合)
- 値の範囲。例えば境界を含む、含まない。

- 比較のセマンティクス (equals, less than, greater than など)
- 他のクエリと組み合わせたときの影響。例えば *and* なのか *or* なのか

クエリパラメータがどのように命名され、どのように使われるかは、個々のAPI設計者次第です。次の例を参考にしてください。

- `name=Zalando`, プロパティの等価性にもとづく要素のクエリ
- `age=5`, 論理的なプロパティにもとづく、要素のクエリ
    - `age` という要素は、存在せず `birthday` のみ持っているケースを想定。
- `max_length=5`, 上限、下限にもとづく (`min` と `max`)
- `shorter_than=5`, 特定の用語を使う。例えば *length*
- `created_before=2019-07-17` や `not_modified_since=2019-07-17`
    - 特定の用語を使う。例えば: *before*, *after*, *since*, *until*

私たちは、特定の名前を支持したり反対したりはしません。最終的にAPIはそのドメインに最も適した用語を自由に選択すべきだからです。

SHOULD JSONを使った複雑なクエリ言語を設計する

<236, query parameters>>に基づく最小のクエリ言語は、単一の方法のみで結合される少ないフィルタ(例えば *and* セマンティクス) を使う単純なユースケースに適しています。一般的にもシンプルなクエリ言語の方が、複雑なものよりも好まれます。

APIによってはより複雑なクエリ言語が必要なこともあります。代表的な例は検索(ファセット検索を含む)APIと、製品カタログAPIです。

これらのAPIは他のAPIと以下の点で異なります。

- 尋常でない多くの利用可能なフィルタを使う
- 動的で拡張可能なリソースモデルのための動的フィルタ
- 演算子の自由な選択 例えば `and`, `or` , `not`

特定の複雑なクエリ言語に合うAPIは、ネストしたJSONデータ構造を使い、Open APIを使って直接定義するのがよいでしょう。これには以下のメリットがあります。

- クライアントがデータ構造を扱うのが簡単である。
    - 特別なライブラリのサポートが必要ない
    - 文字列結合や手動エスケープが必要ない
- サーバがデータ構造を扱うのが簡単である。
    - 特別なトークナイザが必要ない。

    - セマンティクスはテキストトークンよりもデータ構造にある。
- 他のHTTPメソッドでも構成できる。
- APIはOpen APIで完全に定義しきれる。
    - 外部ドキュメントや文法が必要ない。
    - 既存の手段は誰もが知っている。

JSON-specific rules と、おそらくきっとGET-with-bodyパターンを使うことになるだろう。

例

次のJSONドキュメントは、構造化クエリがどのようになるかを示したものです。

```
{
  "and": {
    "name": {
      "match": "Alice"
    },
    "age": {
      "or": {
        "range": {
          ">": 25,
          "<=": 50
        },
        "=": 65
      }
    }
  }
}
```

以下からもインスピレーションが得れるでしょう。

- Elastic Search: Query DSL
- GraphQL: Queries

MUST 暗黙的なフィルタリングをドキュメント化する

あるコレクションリソースやクエリが、持っている要素全てではなく、現在のクライアントにアクセスが許可されたものだけを返すことがあります。

次の場合に、暗黙的フィルタリングされます。

- 親のGETリクエストで返されるリソースのコレクション
- リソースの詳細で返されるフィールド

そのような場合、暗黙的なフィルタリングはAPI仕様として(そのdescriptionに)書かれなくてはなりません。

暗黙的なフィルタリングするときは、caching considerations も考えよう。

例:

会社 *Foo* の従業員が当社の企業間サービスの一つにGETアクセスする場合、法律上の理由から、会社が所有または契約管理していない他のビジネスパートナーを法事してはなりません。私たちが、会社 *Bar* と一緒にビジネスしていることは決してバレてはなりません。

Foo で動作するコンシューマからはレスポンスは次のように見える。

```
{
    "items": [
        { "name": "Foo Performance" },
        { "name": "Foo Sport" },
        { "name": "Foo Signature" }
    ]
}
```

Bar で動作するコンシューマからはレスポンスは次のように見える。

```
{
    "items": [
        { "name": "Bar Classics" },
        { "name": "Bar pour Elle" }
    ]
}
```

API仕様はこのように何を特定するかを示すべきです。

```
paths:
  /business-partner:
    get:
      description: >-
        Get the list of registered business partner.
        Only the business partners to which you have access to are
returned.
```

12. HTTPステータスコードとエラー

MUST 成功とエラーレスポンスを規定する

APIは機能、業務の観点で定義され、実装の観点からは切り離され抽象化しなければなりません。 成功と失敗のレスポンスは、APIが正しく使われるために必要不可欠な部分です。

だからAPI仕様においては、**すべての**成功とサービス固有のエラーレスポンスを定義しなければなりません。 両者ともインタフェース定義の一部であり、サービスクライアントが標準だけでなく例外も、 正しく扱うための重要な情報を提供するものです。

ヒント: たいていの場合、すべての技術的なエラー、特にサービスプロバイダに制御されないようなものを、ドキュメント化するのは役に立ちません。 レスポンスコードがアプリケーション固有の機能の意味を伝えないか、または 追加の説明を必要とするような標準でない使われ方をする限りは、 複数のエラーレスポンス仕様は、次のパターンを用いて組み合わせることができます。(**MUST** 永続的で不変であるリモート参照のみを使うも参照)

```
responses:
  ...
  default:
    description: error occurred - see status code and problem object for
more information.
    content:
      "application/problem+json":
        schema:
          $ref:
'https://opensource.zalando.com/problem/schema.yaml#/Problem'
```

API設計者は関連するオンラインAPIドキュメントの一部として、 **トラブルシューティングボード** について考えなければなりません。 API固有のエラーについての情報とハンドリングの指針を提供し、API仕様からリンクを通じて参照されるものにもなります。 これはサービスのサポート業務を減らし、サービス利用者、提供者双方の業績に貢献します。

MUST 標準のHTTPステータスコードを使う

標準のHTTPステータスコードのみを使い、その意味に沿うように一貫性をもった設計をしなければなりません。 どうかHTTPステータスコードを新たに発明しないようにしてください。

RFCの標準では ~60 の異なるHTTPステータスコードと同時にその意味も定義されていてます。 (主に RFC7231 と RFC-6585) — そして draft legally-restricted-status のように新しいものもあります。 すべてのエラーコードは Wikipedia か https://httpstatuses.com/ で '非公式なコード'(NginxのようなWebサーバで使われるもの)を含んだものを 見ることができます。

以下によく共通で使う(RFC標準と整合性ある)HTTPステータスコードを、理解の助けになるよう一覧にしました。 ここに載ってないHTTPステータスコードを使っても良いですが、 その場合、API定義に明示しなくてはなりません。

重要: ここに定義された意味でコードを使う限りは、そうする必要はありません。 一貫性のない定義をしてしまうリスクは低いし、常識をドキュメントに書きすぎると可読性が下がるからです。 HTTPステータスコード

がリストにない、または使うには追加の情報が必要とされるときだけ、API仕様にレスポンスのHTTPステータスコードの詳細を明記しましょう。

成功コード

Code	Meaning	Methods
200	OK - 標準の成功レスポンス	`<all>`
201	Created - エンティティが正常に作成されたこととを示す。空のレスポンスでも作成されたリソースを返してもよい。がそのリソースのURLをLocationヘッダにセットする。 (より詳細は 共通のヘッダ 参照) *常に* Locationヘッダをセットすること。	`POST`, `PUT`
202	Accepted - リクエストは成功し非同期で処理されている。	`POST`, `PUT`, `DELETE`, `PATCH`
204	No content - レスポンスボディがない。	`PUT`, `DELETE`, `PATCH`
207	Multi-Status - バッチ/バルクリクエストで、レスポンスボディは複数のステータスを含んでいる。**MUST** バッチリクエストやバルクリクエストには 207 を使う 参照	`POST`

リダイレクトのコード

Code	Meaning	Methods
301	Moved Permanently - 以後のリクエストはすべて与えられたURIに直接送るようにすべき。	`<all>`
303	See Other - GETメソッドを使って別のURIへリクエストを送ってくれ。	`PATCH`, `POST`, `PUT`, `DELETE`
304	Not Modified - If-Modified-Since や If-None-Match ヘッダで送られた日付やバージョンから、リソースは何も変更されていない。	`GET`

クライアントサイドのエラーコード

Code	Meaning	Methods
400	Bad request - 一般的な / 未知のエラー。入力のペイロードが業務ロジックバリデーションでエラーになったときにも送出される。	`<all>`
401	Unauthorized - ユーザはログインしなければならない。(“Unauthenticated”の意)	`<all>`
403	Forbidden - ユーザはこのリソースのアクセス権限がない。	`<all>`

Code	Meaning	Methods
404	Not found - リソースが見つからない。	`<all>`
405	Method Not Allowed - メソッドがサポートされていない。OPTIONSで調べることができる。	`<all>`
406	Not Acceptable - リクエストで送られたAcceptヘッダにしたがったレスポンスを返すことができない。	`<all>`
408	Request timeout - リソース待ちでサーバがタイムアウトした。	`<all>`
409	Conflict - リクエストは競合が発生したために完遂できなかった。例えば2つのクライアントが同じリソースを作成しようとしたり、同時に整合性の保てない更新要求が発生するようなケース。	`POST`, `PUT`, `DELETE`, `PATCH`
410	Gone - リソースがもう存在しない。例えば、意図して削除されたリソースにアクセスしたケース。	`<all>`
412	Precondition Failed - 条件に合わないリクエストがされた。例えばIf-Matchを満たさないケース。 楽観ロックで使われる。	`PUT`, `DELETE`, `PATCH`
415	Unsupported Media Type - 例えばクライアントがContent-Typeなしでリクエストボディを送っていたケース	`POST`, `PUT`, `DELETE`, `PATCH`
423	Locked - 悲観ロック。例えば、処理中。	`PUT`, `DELETE`, `PATCH`
428	Precondition Required - サーバは条件付きリクエストを要求する。例えば、更新が失われるのを避けるために。(**MAY** 処理するプリファレンスを示すために `Prefer`ヘッダのサポートを検討しよう参照)	`<all>`
429	Too many requests - クライアントが大量のリクエストを送ってきた。 **MUST** レート制限のためのヘッダと429コードを使う 参照。	`<all>`

サーバサイドのエラーコード

Code	Meaning	Methods
500	Internal Server Error - サーバで予期しないエラーが起きたことを示す。(クライアントのリトライは単純には行えない可能性があります)	`<all>`
501	Not Implemented - サーバはリクエストを実行できない (暗に将来実行可能になることを指す)。	`<all>`

Code	Meaning	Methods
503	Service Unavailable - サーバが(一時的に)利用できない (つまり高負荷のため)⍰—⍰クライアントのリトライは単純には行えない可能性があります。可能なら、サービスはクライアントにどれくらい待てば良いかを指示するために、`Retry-After`を設定します。	`<all>`

MUST もっとも状況にあったHTTPステータスコードを使う

処理結果やエラー状況を返すとき、もっとも適したHTTPステータスコードを使わねばなりません。

MUST バッチリクエストやバルクリクエストには 207 を使う

APIには性能上の理由から、つまり通信と処理を効率化する目的で、`POST`を使った *バッチ* または *バルク* リクエストを 提供する必要があります。 この場合、サービスはバッチまたはバルクリクエストの各パートに対応した複数のレスポンスコードを 通知する必要があるかもしれません。 HTTPはバッチ/バルクリクエストとレスポンスの扱いに関して、指針を示していないので、 私たちは次のようなアプローチを定義します。

- バッチ/バルクリクエストには、 **常に** ステータスコード207を返さなければならない。 ただし個々のパートを処理する前にエラーが発生した場合はその限りではない。
- バッチ/バルクレスポンスは、 **常に** バッチ/バルクリクエストの各パートに関する十分なステータスと モニタリング情報を含む、複数状態をもつオブジェクトを、ステータスコード207とともに返す。
- バッチ/バルクリクエストは、もしサービスが個々のパートを処理する前にエラーが発生したり、 予期しないエラーが発生した場合は、4xx/5xxのステータスコードを返すかもしれない。

すべてのパートで処理が *失敗* したり、各パートが *非同期に* 実行される *場合においても* このルールが適用されます! 一貫した方法で、クライアントがバッチ/バルクリクエストの個々の結果を精査しなくてはならないことを意図しています。

注意: *バッチ* とは独立した処理を起動するリクエストの集合であり、 *バルク* とは1つのリクエストで独立した作成または更新用リソースの集合である、 と定義しています。処理結果のレスポンスに関していえば、この違いはあまり重要では ありません。

MUST レート制限のためのヘッダと429コードを使う

クライアントのリクエストレートを管理したいAPIは、もしクライアントがリクエストレートを超過したら、429 (Too Many Requests)レスポンスコードを使わなければなりません (RFC 6585参照)。 そのようなレスポンスは、クライアントにそのような追加の情報を知らせるために、 ヘッダをセットしなくてはなりません。その手段は次の2つがあります。

- クライアントが次のリクエストを送るまで、どれくらい待てばよいかを指示するための、`Retry-After`ヘッダを返す。 Retry-Afterヘッダはリトライできるようになる日時をHTTP dateで表現したものか、 遅延秒数の何れかを含みます。どちらも許容されますが、APIでは遅延秒数を使うのを優先します。
- 'X-RateLimit' ヘッダトリオを返す。サーバは(後述する)これらのヘッダを使って、与えられたタイムウィンドウ内で 許容されるリクエストの数や、ウィンドウがいつリセットされるかの形式で、サービスレベ

ルを表現します。

'X-RateLimit' ヘッダには、以下のようなものがあります。

- `X-RateLimit-Limit`: クライアントがこのウィンドウ内で最大リクエストできる数
- `X-RateLimit-Remaining`: 現在のウィンドウでリクエストできる残数
- `X-RateaLimit-Reset`: レート制限ウィンドウがリセットされる秒数。 これはGitHubやTwitterの同名のヘッダとは異なり、UTCエポック秒数を返すことに **注意** します。

両方のアプローチを認めている理由は、APIごとに異なるニーズが存在するからです。 Retry-After は一般的な負荷やリクエストのスロットリングに関しては十分なものですが、 テナントや指定取引先のような対象毎にスロットを用意する場合においては適していません。 これによって、リソースオーナーはクライアントのリクエストに関して、管理しなくてはならない状態の数を最小化できるようになります。 一方、'X-RateLimit' ヘッダは、クライアントが既存の取引先やテナント毎にシナリオを用意するのに適しています。 'X-RateLimit' ヘッダは一般的に429のときだけでなく、すべてのリクエストに対して付与されます。 これはそのAPIを実装したサービス与えられたウィンドウで、各スロット対象毎にリクエストの数を 追跡できる能力があることを暗に示しています。

MUST Problem JSONを使う

RFC 7807 でProblem JSONオブジェクトと、 `application/problem+json` メディアタイプが定義されています。 処理中に発生したどんな問題も(適切なステータスコードとともに)これを使い、 クライアントサイドのエラー(4xx)か、サーバサイドのエラー(5xx)かに関わらず、 ステータスコードよりも詳細な情報を返すべきです。

Problem JSONオブジェクトのOpenAPIスキーマ定義は、 GitHub上 にあります。

これを使って以下のように定義できます。

```
responses:
  503:
    description: Service Unavailable
    content:
      "application/problem+json":
        schema:
          $ref:
'https://opensource.zalando.com/problem/schema.yaml#/Problem'
```

もしAPIが追加のエラー詳細情報を返す必要があれば、 Problem JSONの拡張としてカスタムの型を定義することもできます。

ヒント (後方互換性のために): このガイドラインの以前のバージョンでは(RFC 7807 が 公開される前だったので)、 `application/x.problem+json` のメディアタイプを返すようにしていました。 この変更前に定義されたAPIサーバは、 クライアントが送る`Accept`ヘッダとエラーレスポンスの`Content-Type`ヘッダの 対応に注意しなければなりません。 またそのようなAPIのクライアントは、両方のメディアタイプを受け付け可能でなければなりません。

MUST スタックトレースを外に見せないようにする

スタックトレースには、APIの一部だけでなく、クライアントが依存すべきでない実装の詳細が含まれます。さらにはスタックトレースは、パートナーやサードパーティが受け取ってはならない機微な情報を漏らしてしまう 可能性があるし、攻撃者に脆弱性についてのヒントを与えることにもなりかねません。

13. 性能

SHOULD 必要な帯域幅を減らし応答性を改善する

APIはクライアントの必要性に応じて、帯域幅を減らすための仕組みをサポートすべきです。 パブリックなインターネットやテレコミュニケーションネットワークのように、 大きなペイロードをもち高トラフィックなシナリオで使われる(かもしれない)APIに有効です。 低帯域での通信を余儀なくされるモバイルWebアプリのクライアントが使うAPIは、その典型例です。 (Zalandoは'モバイルファースト'な企業なので、この点は心にとどめておきましょう)

共通のテクニックは、

- リクエストとレスポンスのボディの圧縮(**SHOULD** `gzip` 圧縮を使う参照)
- リソース属性のサブセットを取得できるよう、フィールドフィルタをクエリに実装する (**SHOULD** フィルタリングによって部分的なレスポンスをサポートする。参照)
- `ETag`と`If-Match`/`If-None-Match`ヘッダを使って、変更のないリソースの再フェッチを避ける (**MAY** If-Match/If-None-MatchヘッダともにEtagのサポートを検討しよう参照)
- 巨大なデータコレクションへのインクリメンタルなアクセスのためのページネーション
- マスタデータのキャッシュ。すなわち滅多に変更のない、または一切変更されないリソース (**MUST** キャッシュ可能な `GET`, `HEAD`, `POST` エンドポイントをドキュメント化する参照)

それぞれの詳細は以下に示します。

SHOULD `gzip` 圧縮を使う

圧縮時間がボトルネックになるほど多くのリクエストを捌かなければならないなど、 特別な理由がない限りは、APIレスポンスのペイロードをgzipで圧縮しましょう。 そうすることでネットワークの転送も速くなるし、フロントエンドの応答性も向上します。

gzip圧縮がサーバペイロードのデフォルトの選択肢ではありますが、サーバは 圧縮しないペイロードもサポートするべきです。クライアントは`Accept-Encoding`リクエストヘッダを 通じてそれをコントロールできます。 RFC-7231 Section 5.3.4 も参照してください。 サーバもまた`Content-Encoding`ヘッダを通じて、gzip圧縮が使われていることを明示すべきです。

SHOULD フィルタリングによって部分的なレスポンスをサポートする。

ユースケースとペイロードサイズに応じて、返却するエンティティのフィールドの フィルタリングをサポートすることによって、必要とするネットワーク帯域を大いに減らこすとができるでしょう。 フィールドクエリパラメータを付けることで、クライアントは欲しいデータに応じて、 フィールドのサブセットを決めることができます。 例は Google AppEngine API's partial responseをみてください。

フィルタなし

```
GET http://api.example.org/users/123 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "cddd5e44-dae0-11e5-8c01-63ed66ab2da5",
  "name": "John Doe",
  "address": "1600 Pennsylvania Avenue Northwest, Washington, DC, United
States",
  "birthday": "1984-09-13",
  "friends": [{
    "id": "1fb43648-dae1-11e5-aa01-1fbc3abb1cd0",
    "name": "Jane Doe",
    "address": "1600 Pennsylvania Avenue Northwest, Washington, DC, United
States",
    "birthday": "1988-04-07"
  }]
}
```

フィルタあり

```
GET http://api.example.org/users/123?fields=(name,friends(name)) HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json

{
  "name": "John Doe",
  "friends": [ {
    "name": "Jane Doe"
  } ]
}
```

`fields`クエリパラメータは、レスポンスのオブジェクトで返されるfieldsを決定するものです。例えば、

`(name)` は `name` フィールドだけをもつ `users` オブジェクトを返します。また `(name,friends(name))` は、`name` とネストされた `name` フィールドだけもつ `friends` オブジェクトを返します。

Open APIは公式にはパラメータによって異なるオフジェクトを返すスキーマをサポートしていません。フィールドパラメータを定義するときは、次の説明書きを加えておくことをおすすめします。: エンドポイントは戻りのオブジェクトのフィールドのフィルタリングをサポートする。 Rule #157

`fields`の値の文法は、次の BNF で定義される

```
<fields>             ::= [ <negation> ] <fields_struct>
<fields_struct>      ::= "(" <field_items> ")"
<field_items>        ::= <field> [ "," <field_items> ]
<field>              ::= <field_name> | <fields_substruct>
<fields_substruct>   ::= <field_name> <fields_struct>
<field_name>         ::= <dash_letter_digit> [ <field_name> ]
<dash_letter_digit> ::= <dash> | <letter> | <digit>
<dash>               ::= "-" | "_"
<letter>             ::= "A" | ... | "Z" | "a" | ... | "z"
<digit>              ::= "0" | ... | "9"
<negation>           ::= "!"
```

注意: 驚き最小化の原則 にしたがい、デフォルト値を使って`fields`パラメータを定義すべきではありません。結果は直感に反するので、API利用者は混乱してしまうからです。

SHOULD サブリソースの任意の埋め込みを可能にする

関連するリソースを組み込むこと(*リソース展開* として知られる)は、リクエスト数を減らすためには すごくよい手段です。 クライアントが前もって必要な関連リソースを知っている場合は、クライアントからサーバに、 データをEagarにプリフェッチできるよう指示します。 これはサーバで最適化されるのか(例えば、データベースのJOIN)、 一般的な手段(例えば透過的にリソースを差し込むHTTPプロキシ)で実現されるのかは、実装次第です。

命名に関しては **MUST** クエリストリングの規約を使う を参照ください。例えば埋め込みリソース展開には "embed" を使います。 埋め込みクエリには、前述のフィルタリングと同様の BNF 文法を使うようにしてください。

サブリソースの埋め込みは、例えばある注文がそのサブリソース (/order/{orderId}/items) として注文品目をもつような場合には、以下のようにみえます。

```
GET /order/123?embed=(items) HTTP/1.1

{
  "id": "123",
  "_embedded": {
    "items": [
      {
        "position": 1,
        "sku": "1234-ABCD-7890",
        "price": {
          "amount": 71.99,
          "currency": "EUR"
        }
      }
    ]
  }
}
```

MUST キャッシュ可能な GET, HEAD, POST エンドポイントをドキュメント化する

キャッシュは多くのことを考慮しなければなりません。例えば、一般的なレスポンス情報のキャッシュ可能性や、SSLを使ったエンドポイントを保護するガイドライン、リソースの更新とキャッシュ無効化のルール、複数のAPI利用者の存在などがあります。結果として、キャッシュは最良でも複雑(一貫性の観点などから)、最悪の場合は逆に非効率的なものになります。

頻繁に使用され、そのためにレート制限されたマスタデータサービス、すなわち、作成後にほとんどあるいは全く更新されないデータなど、サービスがそれ自体を保護することをサポートしない限り、クライアントサイドでの、また透過的なWebキャッシュを使うことは避けるべきです。

デフォルトでは、API提供者と利用者は常に`Cache-Control`ヘッダを`Cache-Control: no-store`にセットすべきであり、もし`Cache-Control`が設定されていなければ、同じく`Cache-Control: no-store`がセットされたものとして扱うべきです。

注意: このデフォルト設定をドキュメント化する必要はありません。ただし、フレームワークがデフォルトでこのヘッダの値を付加していることを確認するか、あるいは手動で(例えば下記のようなSping Securityのベストプラクティスを使って) これを確認してください。このデフォルトから外れた設定は、十分なドキュメント化が必要です。

```
Cache-Control: no-cache, no-store, must-revalidate, max-age=0
```

もしサービスが本当にキャッシュのサポートを必要とするなら、以下のルールにしたがってください。

- キャッシュ可能な`GET`, `HEAD`, `POST`のエンドポイントはすべて、レスポンスに`Cache-Control`, `Vary`, `ETag`ヘッダのサポートを宣言することによってドキュメント化する。 **注意:** `Expires`ヘッダは、キャッ

シュ生存期間の冗長で曖昧な定義を避けるため、定義してはならない。これらのヘッダのデフォルト文書を以下に示す。

- `Cache-Control`と`Vary`を使って正しくキャッシュ境界、すなわち生存期間やキャッシュ制約を定義し、キャッシュのサポートを明記しよう。以下でベストプラクティスを示す。
- キャッシュをウォームアップし、更新するのに効率的な方法を提供する。つまり、以下のようなものである。
    - 一般的には、`If-Match` / `If-None-Match` ヘッダと一緒に `ETag` を全ての キャッシュ可能なエンドポイントでサポートする。
    - 巨大なデータには、`HEAD`リクエストや`If-None-Match`ヘッダとともに`GET`を使う効率的なリクエストで、更新のチェックをする。
    - 小さなデータセットでは、`ETag`をサポートした`GET`リクエストを提供し、{If-Non-Match}付きの`HEAD`リクエストや`GET`リクエストで更新のチェックをする。
    - 中くらいのサイズのデータセットは、ページネーションとともに`ETag`をサポートする`GET`リクエストと、与えられた`<entity-tag>`以降の変更に対してレスポンスを制限する`GET`リクエストをフィルタリングするための`<entity-tag>`を使う。 **注意:** これは一般的なクライアントやHTTPレイヤでのプロキシキャッシュではサポートされない。

ヒント: キャッシュを適切にサポートするために、失敗した`HEAD`リクエストまたは`GET`リクエストでは、304ではなく`If-None-Match: <entity-tag>`を指定したコンテンツなしの412を返す必要があります。

```
components:
  headers:
  - Cache-Control:
      description: |
        The RFC 7234 Cache-Control header field is providing directives to
        control how proxies and clients are allowed to cache responses
results
        for performance. Clients and proxies are free to not support
caching of
        results, however if they do, they must obey all directives
mentioned in
        [RFC-7234 Section 5.2.2](https://tools.ietf.org/html/rfc7234) to
the
        word.

        In case of caching, the directive provides the scope of the cache
        entry, i.e. only for the original user (private) or shared between
all
        users (public), the lifetime of the cache entry in seconds (max-
age),
        and the strategy how to handle a stale cache entry (must-
revalidate).
        Please note, that the lifetime and validation directives for
shared
        caches are different (s-maxage, proxy-revalidate).

      type: string
      required: false
      example: "private, must-revalidate, max-age=300"

  - Vary:
      description: |
        The RFC 7231 Vary header field in a response defines which parts
of
        a request message, aside the target URL and HTTP method, might
have
        influenced the response. A client or proxy cache must respect this
        information, to ensure that it delivers the correct cache entry
(see
        [RFC-7231 Section
        7.1.4](https://tools.ietf.org/html/rfc7231#section-7.1.4)).

      type: string
      required: false
      example: "accept-encoding, accept-language"
```

ヒント: ETagソースには**MAY** If-Match/If-None-Matchヘッダともに**Etag**のサポートを検討しようを参照してください。

`Cache-Control`のためのデフォルト設定は、標準のOAuth認証を持つエンドポイントに対する `private` ディレクティブと、クライアントが古いキャッシュエントリを使わないようにするための `must-revalidate` ディレクティブが含まれている必要があります。 最後に `max-age` ディレクティブは、マスタデータの変更率とクライアントの一貫性を保つための 要件に応じて、数秒(`max-age=60`)から数時間(`max=age86400`)の間の値に設定する必要があります。

```
Cache-Control: private, must-revalidate, max-age=300
```

`Vary`のデフォルト設定を正しく決めるのはもっと難しいことです。 APIエンドポイントに大きく依存します。例えば、圧縮をサポートするかどうか、 異なるメディアタイプを受け付けるかどうか、他のリクエスト固有のヘッダを 必要とするかどうか、など。正しいキャッシュをサポートするには、この値を 注意深く選択してください。出発点となるデフォルト値は、おそらく以下のものでしょう。

```
Vary: accept, accept-encoding
```

いずれにせよ、これはクライアントに一般的なHTTPレイヤクライアントとプロキシキャッシュを クライアントが使おうとするときのみ、これが関係してきます。

注意: HTTPレベルでの一般的なクライアントとプロキシキャッシュは、設定が難しいです。 それゆえに、(おそらく分散)キャッシュを、アプリケーションのサービス(またはゲートウェイ)レイヤ に直接用意することを強く推奨します。`Vary`ヘッダの解釈から解放され、`Cache-Control`と `ETag`ヘッダの理解も非常に単純になります。さらにはキャッシュパフォーマンスやオーバーヘッドに ついても非常に効率的にもなるし、高度なキャッシュ更新とウォームアップのパターン.もサポートできるようになります。

いずれにせよ、どんなクライアントキャッシュやプロキシキャッシュを導入する前には、 RFC 7234 を注意深く読んでください。

14. ページネーション

MUST ページネーションをサポートする

リストデータへのアクセスは、クライアントサイドの一括処理と繰り返し操作のために、ページネーションをサポートしなければなりません。これは数百エントリ以上の(になる可能性のある)リストすべてにあてはまります。

2つのページネーションのテクニックがあります。

- Limit/Offsetベース: 最初のページエントリをオフセット数値で特定する
- cursor/limitベース — またの名をキーベース — ページネーション: 単一のキー要素で最初のページエントリを特定する (Facebookのガイドも見てください)

ページネーションの技術的概念は、問題がユーザエクスペリエンスと結びついていることも考慮しなければなりません。 この 記事 で述べられているとおり、 特定のページへのジャンプは、「前へ」「次へ」のページリンク(**SHOULD** 適用可能なところではページネーションリンクを使う参照)よりもあまり使われることはありま

せん。
それがオフセットベースのページネーションよりも、カーソルベースのページネーションを指向したい理由です。

注意: ページネーションの一貫したルックアンドフィールを提供するため、**MUST** クエリストリングの規約を使うで定義された共通のクエリパラメータ名を使わなければなりません。

SHOULD オフセットベースのページネーションを避け、カーソルベースのページネーションを使う

カーソルベースのページネーションは、オフセットベースのページネーションと比較すると、 いい感じでより効率的です。 データ量が多くなってきた時やNoSQLデータベースのストレージでは特に顕著です。

カーソルベースのページネーションを選択する前に、次のトレードオフを検討しておきましょう。

- 使い勝手とフレームワークのサポート
    - オフセットベースのページネーションはカーソルベースよりもよく知られており、フレームワークがサポートしていたり、APIクライアントで簡単に使えたりする
- ユースケース: とあるページへジャンプする
    - (100ページ中の51ページのように) 特定のページにジャンプするようなユースケースは、カーソルベースでは実現できない
- データの変更は結果セットのページに異常を引き起こす可能性がある
    - オフセットベースのページネーションは、ページ遷移の間に更新や削除がされると、結果の重複やロストを引き起こす可能性がある。
    - カーソルベースのページネーションを使うときは、2つのページを取得する間にカーソルの指し示すエンティティの削除がおこなわれると、ページングを継続することはできない。
- パフォーマンスの考慮 - オフセットベースのページネーションを使ったサーバ処理は効率的に実行するのが難しい
    - データベースのメインメモリにデータが存在しない場合は特に、コストの高い処理になる。
    - 共有データベースかNoSQLか?
- カーソルベースのナビゲーションは、結果の総件数が必要だったり、後方へのページネーションをサポートする必要がある場合には実現できないかもしれません。

ページネーションのために使われる`cursor`は、ページへのOpaqueポインタで、クライアントが **検査** したり **構築** したりしてはならない。通常は安全にコレクションを再作成できるように、ページの位置、すなわち、最初または最後のページ要素の識別子、ページネーションの方向、適用されたクエリフィルタを(暗号化)エンコードしたものです。`cursor`は次のように定義されます。

```yaml
Cursor:
  type: object
  properties:
    position:
      description: >
        Object containing the identifier(s) pointing to the entity that is
        defining the collection resource page - normally the position is
        represented by the first or the last page element.
      type: object
      properties: ...

    direction:
      description: >
        The pagination direction that is defining which elements to choose
        from the collection resource starting from the page position.
      type: string
      enum: [ ASC, DESC ]

    query:
      description: >
        Object containing the query filters applied to create the
collection
        resource that is represented by this cursor.
      type: object
      properties: ...

    query_hash:
      description: >
        Stable hash calculated over all query filters applied to create
the
        collection resource that is represented by this cursor.
      type: string

  required:
    - position
    - direction
```

カーソルベースのページネーションのためのページ情報は、cursorの集合で構成する必要があります。nextに加えて、prev,first,last,selfを次のようにサポートすることもあります。(リンク関連フィールド参照)

```
{
  "cursors": {
    "self": "...",
    "first": "...",
    "prev": "...",
    "next": "...",
    "last": "..."
  },
  "items": [... ]
}
```

注意: `cursor`集合のサポートは**SHOULD** 適用可能なところではページネーションリンクを使うが好まれて、削除されるかもしれません。

さらには以下の文書もあります:

- Twitter
- Use the Index, Luke
- Paging in PostgreSQL

SHOULD 適用可能なところではページネーションリンクを使う

クライアント設計を単純にするために、APIはコレクションのページネーションに適用できるときはいつでもsimplified hypertext controlsをサポートするべきです。`next`に加えて、`prev`, `first`, `last`, `self`のサポートを含むこともあります。(詳細はリンク関連フィールド参照)

このページ内容は`items`を通じて送信さる一方、`query`オブジェクトは次のように適用されたクエリフィルタも含むかもしれません。

```
{
  "self": "http://my-service.zalandoapis.com/resources?cursor=<self-
position>",
  "first": "http://my-service.zalandoapis.com/resources?cursor=<first-
position>",
  "prev": "http://my-service.zalandoapis.com/resources?cursor=<previous-
position>",
  "next": "http://my-service.zalandoapis.com/resources?cursor=<next-
position>",
  "last": "http://my-service.zalandoapis.com/resources?cursor=<last-
position>",
  "query": {
    "query-param-<1>": ...,
    "query-param-<n>": ...
  },
  "items": [...]
}
```

注意: 例えばGET With Bodyが必要なときのような、複雑な検索リクエストでは、cursorはすべてのクエリフィルタをエンコードできないかもしれない。この場合、ページ位置とcursorの方向だけをエンコードし、ボディでクエリフィルタを送るのがベストプラクティスでしょう。レスポンスも同様です。ページネーションの順序を守るため、この場合、cursorが適用されたすべてのクエリフィルタのハッシュを含ませ、使う前に検証するのがおすすめです。

注目: 必要性がない限り、トータル数を与えるのを避けるべきです。多くの場合、全件カウントをサポートすると、システムと性能に多大な影響が出ます。特にデータセットが増大し、リクエストが複雑になった場合は、フィルタによってフルスキャンが実行されます。これはAPIに関連した実装の詳細ですが、サービスの生存よりも、カウント機能が重要なのかよく考えましょう。

15. ハイパーメディア

MUST REST成熟モデル2を使う

私たちは REST 成熟度レベル2 のイケてる実装を目指します。 そうすることでHTTP動詞とステータスコードをフル活用した、リソース指向APIを構築できるようになるからです。 これらのガイドラインを通じて、これは多くのルールによって表現されています。

- **MUST** アクションを避ける — リソースについて考える
- **MUST** URLに動詞を入れない
- **MUST** HTTPメソッドを正しく使う
- **MUST** 標準のHTTPステータスコードを使う

これはHATEOASではありませんが、以下に示すルールで記述されるように、 APIに正しいリンク関連を設計することにとらわれるべきではありません。

MAY REST成熟モデル3を使う - HATEOAS

私たちは基本的には、 REST 成熟レベル3 を実装すことはおすすめしません。 HATEOASは、クライアントとサーバのREST APIを通じたやりとりしたり、 私たちのeコマースのSaaSプラットフォームの一部として複雑な業務機能を提供したりする、 私たちのSOA文脈においては、価値のない複雑さをAPIにもたらします。

私たち主な関心は、HATEOASのもたらす利点にあります。 (詳細な議論は RESTistential Crisis over Hypermedia APIs や Why I Hate HATEOAS 、を参照ください)

- 私たちは標準仕様言語でコード外に明示的にAPIを定義して、APIファーストの原則にしたがいます。 HATEOASはSOAクライアントエンジニアにとって、APIの自己記述性にはあまり価値を感じません: クライアントエンジニアは、APIリファレンス定義に、(リソースの状態に依存した)必要なリンクや使い方の記述を見ることができるのですから。
- 一般的なHATEOASクライアントは、APIについての前提知識を必要とせず、与えられたハイパーメディア情報に 基づいたAPIの機能を探すことができるものですが、これは理論的な概念で、 私たちは実際に動いてるのを見たことがないし、私たちのSOA機構にフィットしません。 またOpenAPIの記述フォーマット(とそのツール)は、HATEOASのサポートも十分ではありません。
- 実際、HATEOASに似た(HALやJSON APIのような)仕様も、URLエンドポイントやHTTPメソッドの性質から 情報を取り出すことによって、APIナビゲーションをサポートします。 したがってハイパーメディアはドメインモデルが徐々に変化していくときには、 結局クライアントは手動での変更が余儀なくされるのです。
- ハイパーメディアは人間にとっては意味のあるものですが、SOAクライアントにとってはそうでもありません。 私たちは、SOAクライアントがサービスドメイン境界にいるフロントエンドや人間に価値を届けることができる ユースケースを想定しているのです。
- ハイパーメディアは、APIクライアントが'discovering'を使わずに、ショートカットを実装したり、直接対象のリソースをターゲットにすることを防げません。

しかし、私たちはHATEOASを禁止するわけではありません。 その制限を理解し、複雑さを代償としてもより価値のある利用シーンがあるのであれば、HATEOASを使ってもかまいません。

MUST 絶対URIを使う

他のリソースへのリンクは、絶対URIでなくてはなりません。

動機: 相対URI (相対URLが絶対パスを使おうが相対パスを使おうが) の形式を露出させると、 クライアントサイドでは複雑性の混入を避けることができません。 埋め込みサブリソースのような機能を使うときは、絶対URIが与えられないとすれば、 ベースURIを明確にする必要があります。 絶対URIを使わないことの利点は、ペイロードサイズを小さくできること程度ですが、 それならば GZip圧縮を使った方がよいでしょう。

MUST 共通のハイパーテキストコントロールを使う

他のリソースへのリンクを埋め込むとき、共通のハイパーテキストコントロールオブジェクトを 使わなくてはなりません。ハイパーテキストコントロールオブジェクトには、少なくとも1つの属性が含まれます。

- `href`: ハイパーテキストコントロールがリンクしているリソースのURI。 私たちのすべてのAPIは、URI

スキームとして HTTP(s) を使っている。

どんなハイパーテキストコントロールを含むAPIにおいても、 属性名 `href` はハイパーテキストコントロールの範囲内での使用用途で予約語となります。

ハイパーテキストコントロールのスキーマは、以下のモデルから導き出されます。

```
HttpLink:
  description: A base type of objects representing links to resources.
  type: object
  properties:
    href:
      description: Any URI that is using http or https protocol
      type: string
      format: uri
  required:
    - href
```

`HttpLink` のようなオブジェクトをもつ属性の名前は、リンクとリンク先のリソースを含む オブジェクトとの関係を明記します。 実装は IANA Link Relation Registry から適切なものを選んで、その名前を使うべきです。 このガイドでは属性名にはスネークケースが使われていますが、 IANAのリンクリレーション名は、ハイフンケース表記が使われています。IANA名のハイフンは、 アンダースコアに変換しなければなりません。 (例えば、IANAリンクリレーションタイプ `version-history` は属性 `version_history` になります)

特定のリンクオフジェクトは、リンクに加えて、 リンク先のリソースに関する追加情報や、リンク元のリソースとリンク先のリソースの関係など、 その他の属性を含ませることができます。

例えば、"Person" リソースを提供するサービスは、他の誰かと結婚していることを、 その結婚相手の (`id`, `name`) だけでなく、いつから配偶者関係にあるかを示す (`since`) などを含んだハイパーテキストコントロールでモデル化します。

```
{
  "id": "446f9876-e89b-12d3-a456-426655440000",
  "name": "Peter Mustermann",
  "spouse": {
    "href": "https://...",
    "since": "1996-12-19",
    "id": "123e4567-e89b-12d3-a456-426655440000",
    "name": "Linda Mustermann"
  }
}
```

ハイパーテキストコントロールは、JSONモデルの範囲ではどこでも使えます。 この仕様においては HAL が使用可能ですが、APIの理解しやすさや使いやすさがもたらす価値よりも、 メタデータとデータの構造的な分離の有害さが上回るため、私たちはHALはおすすめいないし使いません。

SHOULD ページネーションや自己参照にシンプルなハイパーテキストコントロールを使う

コレクション内でのページネーションや自己参照のためのハイパーテキストコントロールは、 拡張共通ハイパーテキストコントロールを使うよりも、 リンクリレーション で定義されている (`next` `prev` `first`, `last`, `self`) 組合せたシンプルなURIを使うべきです。

[simple-hypertext-control-fields]と**SHOULD** 適用可能なところではページネーションリンクを使うに、より例と詳細な情報があります。

MUST JSONエンティティと一緒にはLinkヘッダは使わない

柔軟性と精度のため、一般的ではないリンクヘッダ構文を使って、リンクを付加するのではなく、JSONペイロードに直接埋め込まれるリンクを好みます。結果として、 RFC-8288で定義される `Link` ヘッダ を使うことは禁止されます。

16. データフォーマット

MUST 構造化データのエンコードはJSONを使う

構造化データを転送するためにJSONエンコードされたボディのペイロードを使いましょう。JSONペイロードは RFC 7159 にしたがわなければなりません。トップレベルの構造としては、(可能であれば)将来拡張可能なように、JSONオブジェクトを使います。これはまた、Arrayを想定しているであろうコレクションリソースにも適用します。**MUST** 常にトップレベルのデータ構造としてJSONオブジェクトを返す参照。

それに加えて、JSONペイロードは部分的に、 RFC 7493 に準拠しなければなりません。

- Section 2.1 の文字のエンコード
- Section 2.3 のオブジェクトの制約

結果としてJSONペイロードは、以下のようになります。

- UTF-8 エンコーディングを使う
- valid Unicode stringsを構成する。つまり文字以外やサロゲートを含んではならない。
- uniq member namesだけを含む。(重複した名前があってはならない)

MAY バイナリデータや別のコンテント表現には、JSONでないメディアタイプを使う

他のメディアタイプは次のようなケースで使われます。

- データがバイナリや構造と関係ないものである。ペイロード構造がパース不要でクライアントがそのまま受け取るものが、このケースにあたります。JPG/PNG/GIFなどのフォーマットの画像ダウンロードがその一例です。

- JSONバージョン以外のデータフォーマット(例えばPDF/DOC/XMLなど)を提供する。これらはコンテントネゴシエーションによって利用可能になるかもしれません。

SHOULD 埋め込みバイナリデータは `base64url` にエンコードする

代替メディアタイプを使ったバイナリーデータが一般的には好まれます。上のルールを参照。

もし代替のコンテント表現が望むものでなければ、バイナリデータは `base64url` エンコードされた文字列プロパティとして、JSONドキュメントに埋め込まれるべきでです。 RFC 7493 Section 4.4 参照。

SHOULD 標準のメディアタイプとして `application/json` を使う

以前このガイドラインでは、`application/x.zalando.article+json` のようなカスタムのメディアタイプを使ってもよいとしました。これは、media type versioningで必要な場合以外では、おすすめしないし避けるべきです。かわりに、標準のメディアタイプである `application/json` (または `application/problem+json` **MUST** Problem JSONを使う) を使いましょう。

x で始まるカスタムのメディアタイプは、JSONの標準メディアタイプと比較して何のメリットもないばかりか、自動化をより難しくしてしまいます。これはまた RFC 6838で使用を抑制されています。

SHOULD 標準化されたプロパティフォーマットを使う

JSON Schema と Open API では、いくつか有用で広く使えるプロパティフォーマットを定義しています。次の表は、Eコマース環境で特に役に立つ追加のフォーマットをいくつか含んでいます。

型	フォーマット	仕様	例
`integer`	`int32`		`7721071004`
`integer`	`int64`		`772107100456824`
`integer`	`bigint`		`77210710045682438959`
`number`	`float`	IEEE 754-2008	`3.1415927`
`number`	`double`	IEEE 754-2008	`3.141592653589793`
`number`	`decimal`		`3.14159265358979323846264338327 9`
`string`	`bcp47`	BCP 47	`"en-DE"`
`string`	`byte`	RFC 7493	`"dGVzdA=="`
`string`	`date`	RFC 3339	`"2019-07-30"`
`string`	`date-time`	RFC 3339	`"2019-07-30T06:43:40.252Z"`

型	フォーマット	仕様	例
string	email	RFC 5322	"example@zalando.de"
string	gtin-13	GTIN	"5710798389878"
string	hostname	RFC 1034	"www.zalando.de"
string	ipv4	RFC 2673	"104.75.173.179"
string	ipv6	RFC 2673	"2600:1401:2::8a"
string	iso-3166	ISO 3166-1 alpha-2	"DE"
string	iso-4217	ISO 4217	"EUR"
string	iso-639	ISO 639-1	"de"
string	json-pointer	RFC 6901	"/items/0/id"
string	password		"secret"
string	regex	ECMA 262	"^[a-z0-9]+$"
string	time	RFC 3339	"06:43:40.252Z"
string	uri	RFC 3986	"https://www.zalando.de/"
string	uri-template	RFC 6570	"/users/{id}"
string	uuid	RFC 4122	"e2ab873e-b295-11e9-9c02-…"

MUST 標準の日付・時刻フォーマットを使う

JSONペイロード

SHOULD 日付型のプロパティ値はRFC 3339に準拠する で日付・時刻のフォーマットについて書いています。

HTTPヘッダ

独自のものを含むHTTPヘッダは、 RFC 7231 で定義されている日付フォーマット を使いましょう。

SHOULD 国、言語、通貨のコードは標準を使う

国、言語、通貨は次の標準コードを使いましょう。

- ISO 3166-1-alpha2 国コード
  - (“UK”ではなく“GB”を使う。Zalandoでは「UK」というワードが使われているのを見かけるかもしれないが)
- ISO 639-1 言語コード

    - BCP 47 (based on ISO 639-1) for language variants
- ISO 4217 通貨コード

MUST 数値型と整数型のフォーマットを定義する

APIで `number` または `integer` の型のプロパティを定義するときは、 クライアントが誤った精度を使って、無意識に値が変わってしまわないように、精度を定義しなければなりません。

型	フォーマット	値の範囲
integer	int32	integer between -2^{31} and $2^{31}-1$
integer	int64	integer between -2^{63} and $2^{63}-1$
integer	bigint	arbitrarily large signed integer number
number	float	IEEE 754-2008/ISO 60559:2011 binary32 decimal number
number	double	IEEE 754-2008/ISO 60559:2011 binary64 decimal number
number	decimal	arbitrarily precise signed decimal number

精度はクライアントとサーバの双方で、もっとも適した言語の型に変換されなければなりません。 例えば、次の定義においてJavaでは、 `Money.amount` は `BigDecimal` に、 `OrderList.page_size` は `int` または `Integer` に変換されるでしょう。

```
components:
  schemas:
    Money:
      type: object
      properties:
        amount:
          type: number
          description: Amount expressed as a decimal number of major
currency units
          format: decimal
          example: 99.95
        ...

    OrderList:
      type: object
      properties:
        page_size:
          type: integer
          description: Number of orders in list
          format: int32
          example: 42
```

17. 共通のデータ型

広く利用されるデータオブジェクトの定義です。

SHOULD 共通のお金オブジェクトを使う

以下のような共通のお金構造を使いましょう。

```
Money:
  type: object
  properties:
    amount:
      type: number
      description: >
        The amount describes unit and subunit of the currency in a single
value,
        where the integer part (digits before the decimal point) is for
the
        major unit and fractional part (digits after the decimal point) is
for
        the minor unit.
      format: decimal
      example: 99.95
    currency:
      type: string
      description: 3 letter currency code as defined by ISO-4217
      format: iso-4217
      example: EUR
  required:
    - amount
    - currency
```

Moneyのグローバルスキーマを参照をincludeするとよい。

```
SalesOrder:
  properties:
    grand_total:
      $ref: 'https://opensource.zalando.com/restful-api-guidelines/money-
1.0.0.yaml#/Money'
```

APIが閉じたデータ型として、Moneyを取り扱わなくてはならないことに注意しましょう。つまり、インスタンス階層で使わることを意味しません。次のような使い方が許されないことを意味します。

```
{
  "amount": 19.99,
  "currency": "EUR",
  "discounted_amount": 9.99
}
```

Cons

- リスコフの置換原則 に違反している。
- 既存のライブラリのサポート(例えば、 Jackson Datatype Money)にのらない。
- amountが一緒に組み合わさるので、柔軟性が損なわれる。(例えば、複合的な通貨が表現できない)

より良いアプローチは 継承よりもコンポジション を使うことです。

```
{
  "price": {
    "amount": 19.99,
    "currency": "EUR"
  },
  "discounted_price": {
    "amount": 9.99,
    "currency": "EUR"
  }
}
```

Pros

- 継承がないので、置換原則にまつわる問題が無い。
- 既存のライブラリでサポートされる。
- 結合がない。つまり複合通貨も表現できる。
- 価格は自己記述的になり、アトミックな値となる。

注意

ビットコインのトランザクションのように、高い精度を要求する業務もあるので、API仕様に明記していない限りは、アプリケーションは制限なしの精度を受け取れるように 準備しておかなければなりません。 ユーロの正しい表記を例示します。

- `42.20` or `42.2` = 42 Euros, 20 Cent
- `0.23` = 23 Cent
- `42.0` or `42` = 42 Euros

- `1024.42` = 1024 Euros, 42 Cent
- `1024.4225` = 1024 Euros, 42.25 Cent

特定の言語でこのインタフェースを実装したり計算したりする際には、 "amount"フィールドを決して `float` や `double` 型に変換してはなりません。 そうしないと精度が失われてしまいます。代わりにJavaの `BigDecimal` のような正確なフォーマットを使いましょう。 詳細は Stack Overflow

いくつかのJSONパーサ(例えばNodeJS)は、デフォルトでnumberをfloatに変換してしまいます。 メリデメの議論を経て、私たちは金額のフォーマットに"decimal"を使うことに決めました。 OpenAPIフォーマットの標準ではないけれど、パーサがnumberをfloatやdoubleに変換してしまうことを避けることができるからです。

MUST 共通のフィールド名やセマンティクスを使う

複数の場所で使われるフィールドの型があります。すべてのAPI実装にわたって一貫性を保つために、 どんなときでも適用可能な共通のフィールド名とセマンティクスを使わなければなりません。

一般的なフィールド

APIに繰り返し出てくるフィールドは以下のようなものです。

- `id`: オブジェクトのID。 IDは数値でなく文字列でなくてはなりません。IDは文書化されているコンテキストの範囲でユニークかつ不変です。一度オブジェクトに付与されたら変更されてはならないし、再利用してもいけません。
- `xyz_id`: オブジェクトが別のオブジェクトのIDを持つ場合、相手オブジェクト名に `_id` を付与した名前を使いましょう。 (e.g. `customer_number` ではなく `customer_id` ; 子ノードから親ノードを参照する場合は、たとえ両方が `Node` 型であっても、`parent_node_id` とします)
- `created_at` オブジェクトが作られた日時。`date-time` 型でなくてはなりません。
- `modified_at`: オブジェクトが更新された日時。`date-time` 型でなくてはなりません。
- `type`: オブジェクトの種類。このフィールドの型はstringとするべきです。typeはエンティティについてのランタイム情報を与えます。
- `ETag`: 埋め込みサブリソースのETag。続くPUT/PATCHの呼び出しでETagを渡すのに使われる。(結果エンティティにおける ETag参照)

JSONスキーマの例:

```yaml
tree_node:
  type: object
  properties:
    id:
      description: このノードの識別子
      type: string
    created_at:
      description: このノードがいつ作られたか
      type: string
      format: 'date-time'
    modified_at:
      description: このノードが最後に更新されたのはいつか
      type: string
      format: 'date-time'
    type:
      type: string
      enum: [ 'LEAF', 'NODE' ]
    parent_node_id:
      description: このノードの親ノードの識別子
      type: string
  example:
    id: '123435'
    created: '2017-04-12T23:20:50.52Z'
    modified: '2017-04-12T23:20:50.52Z'
    type: 'LEAF'
    parent_node_id: '534321'
```

これらのプロパティはいつも必要というわけではありませんが、これを慣例にしておくことで、 APIクライアント開発者にとってZalandoリソースの共通理解が容易になるわけです。 異なる名前が使われたり、APIごとにこれらの型が違ったりすると、API利用者にとっては不便なものになってしまいますからね。

リンク関連フィールド

ページネーションとコレクション上での繰り返しをするために、シンプルなハイパーテキスト制御を使って一貫したルックアンドフィールを提供するには、レスポンスオブジェクトは、以下の共通パターンにしたがうべきです。

- `self`: 同一のコレクションオブジェクトまたはページを指し示す、ページネーションレスポンスまたはオブジェクトへのリンクおよびカーソル
- `first`: 最初のコレクションオブジェクトまたはページを指し示す、ページネーションレスポンスまたはオブジェクトへのリンクおよびカーソル
- `prev`: 前のコレクションオブジェクトまたはページを指し示す、ページネーションレスポンスまたはオブジェクトへのリンクおよびカーソル
- `next`: 次のコレクションオブジェクトまたはページを指し示す、ページネーションレスポンスまたはオブジェクトへのリンクおよびカーソル

- `last`: 最後のコレクションオブジェクトまたはページを指し示す、ページネーションレスポンスまたはオブジェクトへのリンクおよびカーソル

ページネーションのレスポンスは、ページ内容を送るためのArrayフィールドを追加で持ちます。

- `items`: 現在のページのすべてのアイテムをもつリソースの配列 (`items` はリソースの名前でもよい).

ユーザ体験を単純化するため、適用されたクエリフィルタを使います。(`GET With Body`も参照)

- `query`: コレクションリソースをフィルタする検索リクエストでの適用されたクエリフィルタを含むオブジェクト。

結果として、ページネーションリンクを使った標準のレスポンスページは、以下のようになります。

```
ResponsePage:
  type: object
  properties:
    self:
      description: 現在のページを指すページネーションリンク。
      type: string
      format: uri
    first:
      description: 最初のページを指すページネーションリンク。
      type: string
      format: uri
    prev:
      description: 前のページを指すページネーションリンク。
      type: string
      format: uri
    next:
      description: 次ページを指すページネーションリンク。
      type: string
      format: uri
    last:
      description: 最後のページを指すページネーションリンク。
      type: string
      format: uri

     query:
       description: >
        コレクションリソースに適用されたクエリフィルタを含むオブジェクト。
       type: object
       properties: ...

     items:
       description: コレクションアイテムのArray。
       type: array
       required: false
       items:
         type: ...
```

レスポンスページは、コレクションや現在のページに関する追加のメタデータを含むかもしれません。

住所フィールド

住所の構造は国の違いを含む様々な機能、ユースケースに影響します。 住所に関するすべての属性は、以下で定義された名前とセマンティクスにしたがいます。

```
addressee:
  description: a (natural or legal) person that gets addressed
  type: object
```

```
  properties:
    salutation:
      description: |
        a salutation and/or title used for personal contacts to some
        addressee; not to be confused with the gender information!
      type: string
      example: Mr
    first_name:
      description: |
        given name(s) or first name(s) of a person; may also include the
        middle names.
      type: string
      example: Hans Dieter
    last_name:
      description: |
        family name(s) or surname(s) of a person
      type: string
      example: Mustermann
    business_name:
      description: |
        company name of the business organization. Used when a business is
        the actual addressee; for personal shipments to office addresses,
use
        `care_of` instead.
      type: string
      example: Consulting Services GmbH
  required:
    - first_name
    - last_name

address:
  description:
    an address of a location/destination
  type: object
  properties:
    care_of:
      description: |
        (aka c/o) the person that resides at the address, if different
from
        addressee. E.g. used when sending a personal parcel to the
        office /someone else's home where the addressee resides
temporarily
      type: string
      example: Consulting Services GmbH
    street:
      description: |
        the full street address including house number and street name
      type: string
      example: Schönhauser Allee 103
```

```
    additional:
      description: |
        further details like building name, suite, apartment number, etc.
      type: string
      example: 2. Hinterhof rechts
    city:
      description: |
        name of the city / locality
      type: string
      example: Berlin
    zip:
      description: |
        zip code or postal code
      type: string
      example: 14265
    country_code:
      description: |
        the country code according to
        [iso-3166-1-alpha-2](https://en.wikipedia.org/wiki/ISO_3166-
1_alpha-2)
      type: string
      example: DE
  required:
    - street
    - city
    - zip
    - country_code
```

特定データ型におけるフィールドのグルーピングやカーディナリティは、特定のユースケースに基づいています。 (例えば、宛先をモデル化するときは受取人と住所のフィールドの組み合わせをるけれども、ユーザと住所をモデル化するときは、受取人と住所は別にする、ということです)

18. 共通のヘッダ

このセクションでは私たちが毎日使う中で疑問に思ったり、あまり知られてないけれど 特定の状況では役に立ったりするいくつかのヘッダについて記述します。

MUST `Content-*` ヘッダを正しく使う

Contentやエンティティに関するヘッダには、`Content-` のプレフィクスが付いています。 これらにはメッセージボディの内容に関することが書かれていて、HTTPリクエストとレスポンスの両方で使用されます。 共通的に使われるContentヘッダは次のようなものですが、その限りではありません。

- `Content-Disposition` はファイルとして保存されることを意図したり、そのときのデフォルトのファイル名を与えるのに使う。
- `Content-Encoding` はContentに適用される圧縮/暗号アルゴリズムを示す。

- `Content-Length` はContentの長さをbyte長で示す。
- `Content-Language` はボディがある人間のための言語で書かれていることを示す。
- `Content-Location` はボディが別の場所にあることを示す (**MAY** Content-Location ヘッダを使う により詳細があります)。
- `Content-Range` はボディの一部を取得することを指し示すのに使われる。
- `Content-Type` は、ボディのメディアタイプを示す。

MAY 標準のヘッダを使う

このリスト を使い、Open API定義にサポートするヘッダを記述します。

MAY Content-Location ヘッダを使う

Content-Location ヘッダは *任意* であり、成功した書き込み操作(`PUT`, `POST`, `PATCH`)や読み込み操作(`GET`, `HEAD`) で使われ、キャッシュ位置を示したり、リソースの実際の場所を受信者に通知したりします。 これによりクライアントはリソースを識別し、このヘッダの付いたレスポンスを受け取ったらローカルコピーを更新することができるのです。

Content-Locationヘッダは、次のユースケースを実現するのに使われます。

- `GET`や`HEAD`で、リクエストされたURIとは異なる場所が、 返されるリソースはコンテントネゴシエーションに依存したものであったり、リソース固有の識別子を与えることを示すのに使われる。
- `PUT`や`PATCH`では、リクエストされたURIと同一の場所を指し、 返却されたリソースが、新しく生成/更新されたリソースの現在の表現であることを明示するのに使われる。
- `POST`や`DELETE`では、 リクエストされたアクションに対するレスポンスに、ステータスレポートリソースが含まれることを示すのに使われる。

注意: Content-Locationヘッダを使用する際には、Content-Typeヘッダも正しく 設定しなければならない。例えば、以下のように。

```
GET /products/123/images HTTP/1.1

HTTP/1.1 200 OK
Content-Type: image/png
Content-Location: /products/123/images?format=raw
```

SHOULD `Content-Location` の代わりに `Location` ヘッダを使う

セマンティクスやキャッシュに関して、`Content-Location`を正しく使うのは 難しいので、私たちは`Content-Location` の使用を *推奨していません* 。 たいていの場合、`Content-Location`特有の曖昧さや複雑さに悩まされる代わりに、 `Location`ヘッダを使うことで、クライアントにリソースの場所を直接知らせることで十分です。

より詳細な話が、RFC 7231 7.1.2 Location , 3.1.4.2 Content-Location にあります。

MAY 処理するプリファレンスを示すために`Prefer`ヘッダのサポートを検討しよう

`Prefer`ヘッダは {RFC7240}[RFC 7420] で定義されており、クライアントがサーバの振る舞いをリクエストするのに使われます。多くのプリファレンスが事前定義されていて拡張も可能です。 `Prefer`ヘッダのサポートは、任意でありAPI設計の裁量次第ですが、 既存のインターネット標準と同様に、独自の"X-"ヘッダを定義して処理することをおすすめします。

`Prefer`ヘッダはAPI定義に次のように定義します。

```
components:
  headers:
  - Prefer:
      description: >
        RFC7240のPreferヘッダは特定のサーバの挙動が、そのクライアントにとって望ましくリクエストが ([RFC7240](https://tools.ietf.org/html/rfc7240)参照)
        このAPIでは次の振る舞いがサポートされる。

        # (APIまたはAPIエンドポイントによって)
        * **respond-async** は結果を待つかわりに、202 - accepted -
を使って非同期にできるだけ速く応答を返すようにサーバに伝える。
        * **return=<minimal|representation>** はリソースなしで
204を使い応答を返して欲しい場合(minimal)、リソースありで200や201を使い応答を返して欲しい場合(representation)を使い分ける。
        * **wait=<delta-seconds>**
はリクエストを同期的に処理する最大時間を示すのに使う。
        * **handling=<strict|lenient>**
はエラーに対して厳格でレポートするか、あるいは寛容で可能な限り処理を継続するかをサーバに指示するのに使う。
      in: header
      type: string
      required: false
```

注意: APIエンドポイントがサポートする`Prefer`ヘッダの仕様へは振る舞いだけをコピーしよう。必要ならそれぞれのサポートされるユースケースで異なる`Prefer`ヘッダを明記しよう。

サポートするAPIは`Preference-Applied`ヘッダを返してもよい。これは RFC 7240 で定義され、プリファレンスが適用されたかどうかを指し示す。

MAY If-Match/If-None-MatchヘッダともにEtagのサポートを検討しよう

リソースが作成、更新されるときは、コンフリクトの発生を検知し、'更新データのロスト'や'重複して作成される'問題を防ぐ必要があります。 [RFC-7232]{RFC 7232 "HTTP: Conditional Requests"} にしたがい、`ETag` ヘッダを`If-Match`または`If-None-Match`の条件ヘッダとともに使うことで、それが出来るようになります。 `ETag: <entity-tag>` ヘッダの内容は、(a) レスポンスボディのハッシュ値か、 (b) エンティティの最終更新日時フィールドのハッシュ値、(c) エンティティのバージョンの番号または識別子の何れかにします。

`PUT`, `POST`, `PATCH`の同時更新操作でコンフリクトが発生したことを検出するために、サーバは `If-Match: <entity-tag>` ヘッダがあれば、更新エンティティのバージョンが、リクエストの{entity-tag>}と一致しているかをチェックしなければなりません。 もし一致するエンティティがなければ、412 - precondition failed のステータスコードを返すようにします。

他のユースケース、リソース生成時にコンフリクトを検出する方法と同様に、 `If-None-Match: *` が使えます。 もしエンティティにマッチするものがあれば、既に同じリソースが作成されていることを 示すので、412 - precondition failed のステータスコードを返します。

`ETag`, `If-Match`, `If-None-Match`ヘッダは、API定義においては次のように定義されます。

```yaml
Etag:
  name: Etag
  description: |
    The RFC7232 ETag header field in a response provides the current
entity-
    tag for the selected resource. An entity-tag is an opaque identifier
for
    different versions of a resource over time, regardless whether
multiple
    versions are valid at the same time. An entity-tag consists of an
opaque
    quoted string, possibly prefixed by a weakness indicator.

  in: header
  type: string
  required: false
  example: W/"xy", "5", "7da7a728-f910-11e6-942a-68f728c1ba70"

IfMatch:
  name: If-Match
  description: |
    The RFC7232 If-Match header field in a request requires the server to
    only operate on the resource that matches at least one of the provided
    entity-tags. This allows clients express a precondition that prevent
    the method from being applied if there have been any changes to the
    resource.

  in: header
  type: string
  required: false
  example:  "5", "7da7a728-f910-11e6-942a-68f728c1ba70"

IfNoneMatch:
  name: If-None-Match
  description: |
    The RFC7232 If-None-Match header field in a request requires the
server
    to only operate on the resource if it does not match any of the
provided
    entity-tags. If the provided entity-tag is `*`, it is required that
the
    resource does not exist at all.

  in: header
  type: string
  required: false
  example: "7da7a728-f910-11e6-942a-68f728c1ba70", *
```

別のアプローチについての議論は、RESTful APIにおける楽観ロック セクションも参照ください。

MAY `Idempotency-Key` ヘッダのサポートを検討しよう

リソースを生成したり更新したりするとき、タイムアウトやネットワーク障害のためにリトライするケースで、重複実行を避けるため同じレスポンを返す強い冪等性が役に立ったり、必要になったりします。一般的にこれはクライアント固有の *一意のリクエストキー* をリソースの一部ではなく、`Idempotency-Key`ヘッダを通じて送信することによって実現されます。

一意のリクエストキー は、一時的(例えば24時間くらい)に保存され、成功したか失敗したかによらずレスポンスと(これはオプションですが)最初のリクエストのハッシュを一緒に格納します。サービスは、冪等動作を保証するために、リクエストをリトライする代わりに、キーキャッシュ内の_一意のリクエストキー_を検索し、キーキャッシュからレスポンスを返すことができます。オプションとして、レスポンスを返す前にリクエストのハッシュを使って整合性をチェックできます。キーがキーストアにない場合、リクエストは通常どおり実行され、レスポンスはキーキャッシュに格納されます。

これにより、クライアントは同じレスポンスを複数回受信しながら、タイムアウトやネットワーク障害などの後に安全にリクエストをリトライできます。 **注意:** このコンテキストでのリクエストのリトライは、全く同じリクエストを送信する必要があります。つまり、結果を変更するようなリクエストの更新は禁止されています。キーキャッシュ内のリクエストハッシュは、この誤った使い方から保護することができます。このようなリクエストは、ステータスコード400を使用して拒否することをお勧めします。

重要: 信頼性の高い冪等実行セマンティクスを付与するには、分散システムにおける障害、タイムアウト、同時リクエストの潜在的なすべての落とし穴を考慮して、リソースとキーキャッシュをハードトランザクションセマンティクスで更新する必要があります。これは、ローカルコンテキストを超える正しい実装を非常に難しくします。

`Idempotency-Key` ヘッダーは次のように定義する必要がありますが、有効期限は自由に選択できます。

```
components:
  headers:
  - Idempotency-Key:
      description: |
        Idempotency Keyは、リクエストを一意に識別するためクライアントによって生成
        される自由な識別子である。サービスによって同じリクエストのリトライである
        ことを特定したり、2度同じリクエストを実行せずに、同じリクエストを返すこと
        で冪等な振る舞いを保証するために、サービスによって使われる。

        クライアントは同じキーをもつ一連のリクエストは、さらなるチェックなしに
        同じレスポンスを返すかもしれないことに注意すべきだ。それゆえUUID v4
(Random)や他の十分衝突を回避できるだけのエントロピーをもったランダム文字列を使うこと
        を推奨する。

        Idempotency Keyは24時間で有効期限が切れる。クライアントはこの制限内で
        これらを使う責任を持たねばならない。

      type: string
      format: uuid
      require: false
      example: "7da7a728-f910-11e6-942a-68f728c1ba70"
```

ヒント: キーキャッシュはリクエストログとして意図されていません。したがって、その生存期間は制限すべきだし、そうしないとデータリソースのサイズを簡単に超えてしまうことになります。

注意: `Idempotency-Key` ヘッダはこのセクションの他のヘッダと異なりRFCで標準化されていません。Stripe APIでの使われ方だけを参考にしました。独自ヘッダのセクションの規約とは合いませんが、ヘッダの名前と意味を変えたくはなく、他の共通ヘッダと同じようにこれを扱うことを決めました。

19. 独自ヘッダ

このセクションは、独自ヘッダの定義について共有します。 独自ヘッダは、サービス全体にまたがる懸案事項となるため、一貫した名付けをすべきです。 サービスがこれらをサポートするかどうかは任意です。 それゆえOpen APIの仕様には、これを明示的に可視化する正しい場所があります。 リソースのHTTPメソッドのパラメータ定義を使いましょう。

MUST 独自Zalandoヘッダのみを使う (訳注: Zalando社の固有ルール)

一般的なルールとして、独自HTTPヘッダは避けるべきというのがあります。エンドトゥエンドで複数のサービスを通して渡される必要があるとそれが効いてきます。 APIの一部分だけではなく、それに続くやり取りに必要とされる文脈情報も与えるようなところで独自ヘッダを用いるのは、妥当なユースケースです。

概念的な観点から、API操作の意味と意図はURLパスとクエリパラメータ、メソッド、コンテントで常に 表現されているべきです。ヘッダはフロー制御やコンテントネゴシエーション、認証のようなプロトコル に近い機能を実装するのに使われます。 ですので、ヘッダは一般的な文脈情報のために予約されているのです。

(RFC-7231)

X- ヘッダは、最初は非標準のパラメータのために予約されていましたが、 X- ヘッダを使用は、 RFC 6648 で廃止予定になりました。 これらのヘッダを使う整合性の取れた方法がないので、ガイドラインにしたがう API の利用側・提供側の間でのやりとりの規定は複雑化します。 そのため、ガイドラインではどの X- ヘッダを使用するか、どう使われるかを制限しています。

RFC6648 の中で、 企業固有のヘッダの名前には、組織の名前を含むべきだ、という見解を Internet Engineering Task Force は示しています。 私たちは後方互換の目的で、 X- 始まりのヘッダを使い続けています。

次の独自ヘッダは、このガイドラインで使用法が規定されたものです。 HTTPヘッダフィールド名は、大文字・小文字を区別しないことをお忘れなく。

ヘッダフィールド名	型	説明	値の例
X-Flow-ID	String	より多くの情報は**MUST/SHOULD** 機能本位の命名体系を使うを参照	GKY7oDhpSiKY_gAAAABZ_A
X-Tenant-ID	String	マルチテナントのZalandプラットフォームにおいて リクエストを送信したテナントを識別するためのID。 X-Tenant-IDは、OAuthトークンから 抽出したビジネスパートナーIDに応じてセットする。	9f8b3ca3-4be5-436c-a847-9cd55460c495
X-Sales-Channel	String	セールスチャネルは小売が所有していて、 特定の消費者セグメントが、CFA小売カタログを通じて消費者に提供される特定の製品群に向いていることを表現する。	52b96501-0f8d-43e7-82aa-8a96fab134d7
X-Frontend-Type	String	消費者接点のアプリケーション(Consumer facing applications; CFAs) は、モバイルアプリやブラウザアプリのような異なるフロントエンドのアプリケーションを通じて、 カスタマにビジネスエクスペリエンスを提供する。 例えば特定のクーポンをモバイルにプッシュし使わせるような。 現在、mobile-app, browser, facebook-app, chat-app が使える。	mobile-app
X-device-Type	String	デバイスの種類に依存した (機能やコンテンツを含む) カスタマエクスペリエンスを 提供するようなユースケースがある。 このヘッダ情報がその観点で使われるべきである。現在は、smartphone, tablet, desktop, otherが使える。	tablet

ヘッダフィールド名	型	説明	値の例
X-device-OS	String	前述のデバイス種類の上で、デバイスプラットフォーム、例えば AndroidかiOSかといったようなその違いをハンドリングしたい場合は、これを使う。 現在はiOS, Android, Windows, Linux, MacOSが使える。	Android
X-Mobile-Advertising-ID	String	iOSの IDFA (Apple Identifier for mobile Advertising) またはAndroidの GAID (Google mobile Advertising Identifier)である。モバイルデバイスのOSによって与えられるカスタマがリセット可能な一意の識別子であり、パーソナライズド広告に利用される。呼び出されたs-ビスは別のサービスを呼び出すときにこの値も渡すようにしておくべきだ。それぞれのモバイルプラットフォームで、カスタマはこの機能を無効に設定できる。	b89fadce-1f42-46aa-9c83-b7bc49e76e1f

例外: このガイドラインの唯一の例外は、 **MUST** レート制限のためのヘッダと429コードを使う の定義として使われる hop-by-hopの X-RateLimit- ヘッダです。

MUST 独自ヘッダを伝搬する

すべてのZalandoの独自ヘッダは、End-to-Endのヘッダです。 [2]

上で定義したすべてのヘッダが、サービスの呼び出しチェーンで伝搬されなければなりません。 ヘッダの名前と値は書き換えてはなりません。

例えば、X-Device-Type のようなカスタムヘッダの値は、 デバイス種別の情報を使われ、クエリ結果に直接影響を与えることがあります。またリコメンド結果にも及ぶことがあります。さらにカスタムヘッダの値が、クエリ結果に間接的に影響することもあります。（デバイスタイプの情報がリコメンド結果に間接的に影響するなど）

時々、独自ヘッダの値は、後続のリクエストにおいてエンティティの一部として使われることがあります。そのような場合は、冗長かもしれませんが、独自ヘッダを後続リクエストのヘッダとしても付けて送らなければなりません。

MUST X-Flow-ID をサポートする

Flow-ID はサービスのAPIやイベントを通じて渡される汎用のパラメータで、ログファイルやトレースに書かれる。 その結果 *Flow-ID* を使うと、システム内のコールフローを追うのが簡単になり、特定の呼び出しによって開始されるサービスのアクティビティの関係性が明らかになります。これは運用上のトラブルシューティングやログ分析に非常に役に立ちます。 *Flow-ID* の主な用途は、当社のSaaSファッションコマースのサービス呼び出しと内部のプロセスフロー(API経由で同期的に実行されたものだけでなく、送出されたイベントによって非同期に実行されるものも)を追跡することです。

データ定義

Flow-ID は以下を通じて渡されます。

- `X-Flow-ID` 独自ヘッダを介したRESTful APIリクエスト(**MUST** 独自ヘッダを伝搬する参照)
- `flow_id` イベントフィールドをもつ送出されたイベント(metadata参照)

以下の形式が使えます。

- `UUID` (RFC-4122)
- `base64` (RFC-4648)
- `base64url` (RFC-4648 Section 5)
- `[a-zA-Z0-9/+_-=]` の文字を使った最大128文字のランダムな一意の文字列

注意: もしレガシーなシステムが、特定の形式や長さの *Flow-ID* しか扱えないならば、そのAPI仕様でこの制限を明記しなければなりません。寛大に不正な文字を削除したり、サポートされた長さに切り詰めたりします。

サービスガイド

- サービスは *Flow-ID* を汎用的な入力としてサポート **しなければならない** ,つまり
  - RESTful APIエンドポイントは、リクエストのヘッダで、`X-Flow-ID`を **サポートしなければならない**
  - イベントリスナーはイベントから メタデータ `flow-id` をサポートしなければならない

注意: APIクライアントはサービスを呼び出したり、イベントを生成するときには、 *Flow-ID* を **付与しなければならない** 。 もし *Flow-ID* がリクエストやイベントになけ r ば、サービスは新たに *Flow-ID* を生成しなければならない。

- サービスは *Flow_ID* を伝播しなければならない。つまり、 以下のようなところで *Flow-ID* を使う。
  - 処理中に呼び出したすべてのAPIや送出したすべてのイベントの入力
  - ログやトレースに書かれたデータフィールド

ヒント: このルールはまた、アプリケーション内部のインタフェースやNakadi経由(訳注: Zalando社のイベントドリブンプラットフォーム)で送出されないイベントにも適用される。

20. APIの運用

MUST Open API仕様を公開する

すべてのサービスアプリケーションは、外に向けたAPIのOpen APIの仕様を公開しなければなりません。 内部API (つまり APIオーディエンス が **component-internal** のもの) は任意ではありますが、APIマネジメント基盤を使うメリットは大きいので、公開しておくにこしたことはありません。

APIはその **OpenAPI仕様** を、プロビジョニングサービスをデプロイするのに使われる **デプロイメント成果物** の中の **/zalando-apis** ディレクトリへコピーすることで公開されます。 ディレクトリは それぞれ1つのAPIについて書かれた **自己完結型のYAML ファイル** だけを置くようにしてください。 (今となってはレガシーな) `.well-nown/schema-discovery` サービスエンドポイントを使ったデプロイよりも、私たちは、この成果物ベースのデプロイを優先します。レガシーな方式は、後方互換のために残してあるだけです。

背景: 動的で複雑な私たちのインフラでは、APIクライアント開発者に、動いているアプリケーションすべてのAPI仕様を、オンラインからアクセス出来るような場所を提供することが重要です。 インフラの一部として、API公開のプロセスはAPI仕様を探し出すのに使われます。

注意: APIを公開するためには、デプロイが成功していることが前提になります。

SHOULD API利用状況をモニタリングする

本番環境で使われるAPIのオーナーは、使っているクライアントについて情報を得るために APIサービスをモニタリングすべきです。その情報は、例えばAPI変更時にレビューをお願い しなきゃいけない相手を特定するのに役立ちます。

ヒント: クライアントを見つけるより良い方法は、OAuthトークンから取り出したclient-idのログを取ることです。

21. イベント

Zalandoのアーキテクチャは疎結合なマイクロサービス中心で作られているので、 私たちは非同期なイベント駆動のアプローチを好みます。 このセクションのガイドラインは、イベントの設計と送信の仕方にフォーカスしたものになります。

イベント、イベントの型および分類

イベントは *イベント型* と呼ぶ項目を使って定義します。 イベント型は送信者によってスキーマを使って宣言され、受信者によって解釈されるイベント構造をもちます。 イベント型は、名前、オーナーアプリケーション(暗黙的にオーナーのチーム)、イベントのカスタムデータ を定義したスキーマ、スキーマがどう進化していくかを宣言している互換モードなどを、 標準情報として宣言します。 イベント型では、またイベントのバリデーションや強化戦略、イベントストリームの中で、 イベントがどうパーティショニングされうるか、のような補足情報を宣言してもかまいません。

イベント型は、(データ変更カテゴリのように) *イベントカテゴリ* に属します。 イベントカテゴリはイベントの種類に共通な追加情報を提供します。

イベント型は チームがみな使えるようAPIリソースとして、典型的には *イベント型レジストリ* に登録して公開します。送受信されるイベントは、そのイベント型の全体の構造と カスタムデータのためのスキーマに対

して、検証済みのものでなくてはなりません。

上述の基本モデルは、 Nakadi プロジェクト として元々開発されたものです。 Nakadiはイベント型のレジストリの参照実装であり、 イベントの送受信者のために、pub-subの検証ブローカとして動作します。

MUST サービスインタフェースの一部としてイベントを取り扱う

イベントはサービスのREST APIへ同じ立場であり、外界に対してのサービスインタフェースの 一部です。 データを送出するサービスは、APIと同じように、イベントを設計における最重要関心事として 扱わなければなりません。 [はじめに]で示した「APIファースト」の原則が、イベントに対しても当てはまります。

MUST レビューできるようにEventのスキーマを作る

イベントを送出するサービスは、他で使えるようにイベントのスキーマを作らなければなりません。 それだけでなく、レビューのためにもイベントの型定義も作りましょう。

MUST イベントスキーマはOpen APIスキーマオブジェクトに準拠する

API仕様にイベントスキーマ仕様も揃えるために、私たちはイベントスキーマの定義にも Open API仕様を使ってスキーマオブジェクトを定義します。 これは他のAPIで使われているリソースに関するデータ変更を表すイベントにとって、特に便利なものです。

Open API スキーマオブジェクト は JSON Schema Draft 4 の **拡張可能なサブセット** です。 便宜上、私たちは以下にその重要な差異を示します。 詳細は Open API スキーマオブジェクト仕様 を参照ください。

Open APIスキーマオブジェクトは、いくつかのJSONスキーマキーワードが削除されているので、 イベントスキーマでもこれらは使わないようにしてください。

- `additionalItems`
- `contains`
- `patternProperties`
- `dependencies`
- `propertyNames`
- `const`
- `not`
- `oneOf`

一方でスキーマオブジェクトは、JSONスキーマキーワードを再定義しているものもあります。

- `additionalProperties`: 互換性保証を謳うイベント型には、このフィールドを使うことに関しては制

約を設けておたほうがよいでしょう。詳細は**SHOULD** イベント型定義では `additionalProperties` を避けるのガイドラインをみてください。

最後に、スキーマオブジェクトは、JSONスキーマのいくつかのキーワードを *拡張しています* 。

- `readOnly`: イベントが論理的にイミュータブルであることを意味します。 `readOnly`は冗長とみなされるかもしれませんが無害です。
- `discriminator`: `oneOf` の代替としてポリモーフィズムをサポートするため。
- `^x-`: ベンダ拡張 の形式でパターン化されたオブジェクトがイベント型スキーマでも使えます。 しかし、汎用目的のバリデータはバリデーションを実行しませんし、無視されるべき処理にフォールバックします。 将来のガイドラインのバージョンでは、イベントのベンダ拡張について、もっとしっかり定義するかもしれません。

MUST イベントはイベント型として登録されていることを保証する

Zalandoのアーキテクチャにおいて、イベントは *イベント型* と呼ばれる 構造を使って登録されます。イベント型では、次のような標準の情報を宣言します。

- イベントカテゴリ。「汎用」や「データ更新」のように、よく知られたものです。
- イベント型の名前
- イベントの対象オーディエンス の定義
- 所有アプリケーション
- イベントのペイロードを定義するスキーマ
- 型の互換モード

イベント型はイベント情報を見つけるのを簡単にし、それが良く構造化されていて、 一貫性があって検証可能であることを保証するものになります。

イベント型のオーナーは、互換モードの選択に気をつけなければなりません。 モードはスキーマの進化の方法を示します。 イベント送信者が既存のイベント受信者に不用意に破壊的変更を与えずに、スキーマを 修正するのにどれだけな柔軟性があるかは、モードの範囲の設計に依存します。

- `none`: たとえ既存のイベント送信者・受信者を破壊しようと、どんなスキーマの修正も受け入れられる。イベントを検証する際は、スキーマで宣言されていない未定義のプロパティも受け入れなければ ならない。
- `forward`: スキーマ `S1` は、以前登録されたスキーマ `S0` が `S1` で定義されたイベントを 読むことができる、すなわちイベント受信者は、 API設計の原則 ガイドラインの ロバストネスの原則にしたがう限り、以前のバージョンを使っている最新のスキーマバージョンで タグ付けされたイベントも読むことができます。
- `compatible`: これは変更が完全な互換性をもつことを意味します。 最初のスキーマバージョンから、送出されたすべてのイベントが 最新のスキーマでも有効なものであるとき、新しいスキーマ `S1` は、 完全互換です。 compatibleモードでは、既存のスキーマへは新しい任意のプロパティと定義の追加のみ許さ

れ、 他の変更は禁止されます。

互換性モードはセマンティックバージョニング (MAJOR.MINOR.PATCH) にしたがう `version`フィールドに影響します。

- 互換モード `compatible` では、イベント型はPATCHまたはMINORバージョンのみ 変更でき、破壊的変更であるMAJORバージョンアップは許されまない。
- 互換モード `forward` では、イベント型はPATCHまたはMINORバージョンのみ 変更でき、破壊的変更であるMAJORバージョンアップは許されない。
- 互換モード `none` では、イベント型はPATCH、MINOR、MAJORすべてのレベルの 変更ができる。

次の例でこの関係性を説明します。

- イベント型の `title` または `description` を変更することは、PATCHレベルとみなす
- イベント型に任意のフィールドを追加することは、MINORレベルの変更とみなす
- 名前の変更やフィールドの削除、必須フィールドの新規追加など、他のすべての変更はMAJORレベルとみなす。

イベント型の主要な構造は、Open APIオブジェクトとして、以下のように定義されます。

```
EventType:
  description: |
    イベント型はスキーマと実行時のプロパティを定義します。必須のフィールドはイベント型の
    作成者が最低限セットすることが期待されているものです。
  required:
    - name
    - category
    - owning_application
    - schema
  properties:
    name:
      description: |
        このEventTypeの名前です。 注意: 全体での一意性と可読性を保つため、
        `<functional-name>.<event-name>` の形式で命名するようにしてください。
      type: string
      pattern: '[a-z][a-z0-9-]*\.[a-z][a-z0-9-]*'
      example: |
        transactions.order.order-cancelled,
        customer.personal-data.email-changed
    audience:
      type: string
      x-extensible-enum:
        - component-internal
        - business-unit-internal
        - company-internal
        - external-partner
```

```
        - external-public
      description: |
        イベント型の対象オーディエンス。ルール #219 でのREST
APIのオーディエンス定義に相当するものです。
    owning_application:
      description: |
        この `EventType` を所有するアプリケーションの名前です。
        (基盤アプリケーションやサービスレジストリで使われます)
      type: string
      example: price-service
    category:
      description: このEventTypeのカテゴリです。
      type: string
      x-extensible-enum:
        - data
        - general
    compatibility_mode:
      description: |
        このスキーマを発展させていくための互換性モードです。
      type: string
      x-extensible-enum:
        - compatible
        - forward
        - none
      default: forward
    schema:
      description: このEventTypeの最新のペイロードのスキーマです。
      type: object
      properties:
        version:
          description: セマンティックバージョニングに基づくバージョン番号です
("1.2.1"のようなもの)。
          type: string
          default: '1.0.0'
        created_at:
          description: スキーマの作成日時
          type: string
          readOnly: true
          format: date-time
          example: '1996-12-19T16:39:57-08:00'
        type:
          description: |
             スキーマ定義のスキーマ言語です。現在はjson_schema (JSON Schema v04)
のみ
             が定義できます。がこれは将来的には他のものも指定可能になるでしょう。
          type: string
          x-extensible-enum:
            - json_schema
        schema:
```

```
          description: |
              フィールド型に定義された文法で表現した文字列としてのスキーマ
          type: string
      required:
        - type
        - schema
    ordering_key_fields:
      type: array
      description: |
        Indicates which field is used for application level ordering of
events.
        It is typically a single field, but also multiple fields for
compound
        ordering key are supported (first item is most significant).

        This is an informational only event type attribute for
specification of
        application level ordering. Nakadi transportation layer is not
affected,
        where events are delivered to consumers in the order they were
published.

        Scope of the ordering is all events (of all partitions), unless it
is
        restricted to data instance scope in combination with
        `ordering_instance_ids` attribute below.

        This field can be modified at any moment, but event type owners
are
        expected to notify consumer in advance about the change.

        *Background:* Event ordering is often created on application level
using
        ascending counters, and data providers/consumers do not need to
rely on the
        event publication order. A typical example are data instance
change events
        used to keep a slave data store replica in sync. Here you have an
order
        defined per instance using data object change counters (aka row
update
        version) and the order of event publication is not relevant,
because
        consumers for data synchronization skip older instance versions
when they
        reconstruct the data object replica state.

      items:
        type: string
```

```
        description: |
          Indicates a single ordering field. This is a JsonPointer, which
is applied
          onto the whole event object, including the contained metadata
and data (in
          case of a data change event) objects. It must point to a field
of type
          string or number/integer (as for those the ordering is obvious).

          Indicates a single ordering field. It is a simple path (dot
separated) to
          the JSON leaf element of the whole event object, including the
contained metadata and data (in
          case of a data change event) objects. It must point to a field
of type
          string or number/integer (as for those the ordering is obvious),
and must be
          present in the schema.
        example: "data.order_change_counter"
    ordering_instance_ids:
      type: array
      description: |
        Indicates which field represents the data instance identifier and
scope in
        which ordering_key_fields provides a strict order. It is typically
a single
        field, but multiple fields for compound identifier keys are also
supported.

        This is an informational only event type attribute without
specific Nakadi
        semantics for specification of application level ordering. It only
can be
        used in combination with `ordering_key_fields`.

        This field can be modified at any moment, but event type owners
are expected
        to notify consumer in advance about the change.
      items:
        type: string
        description: |
          Indicates a single key field. It is a simple path (dot
separated) to the JSON
          leaf element of the whole event object, including the contained
metadata and
          data (in case of a data change event) objects, and it must be
present in the
          schema.
       example: "data.order_number"
```

```
    created_at:
      description: イベント型が新規作成された日時
      type: string
      pattern: date-time
    updated_at:
      description: イベント型の最終更新日時
      type: string
      pattern: date-time
```

イベント型をサポートしているレジストリのようなAPIは、サポートされたカテゴリやスキーマ形式の 集合を含んだモデルを拡張しているかもしれません。 例えばNakadi APIのイベントカテゴリレジストリは、イベントのバリデーションの宣言や 強化戦略、ストリームの中でどうパーティショニングされるかのような補足情報が 記述できるようになっています。(**SHOULD** データ変更イベントにはハッシュパーティション戦略を使う 参照)

MUST イベントが周知のイベントカテゴリに準拠することを保証する

イベントカテゴリ はイベント型の一般的な分類です。 ガイドラインは2つのカテゴリを定義します。

- 汎用イベント: 汎用目的のカテゴリ
- データ更新イベント: データ統合に基づくレプリケーションに使用されるデータの変更について記述するカテゴリ

カテゴリは将来的に成長していくことが予想されます。

カテゴリとは、イベント送信者が準拠しなくてはならないイベントの種類(データ更新イベントなど) に関しての標準を、事前に定義した構造で記述したものです。

汎用イベントカテゴリ

汎用イベントカテゴリ は、Open API スキーマオブジェクトの定義として、 以下のような構造で表せます。

```
GeneralEvent:
  description: |
    汎用目的のイベントの種類です。このイベントに基づくイベントの種類は、
    ドキュメントのトップレベルとして、カスタムスキーマペイロードを定義します。
    ペイロードには、"metadata" フィールドが必要です。
    したがって、このイベント型に基づくイベントのインスタンスは、
EventMetadataの定義と、
    カスタムスキーマ定義の両方に準拠することになります。
    以前はこのカテゴリは、業務カテゴリと呼ばれていました。
  required:
    - metadata
  properties:
    metadata:
        $ref: '#/definitions/EventMetadata'
```

汎用イベントカテゴリに属するイベント型は、ドキュメントのトップレベルに 標準情報のための予約されている metadata フィールドを使って、 カスタムスキーマのペイロードを定義します。 (metadata の内容は、このセクションのずっと下の方に記述してあります)

注意:

- 以前のガイドラインでは、汎用イベントは *業務イベント* と呼んでいた。 カテゴリの構造が、他の種類のイベントでも使われるようになったので、 チームの使い方を反映して名前を変更した。
- 汎用イベントは元の業務プロセスを駆動するイベントを定義する目的でも、今でも有用だし、そういう使い方にはおすすめする。
- Nakadiのブローカーは、汎用カテゴリを業務カテゴリとして参照し、イベント型は「business」というキーワードで登録される。それ以外のJSONの構造は同じである。

カテゴリの使い方に関するガイドは **MUST** 業務プロセスのステップと到達点を通知するために、汎用イベントカテゴリを使う により詳細があります。

データ更新イベントカテゴリ

データ更新イベントカテゴリ は、Open API スキーマオブジェクトの定義として、 以下のような構造で表せます。

```yaml
DataChangeEvent:
  description: |
    エンティティの変更の表現です。必須フィールドは、送信者によって送られることが
    期待され、そうでないフィールドはpub/subブローカのような仲介者によって、
    付加される可能性があります。 イベント型に基づくイベントのインスタンスは、
    DataChangeEventの定義とカスタムスキーマ定義の両方に準拠します。
  required:
    - metadata
    - data_op
    - data_type
    - data
  properties:
    metadata:
      description: このイベントのメタデータです。
      $ref: '#/definitions/EventMetadata'
    data:
      description: |
        イベント型のカスタムペイロードを含みます。ペイロードは、メタデータオブジェクトの
        `event_type`
フィールドに宣言されたイベント型と関連したスキーマに準拠しなければ
        なりません。
      type: object
    data_type:
      description: 変更された(業務)データエンティティの名前です。
      type: string
      example: 'sales_order.order'
    data_op:
      type: string
      enum: ['C', 'U', 'D', 'S']
      description: |
        エンティティに対して実行した操作の種類です。
        - C: エンティティの新規作成
        - U: エンティティの更新
        - D: エンティティの削除
        - S: ある時点でのエンティティのスナップショット作成
```

データ更新イベントカテゴリは、構造的に汎用イベントカテゴリとは異なります。 `data` フィールドでカスタムペイロードを定義し、 `data_type` にデータ変更に関する 固有の情報を定義します。 例えば次の例では、 `a` と `b` のフィールドは、 `data` フィールドの内側に おかれたカスタムペイロードの一部です。

データ更新イベントカテゴリの使い方の指針は、以下のガイドラインも参照ください。

- **SHOULD** データ変更イベントがAPI表現にマッチすることを保証する
- **MUST** 変化を通知するためにデータ変更イベントを使う
- **SHOULD** データ変更イベントにはハッシュパーティション戦略を使う

汎用カテゴリもデータ更新イベントカテゴリも、 メタデータ に関しては、 共通の構造をもちます。 メタデー

タの構造は、Open APIスキーマオブジェクトとして以下のように表せます。

```
EventMetadata:
  type: object
  description: |
    Carries metadata for an Event along with common fields. The required
    fields are those expected to be sent by the producer, other fields may
be
    added by intermediaries such as publish/subscribe broker.
  required:
    - eid
    - occurred_at
  properties:
    eid:
      description: このイベントの識別子です。
      type: string
      format: uuid
      example: '105a76d8-db49-4144-ace7-e683e8f4ba46'
    event_type:
      description: このイベントのEventTypeの名前です
      type: string
      example: 'example.important-business-event'
    occurred_at:
      description: イベントが送信者によって作成された日時
      type: string
      format: date-time
      example: '1996-12-19T16:39:57-08:00'
    received_at:
      description: |
        ブローカのような仲介者にイベントが届いた日時
      type: string
      readOnly: true
      format: date-time
      example: '1996-12-19T16:39:57-08:00'
    version:
      description: |
        このイベントをバリデーションするのに使われるスキーマのバージョンです。
        これは仲介者によって。This may be enriched upon reception by
intermediaries.
        この文字列にはセマンティックバージョニングが使われます。
      type: string
      readOnly: true
    parent_eids:
      description: |
        このイベントが生成される原因となったイベントの識別子です。
        イベント送信者がセットします。
      type: array
      items:
        type: string
```

```
        format: uuid
      example: '105a76d8-db49-4144-ace7-e683e8f4ba46'
    flow_id:
      description: |
        (X-Flow-Id HTTPヘッダと対応した) このイベントのflow-idです。
      type: string
      example: 'JAh6xH4OQhCJ9PutIV_RYw'
    partition:
      description: |
        このイベントに割り当てられたパーティションを示します。
        あるイベント型のイベントがパーティションに分割されるシステムで使わます。
      type: string
      example: '0'
```

イベントの送信者と、その最終的な受信者の間で、イベントのバリデーションやイベントの metadata を充実させるような操作がなされる可能性があることに注意してください。 例えばNakadiのようなブローカは、バリデーションしたり、任意のフィールドを追加したり、 あるフィールドが与えられていなければ、デフォルト値などをセットしたりできます。 そんなシステムがどう動くかは、このガイドラインのスコープ外ですが、イベント送信者と受信者が、 それを扱わなくてはならないので、追加の情報をドキュメントに書いておくべきです。

MUST イベントに有用な業務リソースを定義していることを保証する

イベントは業務プロセス/データの分析・モニタリングを含む他のサービスによって 使われることを想定しています。 したがって、サービスドメインのために定義されたリソースや業務プロセスに基づくものであるべきだし、 業務の自然なライフサイクルに即したものであるべきです (**SHOULD** 完全な業務プロセスをモデル化する および **SHOULD** *有用な* リソースを定義する を参照)。

イベント型やトピックスを大量に作るのはコストがかかるので、複数のユースケースで使えるような 抽象的/汎用的なイベント型を定義するようにしましょう。そして、明確なニーズがない限りは イベント型を公開するのは避けましょう。

MUST イベントにカスタマの個人情報データを載せてはならない

APIの権限スコープと同様に、近い将来イベント型の権限もOAuthトークンで渡せるようになるでしょう。 それまでは、次の注意事項にしたがうようにしてください。

- (Eメールアドレス、電話番号などの)機微な情報は、厳重なアクセス管理とデータ保護がされなければならない
- イベント型のオーナーは、それが必須か任意かによらず、機微な情報を公開 **してはならない** 。 例えばイベントが(他のAPIと同様) 注文配送の送付先住所のような個人情報を扱う必要が時々あるが、 これは問題ない。

MUST 業務プロセスのステップと到達点を通知するために、汎用イベントカテゴリを使う

イベントが業務プロセスにおけるステップを表現したものであるならば、 イベント型は汎用イベントカテゴリのものでなくてはならなりません。

単一の業務プロセスにまつわるすべてのイベントは、次のルールを遵守してください。

- 業務イベントには、業務プロセスの実行にあたり全てのイベントを効率的に集約するために、 特定の識別子フィールド (業務プロセスID または "bp-id") を含める。 flow-idと同様。
- 業務イベントには業務プロセス実行にあたり、正しくイベントを順序付けするための方法を含める。 (時系列性を信頼して良い正確なタイムスタンプのような) 単調増加する値が得られないような 分散環境においては、`parent_eids` データがイベント間の因果関係を表すものとて使える。
- 業務イベントは特定のステップ/到達点にて、業務プロセスの実行に対して、新しい情報のみ を含んでいるべきである。
- それぞれの業務プロセスシーケンスは、すべての関連するコンテキスト情報を含んだ 業務イベントによって開始されるべきである。
- 業務イベントは、サービスによって確実に送信されなければならない。

単一のイベント型を使い、状態フィールドで特定のステップを表現する業務業務プロセスのイベント すべてを公開するのがよいのかどうか、 各ステップを表現するために複数のイベント型を使ったほうがよいのかどうか、 現時点では何がベストプラクティスか私たちには分かりません。 与えられた業務プロセスについて、今は私たちはそれぞれの選択肢を評価し、その1つにこだわって みようと、そう考えているのです。

MUST 変化を通知するためにデータ変更イベントを使う

データの作成、更新、削除を表すイベントを送出するとき、イベント型はデータ変更イベントカテゴリ のものでなくてはなりません。

- 変更イベントは、あるエンティティに関連するすべてのイベントを集約できるよう 変更されたエンティティを識別できなくてはなりません。
- 変更イベントは **SHOULD** 明示的にイベントを順序付けする方法を与える
- 変更イベントはサービスによって確実に送信されなければならない。

SHOULD 明示的にイベントを順序付けする方法を与える

エラーが発生した場合、イベントストリームを再構成したり、 ストリームの中での位置からイベントを再現したりすることを、イベント受信者に要求することが あります。 それゆえにイベントは、部分的な発生順を再現できる方法を含んで *いなければなりません* 。

これは、(例えばデータベースで作成する) エンティティのバージョンやメッセージカウンタを 使って実現します。これらは厳密かつ単調に増加する値を使います。

システムタイムスタンプを使うのはあまり良い選択ではありません。分散システムにおいて 正確な時刻同期

は困難だし、2つのイベントが同じマイクロ秒で発生するかもしれないし、
またシステムクロックは、時刻合わせのドリフトやうるう秒で前後する可能性もあるためです。 もしイベントの順番を表すのにシステムタイムスタンプを使えば、設計したイベント順がこれらの影響で 混乱を来さないことを注意深く保証しなければなりません。

分散環境でデータ構造によってこの問題を解消する (CRDTs, logical clocks や vector clocks のような) 仕組みはこのガイドラインのスコープ外であることに **注意** してください。

SHOULD データ変更イベントにはハッシュパーティション戦略を使う

hash パーティション戦略は、イベントが追加されるべき論理パーティションを 計算するためのインプットとして使われるフィールドを、イベント送信者は定義できます。 イベントエンティティの順序をパーティションローカルで決められる間は、スループットを スケールできるようになります。

hash オプションは、特にデータ変更に有用です。それによって、あるエンティティに関連するすべてのイベントを、 パーティションへ一貫性をもって割り当てることができるし、そのエンティティに関する 順序付けされたイベントストリームを提供できるようになるからです。 これは各パーティションが全順序性をもつならば、パーティションをまたいだ順序が サポートするシステムでは保証されないので、パーティションをまたいで送信されたイベントは、 サーバに到着したのとは異なる順序で、イベント受信者に見える可能性があることを示しています。

hash 戦略を使うとき、ほとんどすべての場合、パーティションキーは変更されるエンティティを 表すものであり、 `eid` フィールドやタイムスタンプのようなイベント毎に付与されたり、 変更識別子だったりするものではありません。 これによって、データの変更イベントが、同じエンティティでは同じパーティションに入ることが保証され、 クライアントは効率的にイベントを受信できるようになる。

データ変更イベントが、送信者側が定義したり、ランダムに選択したりと、 独自のパーティション戦略をもつ例外的な場合があるかもしれませんが、 一般的にいって、 ハッシュ が正しい選択しです。 ここでのガイドラインは "should" ですが、"すごくイカした理由がない限りは、must" と読み替えてください。

SHOULD データ変更イベントがAPI表現にマッチすることを保証する

データ変更イベントのエンティティ表現は、REST APIの表現と対応しているべきです。

あるサービスにとって最小限の構造しか持たないようにすることに価値があります。 そうすれば、サービスの利用者にとってはより少ない表現しか使わずにすむし、 サービスオーナーにとっては、保守しなくてはならないAPIが少なくてすみます。 特に、そのドメインに関連していて、実装やローカルの詳細から切り離され抽象化されたイベントのみ 公開するようにすべきです。 システム内で起こるすべての変更を反映する必要はありません。

APIリソース表現と直接関係のないデータ変更イベントを定義する意義がある場合もあります。例えば次のような場合です。

- APIリソース表現がデータストア表現とかなり乖離があるが、物理的なデータの方がデータ統合のための確実に処理するのがより簡単である。

- 集約されたデータの送信。例えば個々のエンティティへの変更データが、 APIのために定義されたものよりも、粒度のあらい表現を含んだイベントが送出されるかもしれない。
- マッチングアルゴリズムのような計算結果や大量に生成されたデータで、 サービスによってエンティティとして保存しないかもしれないイベント。

MUST イベントの権限はAPIの権限に対応しなければならない

リソースがREST APIを通じて同期的に読み取りアクセスでき、イベントを通じて非同期で読み取りアクセスできると すると、同じ読み取り権限が適用されていなければならない。 私たちはデータを保護したいのあって、データのアクセス方法を保護したい訳ではないのだから。

MUST イベント型のオーナーを明示する

イベント定義は、所有者をハッキリさせておかなければなりません。EventTypeの `owning_application` で明示します。

EventTypeのオーナーでその定義に責任をもつのは、1つの送信アプリケーションであることが 多いですが、そのオーナは同種のイベントを送信する複数のサービスの1つであってもよいです。

MUST 全体のガイドラインにしたがってイベントのペイロードを定義する

イベントは他のAPIデータやAPIガイドラインと整合性のとれたものでなくてはなりません。

はじめに で表したすべてが、サービス間でデータをやり取りするイベントに適用されます。 APIと同様にイベントは、私たちのシステムが何をしているのかを表現するための責務を果たし、 高品質に設計された有用なイベントが、私たちの新しく面白いプロダクトやサービス開発を支えるのです。

イベントが他の種類のデータと異なるのは、非同期のpub-subメッセージングのように、 データの伝達に使われるところにあります。だからといって、 例えば検索リクエストやページ分割されたフィードのように、REST APIを使うような ところでイベントが使えない訳ではありません。 サービスのREST APIのために作ったモデルを、イベントでもベースとすることになるでしょう。

次のガイドラインの章がイベントにも適用されます。

- 全般にわたるガイドライン
- APIの命名
- データフォーマット
- 共通のデータ型
- ハイパーメディア

MUST イベントのために後方互換性を維持する

イベントの変更は項目追加や後方互換のある変更を基本としなければなりません。 これは 互換性 ガイドラインの「Must: 後方互換性を崩してはならない」 にしたがうものです。

イベントの文脈では、互換性の事情は複雑です。 イベントの送信者も受信者も高度に非同期化されていて、RESTのクライアント/サーバでは適用できていた content-negotiation を用いた テクニックは使えないためです。 これは後方互換維持のためのより高いハードルを、受信者側に課すことになります。 要求に応じてバージョニングしたメディアタイプを返すということが出来ないためです。

イベントスキーマでは、受信者側から見たときに、以下のものは後方互換性があると 考えられます。

- JSONオブジェクトへの新しい任意のフィールドの追加
- フィールドの並び順の変更 (オブジェクトにおけるフィールドの並びは任意である)
- 配列内の同じ型の値の並び順変更
- 任意のフィールドの削除
- 列挙型の個々の値の削除

また、受信者側から見たときに、以下のものは後方互換性がないと考えられます。

- JSONオブジェクトから必須のフィールドの削除
- フィールドのデフォルト値の変更
- フィールド、オブジェクト、列挙型、配列の型の変更
- 配列内の異なる型の値の並び順変更 (こういった配列はタプルとして知られている)
- 既存のフィールドを再定義した新しい任意のフィールドの追加 (共起制限として知られている)
- 列挙型への値の追加 (`x-extensible-enum` はJSONスキーマでは使えないことに注意)

SHOULD イベント型定義では `additionalProperties` を避ける

イベント型のスキーマでは、スキーマの成長をサポートするため `additionalProperties` の使用を避けるべきです。

イベントはpub-subシステムによって中継されることが多く、共通的にログがとられたり、 後で読み込むためにストレージに保存されたりします。 特に受信者と送信者双方で使われるスキーマは、時間とともに変化していきます。 結果として、クライアント・サーバ型のAPIではあまり起こらなかった互換性と拡張性の問題が、 イベントの設計では重要かつふつうに考えなきゃならいことになってくるのです。 イベントスキーマの成長を可能にするため、ガイドラインは次の点を推奨します。

- イベント送信者は後方互換性を維持し安全にスキーマを修正できるよう、 `additionalProperties` を `true` (つまりワイルドカードの拡張ポイントを意味する) で宣言 **してはならない** 。 かわりに新しい任意のフィールドを定義し、安これらのフィールドを公開する前に、スキーマを更新しなければならない。

- イベント受信者は自分が処理できないフィールドは無視し、エラーを発生させては **いけない** 。 これは送信者によって指定された新しい定義を含むものよりも、古いイベントスキーマが適用されたイベントを 処理しなければならないときに発生する。

上記制約は、イベント型スキーマの将来のリビジョンで、フィールドが追加できないことを意味してはいません。 イベント型の新しいスキーマが、イベント送信前に前にまずフィールドを定義していれば、 互換性のある追加で許されたオペレーションです。 同じ順番で、受信者はAPIクライアントと同様に、スキーマのコピーに情報のないフィールドを無視しなければなりません。 すなわち、 イベント型スキーマが拡張に対して閉じていたとしても `additionalProperties` フィールドがないことを扱うことができないのです。

_フィールド再定義 _ の問題を避けるため、イベント送信者にイベント送信する前に、 フィールドを定義すること要求します。 これはイベント送信者が、既に送出された異なる型のイベントにフィールドを定義したり、未定義のフィールドの型を変更したりしている場合です。 どちらも、 `additionalProperties` を使わないことで防げます。

`additionalProperties` の使用についてのガイドラインは、 互換性 の章のルール **MUST** Open APIの定義をデフォルトで拡張に対して開かれているものとして扱う を参照ください。

MUST ユニークなイベント識別子を使う

イベントの `eid` (イベント識別子)の値は、ユニークでなくてはなりません。

`eid` プロパティは、イベントの標準の metadata の一部であり、 イベントに識別子を与えるものです。 送信クライアントは、イベント送出時にこれを生成し、所有アプリケーションの範囲で ユニーク性を保証しなければなりません。 特に、あるイベント型のストリームをともなうイベントは、ユニークな識別子はマストです。 これはイベント受信者が、 `eid` をイベントがユニークであるとして処理したり、 冪等性のチェックに使ったりするためです。

イベントを受信するシステムが `eid` のユニーク性のチェックすることは任意であるので、 送信者側がイベント識別子のユニーク性を保証する責務があることに注意しましょう。 イベントのユニーク識別子を生成する単純な方法は、UUIDを使うことです。

SHOULD 冪等な順不同の処理を設計する

冪等 で順不同の処理をするものとしてイベントを設計しておくと、 非常にレジリエントなシステムとなります。もしイベントの処理に失敗しても、 送信者と受信者は、処理を一時停止したり、処理結果の整合性を崩すことなく、 イベント処理をスキップしたりディレイさせたりリトライしたりできます。

このように処理順を自由にするには、冪等で順不同な処理設計を明示的にやる必要があります。 イベントが元の順序を推測するのに十分な情報を含むようにしたり、業務ドメインが 順序性によらないような方法で設計するようにします。

データ変更イベントと似た共通の例として、冪等で順不同な処理は、次の情報を送る ことによって達成されます。

- プロセス/リソース/エンティティの識別子
- 単調増加する順序付けられたキー

- 変更後のプロセス/リソースの状態

受信側が現在の状態にだけ関心があるのであれば、各リソースの最新イベントよりも古いものは 無視できます。 受信側がリソースの履歴にも関心があるのであれば、(部分的にでも) 順序性のある一連のイベントを 再生成するために、順番に並んだキーを使います。

MUST イベント型の名前は命名規約にしたがう

イベント型の名前は、次に示すとおり オーディエンス に依存した機能本位の命名に準拠しなければなりません。 (またはそうするべきです。 **MUST/SHOULD** 機能本位の命名体系を使う に詳細と定義があります)

```
<event-type-name>       ::= <functional-event-name> | <application-event-
name>

<functional-event-name> ::= <functional-name>.<event-name>

<event-name>            ::= [a-z][a-z0-9-]* -- 自由なイベント名 (
機能を表す名前)
```

次のアプリケーション固有のレガシーな規約は、 内部 イベント型名に ***のみ*** 適用するようにしてください。

```
<application-event-name> ::= [<organization-id>.]<application-id>.<event-
name>
<organization-id>  ::= [a-z][a-z0-9-]* -- 組織の識別子 (例えばチームIDのような)
<application-id>   ::= [a-z][a-z0-9-]* -- アプリケーションの識別子
```

注意: 同じエンティティをデータ変更イベントとRESTful APIの両方で扱うときは、 一貫性のある名前を使うようにしましょう。

MUST 重複したイベントに備える

イベントの受け手は、重複したイベントを正しく処理できなくてはなりません。

大抵のメッセージブローカとデータストリーミングシステムは、"at-least-once"配信をサポートしています。これはある特定のイベントが、必ず1回以上は受け手に届くことを保証するものです。 別の状況でも、重複したイベントが発生する可能性があります。

例えば、イベントの送信者が(ネットワークの問題によって) 受け手に届かなかったような 状況で発生します。この場合、送信者は同じイベントの再送を試みます。 こうしてイベントバスに受信者が処理すべき同一のイベントが2つ存在することになります。 同じ状態は受信者側でも起こります: イベントは正しく処理したが、その処理が確認出来ない場合です。

Appendix A: リファレンス

私たちが参考にし、ガイドラインのベースにした文書へのリンク集です。

OpenAPI 仕様

- **Open API仕様**
- **Open API仕様のマインドマップ**

標準仕様

- **RFC 3339:** Date and Time on the Internet: Timestamps
- **RFC 4122:** A Universally Unique IDentifier (UUID) URN Namespace
- **RFC 4627:** The application/json Media Type for JavaScript Object Notation (JSON)
- **RFC 8288:** Web Linking
- **RFC 6585:** Additional HTTP Status Codes
- **RFC 6902:** JavaScript Object Notation (JSON) Patch
- **RFC 7159:** The JavaScript Object Notation (JSON) Data Interchange Format
- **RFC 7230:** Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing
- **RFC 7231:** Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content
- **RFC 7232:** Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests
- **RFC 7233:** Hypertext Transfer Protocol (HTTP/1.1): Range Requests
- **RFC 7234:** Hypertext Transfer Protocol (HTTP/1.1): Caching
- **RFC 7240:** Prefer Header for HTTP
- **RFC 7396:** JSON Merge Patch
- **RFC 7807:** Problem Details for HTTP APIs
- **RFC 4648:** The Base16, Base32, and Base64 Data Encodings
- **ISO 8601:** Date and time format
- **ISO 3166-1 alpha-2:** Two letter country codes
- **ISO 639-1:** Two letter language codes
- **ISO 4217:** Currency codes
- **BCP 47:** Tags for Identifying Languages

論文

- Roy Thomas Fielding - Architectural Styles and the Design of Network-Based Software Architectures. これはRESTとは何かについて書かれたものです。

書籍

- **REST in Practice: Hypermedia and Systems Architecture**
- **Build APIs You Won't Hate**
- **InfoQ eBook - Web APIs: From Start to Finish**

ブログ

- **Lessons-learned blog: Thoughts on RESTful API Design**

Appendix B: ツール

これはガイドラインの一部ではありませんが、これらにしたがうのは役に立つかもしれません。 ここで挙げられているツールを使っても、自動的にガイドラインにしたがったことにはなりません。

APIファーストインテグレーション

以下のフレームワークは、OpenAPI YAMLファイルを使って、特にAPIファーストをサポートするよう設計されています。

- **Connexion:** Flask上に構築されたPythonのOpenAPIファーストのフレームワーク
- **Friboo:** SwaggerとOAuthをサポートするClojureでマイクロサービスを書くためのユーティリティライブラリ
- **Api-First-Hand:** API-First Play Bootstrapping Tool for Swagger/OpenAPI specs
- **Swagger Codegen:** template-driven engine to generate client code in different languages by parsing Swagger Resource Declaration
- **Swagger Codegen Tooling:** plugin for Maven that generates pieces of code from OpenAPI specification
- **Intellij IDEA の Swagger プラグイン:** IntelliJ IDEAでSwaggerファイルを編集するためのプラグイン

Swagger/OpenAPI のホームページにも一覧があります Community-Driven Language Integrations が、それらのほとんどがAPIファーストのアプローチには合わないものです。

サポートライブラリ

これらのユーティリティライブラリは、私たちのAPIガイドラインを様々な部分の実装を手助けしてくれるものたちです。(アルファベット順)

- **Problem:** Java library that implements application/problem+json
- **Problems for Spring Web MVC:** library for handling Problems in Spring Web MVC
- **Jackson Datatype Money:** extension module to properly support datatypes of javax.money
- **Tracer:** call tracing and log correlation in distributed systems
- **TWINTIP Spring Integration:** API discovery endpoint for Spring Web MVC

Appendix C: ベストプラクティス

実際のガイドラインの一部ではないけれど、RESTful APIの実装で直面する共通の課題に光明をもたらすべく、ベストプラクティスをこのセクションにまとめます。

RESTful APIにおける楽観ロック

はじめに

楽観ロックは同一エンティティに同時に書き込みが発生し、データが整合性が失われることを防ぐのに使われます。 クライアントは常に最初にエンティティのコピーを取ってきて、これを更新しなければなりません。もしその間に別のバージョンが作られたら、更新は失敗すべきです。これがうまくいくように、更新を実行する前に、クライアントはサービスによってチェックされたバージョンの参照の種類を提供しなければなりません。

詳しくはPUTメソッドの使用法についてのセクションをみてください。PUTを用いたリソースの更新方法についてより詳細に記述しています。

RESTful APIは通常、エンティティのリストを返すような検索のエンドポイントをもちます。更新すべきエンティティの現在のバージョンを取得するため使われる検索エンドポイントと組み合わせて、楽観ロックを実装する方法はいくつかあります。

If-MatchヘッダとETagヘッダ

ETagヘッダは更新前に単一のエンティティリソースに対するGETリクエストを実行することによって取得できます。つまり、検索エンドポイントを使う際には、追加のリクエストが必要ということです。

例:

```
< GET /orders

> HTTP/1.1 200 OK
> {
>   "items": [
>     { id: "O0000042"},
>     { id: "O0000043"}
>   ]
> }

< GET /orders/BO0000042

> HTTP/1.1 200 OK
> ETag: osjnfkjbnkq3jlnksjnvkjlsbf
> { id: "BO0000042", ... }

< PUT /orders/O0000042
< If-Match: osjnfkjbnkq3jlnksjnvkjlsbf
< { id: "O0000042", ... }

> HTTP/1.1 204 No Content

エンティティのETagが、既に更新されて一致しなかったら、以下のレスポンスになります。

> HTTP/1.1 412 Precondition failed
```

Pros

- RESTfulな解決手段です

Cons

- 多くの追加リクエストが必要になってしまう。

結果エンティティにおける ETag

すべてのエンティティに、追加のプロパティとしてETagを付けて返します。 複数のエンティティを含むレスポンスでは、エンティティそれぞれに後続のPUTで使用可能な異なるETagが付与されます。

例:

```
< GET /orders

> HTTP/1.1 200 OK
> {
>   "items": [
>     { id: "O0000042", etag: "osjnfkjbnkq3jlnksjnvkjlsbf", ... },
>     { id: "O0000043", etag: "kjshdfknjqlöwjdsljdnfkjbkn", ... }
>   ]
> }

< PUT /orders/O0000042
< If-Match: osjnfkjbnkq3jlnksjnvkjlsbf
< { id: "O0000042", ... }

> HTTP/1.1 204 No Content

GETの後の更新で、エンティティのETagが変わってしまっていたら、以下のレスポンスが返ります
。

> HTTP/1.1 412 Precondition failed
```

Pros

- パーフェクトな楽観ロックである

Cons

- HTTPヘッダに付与すべき情報が、業務オブジェクトに入り込んでしまっている。

バージョン番号

バージョン番号をエンティティのプロパティに含む方法です。 PUTがリクエストがされたとき、ペイロードに含まれたバージョン番号を、サーバはデータベース中のバージョン番号と突き合わせします。

例:

```
< GET /orders

> HTTP/1.1 200 OK
> {
>   "items": [
>     { id: "O0000042", version: 1, ... },
>     { id: "O0000043", version: 42, ... }
>   ]
> }

< PUT /orders/O0000042
< { id: "O0000042", version: 1, ... }


> HTTP/1.1 204 No Content
```

GETのあと別リクエストで更新されていたら、データベース中のバージョン番号はリクエストボディで与えられたものより、大きな値になっているので、409を返します。

```
> HTTP/1.1 409 Conflict
```

Pros

- パーフェクトな楽観ロックである

Cons

- HTTPヘッダで実現すべき機能が、業務オブジェクトに入り込んでしまっている。

Last-Modified / If-Unmodified-Since

HTTP1.0では、ETagの仕様はなく、楽観ロックには日時に基づいた手法が使われていました。 これは現在でもHTTPプロトコルの一部であり利用できます。

すべてのレスポンスには、HTTP dateを値にもつLast-Modifiedヘッダが含ませます。 PUTリクエストを使った更新をリクエストするとき、クライアントはIf-Unmoified-Since ヘッダを使って、 Last-Modifiedで受け取った値をセットします。 サーバはもしエンティティの最終更新日時が、ヘッダの日時よりも後であれば、このリクエストを拒否します。

GETとPUTの間で発生した変更が上書きされるような状況を効果的に検出できます。 複数の結果エンティティの場合、Last-Modifiedヘッダには、すべてのエンティティの最終更新日時うち最新のものがセットされるでしょう。 これはGETとPUTの間で発生するエンティティのどんな変更も、コンフリクトが検出可能で、バッチの残りをロックすることなく行えることを保証します。

Example:

```
< GET /orders

> HTTP/1.1 200 OK
> Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
> {
>   "items": [
>     { id: "O0000042", ... },
>     { id: "O0000043", ... }
>   ]
> }

< PUT /block/O0000042
< If-Unmodified-Since: Wed, 22 Jul 2009 19:15:56 GMT
< { id: "O0000042", ... }

> HTTP/1.1 204 No Content
```

GETのあと更新され、エンティティの最終更新日時が与えられた日時よりも後であれば、412を返します。

```
[source,http]
> HTTP/1.1 412 Precondition failed
```

Pros

- 昔から使われてきた方法で枯れている。
- 業務オブジェクトに干渉しない。HTTPヘッダのみと使ってロックできる。
- 実装がとても簡単である
- 検索エンドポイントの結果のエンティティを更新するとき、更新リクエスト以外の追加ののリクエストは必要ない。

Cons

- もしクライアントが異なる2つのインスタンスと通信している場合、その時刻同期が完全にできていないと、 ロックは失敗する可能性がある。

結論

私たちは、*`Last-Modified`*/*`If-Unmodified-Since`* か *結果エンティティの`ETag`* のどちらかを 使うことをおすすめします。

Appendix D: 変更履歴

この変更履歴は2016年10月以降の主要な変更のみ記載しています。

主要でない変更とは、表記上のみの修正や既存のガイドラインの軽微な修正(新しいエラーコードの追加など)

です。 主要な変更は、追加のルールにともなう変更、または既存のガイドラインのルール変更です。
後者の変更点を「ルールの変更点」というラベルを付けてまとめました。 すべての変更を知りたくば, Github のコミット を参照ください。

Rule Changes

- `2019-01-24`: Improve guidance on caching (**MUST** メソッド毎に共通の性質を満たす, **MUST** キャッシュ可能な `GET`, `HEAD`, `POST` エンドポイントをドキュメント化する).
- `2019-01-15`: Improve guidance on idempotency, introduce idempotency-key (**SHOULD** `POST` と `PATCH` の冪等設計を検討する, **SHOULD** 冪等な `POST` 設計のためにセカンダリキーを使う).
- `2018-01-10`: Moved meta information related aspects into new chapter メタ情報.
- `2018-01-09`: Changed publication requirements for API specifications (**MUST** Open API仕様を公開する).
- `2017-12-07`: Added best practices section including discussion about optimistic locking approaches.
- `2017-11-28`: Changed OAuth flow example from password to client credentials in セキュリティ.
- `2017-11-22`: Updated description of X-Tenant-ID header field
- `2017-08-22`: Migration to Asciidoc
- `2017-07-20`: Be more precise on client vs. server obligations for compatible API extensions.
- `2017-06-06`: Made money object guideline clearer.
- `2017-05-17`: Added guideline on query parameter collection format.
- `2017-05-10`: Added the convention of using RFC2119 to describe guideline levels, and replaced `book.could` with `book.may`.
- `2017-03-30`: Added rule that permissions on resources in events must correspond to permissions on API resources
- `2017-03-30`: Added rule that APIs should be modelled around business processes
- `2017-02-28`: Extended information about how to reference sub-resources and the usage of composite identifiers in the **MUST** パスセグメントによってリソースとサブリソースを識別できるようにする part.
- `2017-02-22`: Added guidance for conditional requests with If-Match/If-None-Match
- `2017-02-02`: Added guideline for batch and bulk request
- `2017-02-01`: **SHOULD** `Content-Location` の代わりに `Location` ヘッダを使う
- `2017-01-18`: Removed "Avoid Javascript Keywords" rule
- `2017-01-05`: Clarification on the usage of the term "REST/RESTful"
- `2016-12-07`: Introduced "API as a Product" principle

- `2016-12-06:` New guideline: "Should Only Use UUIDs If Necessary"
- `2016-12-04:` Changed OAuth flow example from implicit to password in セキュリティ.
- `2016-10-13:` **SHOULD** 標準のメディアタイプとして `application/json` を使う
- `2016-10-10:` Introduced the changelog. From now on all rule changes on API guidelines will be recorded here.

```
<!-- Adds rule id as a postfix to all rule titles -->
<script>
var ruleIdRegEx = /(\d)+/;
var h3headers = document.getElementsByTagName("h3");
for (var i = 0; i < h3headers.length; i++) {
    var header = h3headers[i];
    if (header.id && header.id.match(ruleIdRegEx)) {
        var a = header.getElementsByTagName("a")[0]
        a.innerHTML += " [" + header.id + "]";
    }
}
</script>

<!-- Add table of contents anchor and remove document title -->
<script>
var toc = document.getElementById('toc');
var div = document.createElement('div');
div.id = 'table-of-contents';
toc.parentNode.replaceChild(div, toc);
div.appendChild(toc);
var ul = toc.childNodes[3];
ul.removeChild(ul.childNodes[1]);
</script>

<!-- Adds sidebar navigation -->
<script>
var header = document.getElementById('header');
var nav = document.createElement('div');
nav.id = 'toc';
nav.classList.add('toc2');
var title = document.createElement('div');
title.id = 'toctitle';

var link = document.createElement('a');
link.innerText = 'API Guidelines';
link.href = '#';

title.append(link);
nav.append(title);
```

```
var ul = document.createElement('ul');
ul.classList.add('sectlevel1');

var link = document.createElement('a');
link.innerHTML = 'Table of Contents';
link.href = '#table-of-contents';
li = document.createElement('li');
li.append(link);
ul.append(li);

var link, li;
var h2headers = document.getElementsByTagName('h2');
for (var i = 1; i < h2headers.length; i++) {
    var a = h2headers[i].getElementsByTagName("a")[0];
    if (a !== undefined) {
        link = document.createElement('a');
        link.innerHTML = a.innerHTML;
        link.href = a.hash;
        li = document.createElement('li');
        li.append(link);
        ul.append(li);
    }
}

document.body.classList.add('toc2');
document.body.classList.add('toc-left');
nav.append(ul);
header.prepend(nav);
</script>

<!-- Global site tag (gtag.js) - Google Analytics -->
<script async src="https://www.googletagmanager.com/gtag/js?id=UA-
130687305-1"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag('js', new Date());

  gtag('config', 'UA-130687305-1');


(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(
){
  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),

m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore
(a,m)
  })(window,document,'script','https://www.google-
```

```
analytics.com/analytics.js','ga');
  ga('create', 'UA-130687305-1', 'auto');

  function trackPageview() {
    var title = (location.hash && location.hash.length > 0) ?
      document.getElementById(location.hash.replace('#','')).textContent :
      document.title;

    ga('send', 'pageview', {'page': location.pathname + location.hash,
'title': title});
  }

  if ("onhashchange" in window)
    window.onhashchange = trackPageview;

  trackPageview(); // track user's first pageview
</script>

<!-- Cookies Consent -->
<link rel="stylesheet" type="text/css"
href="https://cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookieco
nsent.min.css" />
<script
src="https://cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookiecon
sent.min.js"></script>
<script>
window.addEventListener("load", function(){
window.cookieconsent.initialise({
  "palette": {
    "popup": {
      "background": "#eaf7f7",
      "text": "#5c7291"
    },
    "button": {
      "background": "#56cbdb",
      "text": "#ffffff"
    }
  },
  "content": {
    "message": "This web site uses cookies to analyze the general behavior
of visitors."
  }
})});
</script>
```

[1] R.Fieldingの定義だと、REST APIはHATEOAS(レベル3)をサポート しなくてはなりません。 私たちのガイドラインは、完全なREST準拠はさほど推奨していません。 (ハイパーメディア)で示すような限定的なハイパーメディアの使い方をしています。 それでもなお"RESTful API"という言葉を私たちは使います。 他に確立された用語はないし、Webサービス業界ではRESTっぽいものをそう呼んでいるからです。 事実、HATEOASへの完全準拠したAPIは非常に数少ないのが現状です。

[2] HTTP/1.1 standard (RFC-7230) では2つのタイプのヘッダが定義されています。end-to-end と hop-by-hop です。 end-to-endヘッダは、リクエストまたはレスポンスの最終的な受け取り手まで 伝達しなければなりません。 一方、hop-by-hopヘッダは、単一のコネクションの間でしか有効でないものです。