REST API Design Guidelines

Table of Contents

The following contents are based on the Zalando Guidelines

Table of Contents

Motivation

Almost no Software works without interfaces to other systems - the API (Application Programming Interface) is often an elementary part of our daily work. Especially in the age of microservices and distributed systems, **high-quality APIs** are becoming more and more important. The central question is how to design and implement such an API. The answer to this question is often answered by each individual - according to their own knowledge and with their own references for best practices. So there is already a lot of knowledge on the topic of API design.

The idea behind the **unified** // **REST API Guidelines** described below is to consolidate the existing know-how from many minds and areas, and to enrich best practices from literature and third parties. The guidelines should help every developer to design and implement APIs according to a consistent pattern. Furthermore, they help you to integrate your API faster, because the guidelines

are already familiar and therefore the behavior is predictable. As a result, developers who are new to a project get in really fast, because the guidelines are uniformly designed. In addition, the guidelines can be brought to the customer as an additional // asset if required.

The guidelines are especially for **developers, software architects and IT consultants**.

What makes a good API for us at doubleSlash

- **Quick and easy to understand (approachability and learnability)**
  - built from a *user's* perspective - not based on internal model
  - discoverable
  - explicit
  - helpful feedback
- **Hard to use incorrectly (not error-prone)**
  - avoiding implicit user assumptions
  - preventing errors by validation
- **Stable but extensible**
  - complete for the *user's* use cases
  - minimal, no YAGNI
  - hiding implementation details
  - extensible without the need of versioning
- **Consistent**
  - self-consistent
  - consistent across the organisation
  - consistent with common practices & standards
- **Adaptable**
  - without getting too complex
  - keeping the user in control by allowing various flows instead of forcing specific order
- **Optimized**
  - reducing number of requests
  - reducing bandwidth (size of requests and responses)
  - caching
- **Secure**
  - avoiding to provide information that could be used for attacks
  - avoiding to expose sensitive information to unauthorized users
  - designed to prevent enumeration attacks

Structure - How to read

The // REST API Guidelines are divided into several sections:

First an introductory chapter describes the different maturity levels of HTTP APIs.

Then, the resources section covers the structure, naming and versioning of resources. The next section describes best practices for using resource parameters for sorting, filtering, selection, inclusion and pagination.

This is followed by the section error handling which describes a number of useful status codes and how an error message should be written. Next, it is shown how you can secure your API followed by best practices of how to document your API. In addition, these Guidelines describe, how you can improve the performance of your API using caching, compression and asynchronicity.

Subsequently, further topics and concepts like HATEOAS, GraphQL and OData are described. The guide also describes how to use actions as resources and gives advice how to check for existence of a resource.

Finally, examples of existing good REST APIs are shown, followed by references to sources and literature used in these guidelines.

In addition, DOs **[Good]** and DONTs **[Bad]** are highlighted in each section.

General/Introduction

Richardson Maturity Model

Leonard Richardson analyzed a hundred different web service designs and divided them into four categories based on how much they are REST compliant.

Level 0

Level zero of maturity does not make use of any URI, HTTP Methods, and HATEOAS capabilities. These services have a **single URI** and use a **single HTTP method** (typically POST).

Level 1

At level 1 of maturity the API can distinguish between different resources. This level uses **multiple URIs**, where every URI is the entry point to a specific resource. Still, this level uses only one **single HTTP method** like POST.

Level 2

At level 2, correct HTTP verbs are used with each request. It suggests that in order to be truly RESTful, HTTP verbs must be used in API. For each of those requests, the correct HTTP response code is provided. In other words, they are now **multiple URIs** and **multiple HTTP methods**

Level 3

Level 3 is the highest level. It is the combination of level 2 and HATEOAS.

Resources

Naming of Resources

Every good RESTful API is resource based. A resource is a representation of a domain entity for clients. Ideally, all resources represent the domain of the API. A resource should therefore fulfill the following points:

Name the resource as noun in plural

Resources should be expressed as nouns, because they represent professional entities. Avoid abbreviations and acronyms. By using the plural you also get a resource for collection queries.

Good

```
/chapters/3/sections/2/rules/1
```

Bad

```
/chapter/3/section/2/rule/1
```

Name the resource domain specific

As already mentioned several times, REST resources are functional entities. Therefore, it supports the understanding if resources are named specifically. Make sure that the spoken language in everyday project work is also reflected in the resources (Conway's law).

Good

```
/chapters/3/sections/2/rules/1
```

Bad

```
/levels/3/sub-levels/2/sub-sub-levels/1
```

Use spinal-case notation

Use spinal-case notation for resource names.

Good

```
/api-design-chapters/3/chapter-sections/2/section-rules/1
```

Bad

```
/api_design_chapters/3/chapter_sections/2/section_rules/1
/apiDesignChapters/3/chapterSections/2/sectionRules/1
```

Use sub-resources to represent relations between entities

Entities often have dependencies on each other. Such a dependency can be represented by nesting several resources (separated by /).

Good

```
/api-design-chapters/3/chapter-sections/2/section-rules/1
```

Bad

```
/chapter_sections-of-chapter1/2
/section-rules-of-section1/1
```

Identification of Resources

The very first step in designing a REST API based application is – identifying the objects which will be presented as resources. Resources should be defined to cover 90% of all its client's use cases. A useful resource should contain as much information as necessary, but as little as possible. A resource can be a collection or a singleton. For example, "customers" is a collection resource and "customer" a singleton resource. A collection resource can be identified using the Uniform Resource Name (URN) e.g:

```
/customers
```

A singleton resource can be identified using the URN e.g:

```
/customers/{customerId}
```

A resource may contain sub-collection resources. For example "accounts" of a particular "customer" can be identified using the URN:

```
/customers/{customerId}/accounts
```

A specific account for a particular customer can be identified via URN as follows:

```
/customers/{customerId}/accounts/{accountId}
```

Type of Identifier

For security and data protection reasons it is advisable that the identifier is not enumerable or easy to guess. It must not contain personal or private data (URLs appear in access logs, browser history,...). The type of identifier should not allow attackers or competitors to make inferences about the system (enumeration attacks). For practical reasons the identifier should not be subject to change.

Prefer UUIDs as identifiers (and as primary keys).
Consider the specific consequences carefully if you choose another identifier type taking into account enumeration attacks, data and privacy protection, web application firewalls.

Only use enumerable IDs if you can be certain that there is no risk of enumeration attacks in the specific context.

Common type of identifiers:

Identifier Type	Pros	Cons
UUID = *Universally Unique Identifier* Examples: • `/documents/f6c47559-f002-4af6-b7bb-2ddc4146e14d`	• Globally unique across systems • Can be generated independently • Suitable for distributed systems	• Generating large numbers of UUIDs may be inperformant due to lack of entropy • Higher storage and indexing requirements • Lengthy and less human-readable
Enumerable ID *e.g. consecutive integer numbers* Examples: • `/person/1`	• Simple, easy to work with • Efficient generation, indexing and querying in databases	• **Enumeration attacks, data protection / privacy issues** • Allows attackers or competitors to make inferences about the system • Simplifies getting access to private data

Identifier Type	Pros	Cons
Custom Codes *e.g. e-mail, username, ticket key, VIN, ISBN…* Examples: • `/users/max@mustermann.de` • `/users/mmustermann` • `/api/tickets/TKT-23456` • `/vehicles/WBAAV53471J297245` • `/books/978-9-6566-2511-9`	• Easy resource identification	• **Potentially data protection / privacy issues** • **Some data might be blocked by web application firewalls (WAF)** e.g. e-mail • May require additional validation logic • Maintaining uniqueness • Might change
Slugs *human-readable and URL-friendly strings derived from a resource's attributes* Examples: • `/blogs/my-great-article`	• Descriptive • URL-friendly • Improves SEO and user experience	• **Potentially data protection / privacy issues** • Uniqueness checks • Duplicate handling • Might change

Further reading:

- REST APIs: Use UUID to Identify a Resource

Interaction with resources

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Be compliant with the standardized HTTP method semantics summarized as follows:

GET

GET requests are used to read either a single or a collection resource.

- GET requests for collection resources may return either **200** (if the collection result is empty) or **404** (if the collection itself is missing)
- GET requests for individual resources will usually generate a **404** if the resource does not exist
- GET requests must NOT have a request body payload (see GET with body)

Situation	Response
Requested collection does not exist	404 Example: `GET /not-existing-collection` ⇒ 404

Situation	Response
Requested collection is empty	200, `[]` Example: `GET /users` ⇒ 200, `[]`
Requested collection is not empty	200, list of results Example: `GET /users` ⇒ 200, `[ { "id": 1, "username": "ed.example", ⋯ }, { "id": 2, ⋯ },⋯ ]`
Requested individual resource does not exist	404 Example: `GET /users/1` ⇒ 404
Requested individual resource exists	200, the resource Example: `GET /users/1` ⇒ 200, `{ "id": 1, "username": "ed.example", ⋯ }`

POST - Create

POST used to create single resources on a collection resource endpoint, but other semantics on single resource endpoint are equally possible.

- if resource have been **updated** POST request will generate **200**
- if resource have been **created** POST request will generate **201**
- if the request was **accepted but not has been finished yet**, POST request will generate **202**
- if the actual **resource is not returned** POST request will generate **204**

PUT - Update

PUT requests are used to **update** resources but are usually applied to single resources, and not to collection resources.

- PUT requests are usually robust against non-existence of resources by implicitly creating before updating
- if the resource have been updated- with or without content PUT request will generate **200** or **204**
- if the resource have been created PUT request will generate **201**

Usage of POST vs PUT

It is the best practice to prefer POST to PUT for creation of resources. This leaves the resource ID under control of the service and allows concentrating on the update semantic using PUT. For example to create a new product rather use POST than PUT:

Good

```
POST /shop/products HTTP 1.1
```

Bad

```
PUT /shop/products HTTP 1.1
```

PATCH

PATCH requests are used to **update parts** of single resource for example if only a particular subset of resource fields should be replaced.

- PATCH request are usually not robust against non-existing resources
- if resource have been updated with or without updated content returned PATCH request will generate **200** or **204**
- Can be used to update references between existing resources e.g. `PATCH /books/1 { "authorId" : 234 }`.

If patching multivalued fields is needed, have a look at A better way to implement HTTP PATCH operation in REST APIs and RFC 6902.

Usage of PATCH vs PUT

- use PATCH with partial objects to only update parts of a resource, whenever possible. (This is basically JSON Merge Patch, a specialized media type application/merge-patch+json that is a partial resource representation.)
- use PATCH with JSON Patch, a specialized media type application/json-patch+json that includes instructions on how to change the resource.
- use PATCH only with an **explicit reason** why POST or PUT does not fit here
- use PUT over PATCH to update a resource as long as feasible. For example to update a product rather use PUT than PATCH:

Good

```
PUT /shop/products/1 HTTP 1.1
```

Bad

```
PATCH /shop/products/1 HTTP 1.1
```

DELETE

DELETE requests are used to delete single resources but also can be applied to multiple resources using query parameters on the collection resource.

- if the deleted resource is returned with or without content, DELETE request will generate **200** or **204**
- failed DELETE requests will usually generate **404** (if the resource cannot be found) or **410** (if the resource was already deleted before).

DELETE with query Parameters

DELETE requests can have query parameters. Query parameters should be used as filter parameters on a resource.

```
DELETE /resources?param1=value1&param2=value2...&paramN=valueN
```

DELETE with body

Sometimes DELETE requests may require additional information, that cannot be classified as filter parameters.

```
POST /resources/{resource-id}
{
  "prop1": "value1",
  ...
  "propN": "valueN",
}
```

HEAD

HTTP HEAD requests are used to retrieve the header information of single resources and resource collections. For example, if a URL might produce a large download, a HEAD request could read its Content-Length header to check the filesize without actually downloading the file.

- a response to a HEAD method should not have a body. If it has one anyway, that body must be ignored.

In addition, everyone **must** be aware which methods are **safe**, **idempotent** and **cacheable**

Safe Methods

As per HTTP specification, the **GET** and **HEAD** methods should be used only for retrieval of resource representations - and they do not update/delete the resource on the server. Both methods are said to be considered "safe".

Idempotent Methods

The term idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times. In HTTP specification, The methods **GET**, **HEAD**, **PUT** and **DELETE** are declared idempotent methods.

Cacheable Methods

Request methods are considered "cacheable" if it is possible and useful to answer a current client request with a stored response from a prior request. **GET** and **HEAD** are defined to be cacheable.

Versioning

It is best to avoid versioning altogether, because versions give you the illusion you could change your API. You can't force your customers to switch to the new version. So either you need to support all API versions forever or you can only apply backwards compatible change, but then you do not need API versioning anyway. See "API-Design – Wie man es besser (nicht) macht" by Uwe Friedrichsen.

If you cannot avoid versioning use the following best practices:

To identify the version of the API a three number semantic versioning restricted to the format <MAJOR><MINOR><PATCH> for versions as follows is recommended:

- Increment **MAJOR** version when breaking changes* were made
- Increment **MINOR** version when new functionality in a backwards-compatible manner was added
- Increment **PATCH** version when backwards-compatible bug fixes or editorial changes not affecting the functionality were made

*breaking changes are backwards-incompatible changes

Even for very small changes that may affect only one percent of the clients, the MAJOR version would have to be incremented, because semantic versioning is not there to indicate how extensive a change is. Alternatively, a simple scheme like **MAJOR.MINOR** can be used. MAJOR version shows major changes and the MINOR release shows minor changes.

Compatibility

There are two techniques to change APIs without breaking them

- follow rules for compatible extension
- introduce new API versions and still support older versions

Prefer to make **compatible changes** without versioning. The following steps will help to make compatible changes:

- Add only optional never mandatory fields
- Never change the semantic of the fields
- Never change the validation logic to be more restrictive
- return **JSON** object as a top-level data structure to support future extensibility [support compatible extensions by additional attributes]

Prepare Service **Clients** accept compatible API extensions

- Be tolerant with unknown fields in the payload
- Be prepared to handle HTTP status codes not explicitly specified in endpoint definitions

Avoid versioning

When changing your REST API, do it in a compatible way and avoid generating additional API versions. If changes **can't** be done in a compatible way, then follow these three steps:

- create a new resource (variant) in addition to the old resource variant
- create a new service endpoint
- create a new API version supported in parallel with the old API by the same microservice

Deprecation

In some cases it is necessary to phase out an API endpoint, an API version, or an API feature, e.g. if a field or parameter is no longer supported or a whole business functionality behind an endpoint is supposed to be shut down.

As long as the API endpoints and features are still used by consumers these shut-downs are breaking changes and not allowed. To progress the following deprecation rules have to be applied to make sure that changes and actions are well communicated and aligned using deprecation dates.

- must obtain **approval of clients** before shut down.
- must collect **external partner consent** on deprecation time span
- must **reflect deprecation** in API specifications
- must **monitor usage** of deprecated API
- must **not start using deprecated APIs**

Data Formats

Use JSON-encoded body payload

JSON is the standard for transferring data, so REST APIs should accept JSON for request payload and also send responses to JSON. In the response header the content-type should be set to **application/json** to make sure that if we send a response encoded as JSON, it will be interpreted as JSON by the client as well.

Date and Time Format

Use ISO 8601 format for passing in and out dates and times. Use [UTC] as timezone.

Date	**2020-08-03**
date and time in UTC	2015-07-02T14:47:47Z

Language Code Format

Use language codes defined by the ISO 639 code standard.

Code	**Language**
en	English
de	German
it	Italian

Country Code Format

Use country codes defined by the ISO 3166-1-alpha-2 code standard.

Code	**Country**
AD	Andorra
DE	Germany
FR	France
US	United States of America

Currency Format

Use 3-character ISO-4217 codes for specifying currencies.

Code	**Currency**
EUR	Euro
CHF	Swiss franc
USD	US Dollar

Number and Integer Formats

Whenever an API defines a property of type number or integer, the precision must be defined by the format as follows:

Type	Format	Value Range
integer	int32	integer between -231 and 231-1
integer	int64	integer between -263 and 263-1
integer	bigint	arbitrarily large signed integer number
number	float	IEEE 754 binary32 decimal number
number	double	IEEE 754 binary64 decimal number
number	decimal	arbitrarily precise signed decimal number

Parameter

Query parameters should be optional (no error if left out)

Naming of Query Parameters

- Query parameters start with a letter.
- Query parameters should be either camelCase or snake_case (consistent with case standards for field names).
- As always, choose good unambiguous names e.g. "firstname" instead of "name".

Good

```
...?firstName=Max
...?first_name=Max
```

Bad

```
...?first-name=Max
...?name=Max
```

Filtering

URL parameters are the easiest way to add basic filtering to REST APIs. For example:

```
// exact match
GET /persons?firstname=max
GET /persons?firstname=max&surname=muster
```

```
// lower or higher than
GET /persons?age<18
GET /persons?age>18

// list of persons whose firstname contains the "der" word (Leander, Alexander ...)
GET /persons?firstname=%2Ader%2A
```

URL parameters only have a key and a value but filters are composed of three components.

- The property or field name
- The operator (eq, lte, gte)
- The filter value

One way to encode operators is the use of square brackets [] on the key name. For example:

```
// persons older than 21
GET /persons?age[gte]=21
```

Similar to the bracket approach:

```
// persons older than 21
GET /persons?age=gte:21
```

If you require search on your endpoint, you can add support for filters and ranges directly with the search parameter

```
// items that contain the terms "red chair"
GET /items?q=title:red chair
```

Selection

Field Selection (to avoid over-fetching)

Mobile clients display just a few attributes in a list. They don't need all attributes of a resource. Give the API consumer the ability to choose returned fields. This will also reduce the network traffic and speed up the usage of the API.

```
// returning firstname, surname and id of persons
GET /persons?fields=firstname,surname,id
```

A typical answer to the problem of *over-fetching* is to use GraphQL instead of REST. But the problem can often be solved using selection instead.

Inclusion

Inclusion (to avoid under-fetching)

An example request without including associated resources:

```
GET /books/1

=> 200, OK:
{
   "id": 1,
   "title": "My great novel",
   ...
   "authorId": 123
}
```

An example request that includes an associated resource:

```
GET /books/1?include=author

=> 200, OK:
{
   "id": 1,
   "title": "My great novel",
   ...
   "authorId": 123,
   "author": {
      "id": 123,
      "firstname": "Max",
      "surname": "Maximus",
      ...
   }
}
```

An example request that includes an associated resource selecting only some fields:

```
GET /books/1
        ?include=author
        &fields=title,isbn,author.surname

=> 200, OK:
{
   "title": "My great novel",
   "isbn": "978-7-4171-0270-1",
   "author": {
      "surname": "Maximus"
   }
```

```
}
```

An example request that includes multiple associated resources selecting only some fields:

```
GET /books/1
        ?include=author,publisher
        &fields=
            title,isbn,
            author.surname,
            publisher.name

=> 200, OK:
{
   "title": "My great novel",
   "isbn": "978-7-4171-0270-1",
   "author": {
      "surname": "Maximus"
   },
   "publisher": {
      "name": "Maximus Press"
   }
}
```

A typical answer to the problem of *under-fetching* is to use GraphQL instead of REST. But the problem can often be solved using inclusion instead.

Sorting

Allows ascending (+) and descending (-) sorting over multiple fields.

```
// returns a list of persons sorted by ascending age.
GET /persons?sort=+age
```

Pagination

Lists that potentially larger than just a few hundred entries must support pagination to protect the service against overload as well as for best client side iteration and batch processing experience.

There are two popular techniques to support pagination:

- Offset/Limit-based pagination: numeric offset identifies the first page entry
- Cursor/Limit-based: a unique key element identifies the first page entry

To navigate to a specific page, **next/prev page links** were recommended to enhance the user experience. This favors **cursor-based** over offset-based pagination.

Choosing the right pagination technique depends on the particular use cases:

- **Usability/framework support**
  - Offset-based pagination is more widely known, so it has more framework support and is easier to use for API Clients
- **Use Case - jump to a certain page**
  - cursor-based pagination is not feasible for jumping to a particular page in a range. (e.g 51 of 100)
- **Data changes may lead to anomalies in result pages:**
  - Offset-based pagination may create duplicates or lead to missing entries if rows are inserted or deleted between two subsequent paging requests.
  - If implemented incorrectly, cursor-based pagination may fail when the cursor entry has been deleted before fetching the pages.
- **efficient server-side processing with very big data sets is hardly feasible using offset-pagination**

The cursor used for pagination is an opaque pointer to a page, that must never be inspected or constructed by clients. It usually encodes (encrypts) the page position, i.e. the identifier of the first or last page element, the pagination direction, and the applied query filters - or a hash over these - to safely recreate the collection. The cursor may be defined as follows:

```
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
```

The page information for cursor-based pagination should consist of a cursor set, that besides next may provide support for prev, first, last, and self as follows:

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

Error-Handling

HTTP status codes

Define all **success** and service specific **error** responses in your API specification. Both are part of the interface definition and provide important information for service clients to handle standard as well as exceptional situations. But notice, it is not useful to document all technical errors, especially if they are not under control of the service provider. See overview on all HTTP status codes on Wikipedia

Success codes

Status code	Description	Methods
200	**OK** - standard success response	ALL
201	**Created** resource created	POST, PUT
202	**Accepted** for processing - but processing has not been completed	POST, PUT, PATCH, DELETE
204	**No content** - successfully processed but **must not** include a message body	PUT, PATCH, DELETE

The **204** response is usually used as a result of a **PUT** request. The status code allows a server to indicate that the action has been successfully applied to the target resource, but that the client doesn't need to go away from its current page.

For example, a **204** status code is commonly used with document editing interfaces corresponding to a "save" action, such that the document being saved remains available to the user for editing.

Redirection codes

Status code	Description	Methods
301	**Moved Permanently** - This and all future requests should be directed to the given URI.	ALL
303	**See Other** - The response of the request can be found under another URI	POST, PUT, PATCH, DELETE
304	**Not Modified** - resource has not been modified since the version specified	GET, HEAD

Client side error codes

Status code	Description	Methods
400	**Bad Request** - Server cannot/will not process the request due to an apparent client error	ALL
401	**Unauthorized** - Similar to 403, but specially for use when authentication is required and has failed	ALL
403	**Forbidden** - Request contained valid data and was understood by the server, but the server is refusing action	ALL
404	**Not Found** - Requested resource could not be found but may be available in the future	ALL
405	**Method not Allowed** - Requested method is not supported for the requested resource	ALL

Status code	Description	Methods
406	**Not Acceptable** -The requested resource is capable of generating only content not acceptable according to the Accept headers sent in the request	ALL
408	**Request Timeout** - Server timed out waiting for the request	ALL
409	**Conflict** (business logic) - Request could not be processed e.g. create request but resource already exists	POST, PUT, PATCH, DELETE
410	**Gone** - Requested resource is no longer available	ALL
415	**Unsupported Media Type** - request entity has a media type which the server does not support	POST, PUT, PATCH, DELETE
423	**Locked** - The resource that is being accessed is locked	PUT, PATCH, DELETE
428	**Precondition Required** - The origin server requires the request to be conditional	ALL
429	**Too Many Requests** - The user has sent too many requests in a given amount of time	ALL

Server side error codes

Status code	Description	Methods
500	**Internal Server Error** - generic error message	ALL
501	**Not Implemented** - Server either doesn't recognize the request method, or it lacks the ability to fulfill the request	ALL
503	**Service unavailable** - server cannot handle the request	ALL

Error details

Only returning status codes is usually not enough information to handle an error. High quality error codes tell you what went wrong and why it went wrong.

Follow RFC 7807 as closely as possible. Event though is still only a "Proposed Standard" we recommend to follow it as closely as possible.

Consider that some details may be interesting for attackers: Decide carefully what information to include in the error response. The details provided ought to focus on helping the client correct the problem, rather than giving debugging information. Never include technical details like stack-traces or other sensitive information. Please, see the following tip how to deal with these.

Example:
Authentication and authorization: "Username does not exist" may be too much detail. This way an attacker can find out which users exist and which users don't.

Reference to sensitive or technical detail using a log reference: A good way to omit sensitive details without loosing them is to log them in the backend accompanied by a log reference (or error detail reference) and to include the log reference in the response. This helps support employees and developers analyzing and fixing a problem.

- Log references can be implemented using a UUID.
- Include the log reference in the detail message or in a separate field of the error response.
- Always include a log reference for unknown errors.

Three basic information should be included in every error message.

- **HTTP status code** - identify source and range (use standardized status codes)
- **Internal reference ID** - document specific notation of errors
- **Readable error message** - summarize context, cause and general solutions for the particular error

Let's take the **400 - Bad request** example: completely meaningless for the client-side resolution of the problem. The following example is much more precise:

```
< HTTP/1.1 400 Bad Request
< Date: Wed, 31 May 2020 19:01:41 GMT
< Server: Apache/2.4.25 (Ubuntu)
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: application/json
```

```
{
    "statusCode": 400,
    "errorMessage": "Bad Request - Your request is missing parameters. Please verify
and resubmit.",
    "errorId": "CLIENT_MISSING_PARAMETERS",
    "requestURL": https://ft-jpa.ds.net/dabdd/status/v1/register,
    "requestTimestamp": 2020-05-31T19:01.40z"
    "parameterErrors" [
       {
           "name": "user.name",
           "message": "must not contain null value"
       },
       {
           "name": "vehicles",
           "message": "must not be empty"
       }
     ]
}
```

The following example shows an error-message caused by a correct client side request. The parameter was transmitted correctly by the client but cannot be processed on the server side.

```
< HTTP/1.1 422 Unprocessable Entity
< Date: Wed, 31 May 2020 19:01:41 GMT
< Server: Apache/2.4.25 (Ubuntu)
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: application/json

{
    "statusCode": 422,
    "errorMessage": "E-Mail address not known.",
    "errorId": "LOGIN_EMAIL_UNKNOWN",
    "requestURL": "https://ft-jpa.ds.net/dabdd/status/v1/login",
    "requestTimestamp": 2020-05-31T19:01.40z"
 }
```

It’s not helpful to use the same structure for every error-cases. Some error-codes need more, some need less information. With regard to the API design it’s important to identify possible errors at the API and link them to the particular error codes. Furthermore, consider the error responses. Which parameters are returned in the response for which error case? The goal is to have one error code with defined response parameters for each potential error case. To identify the particular response parameter, a few questions should be clarified in advance:

- errors caused by client or internal causes?
- which response parameters are helpful for the customer?
- which response parameters are helpful for the API-provider, to identify the cause of error as fast as possible?

Security

Most APIs are exposed to the Internet, so they need suitable security mechanisms to prevent abuse, protect sensitive data, and ensure that only authenticated and authorized users can access them. (See OWASP TopTen for more detailed information.)

Authentication and Authorization

Connection Security

Secure REST APIs should only provide HTTPS endpoints to ensure that all API communication is encrypted using SSL/TLS. This allows clients to authenticate the service and protects the API credentials and transmitted data.

API Access Control

Because REST APIs are stateless, access control is handled by local endpoints. The most common REST API authentication methods are (Authentication and Authorization),(security-cheatsheet):

JSON Web Tokens (JWT)

Credentials and other access parameters are sent as JSON data structures. These access tokens can be signed cryptographically and are the preferred way of controlling access to REST APIs.

OAuth

Standard OAuth 2.0 mechanisms can be used for authentication and authorization.

API Keys

API Keys can be used to prevent abuse or malicious use of the API. Furthermore, they can reduce the impact of denial-of-service attacks. However, when API Keys are issued to third-party clients, they are relatively easy to compromise.

- Require API keys for every request to the protected endpoint.
- Return **429** - Too Many Requests if requests are coming in too quickly.
- Revoke the API key if the client violates the usage agreement.
- Do not rely exclusively on API keys to protect sensitive, critical or high-value resources.

Restrict HTTP methods

- Apply a whitelist of permitted HTTP Methods e.g. **GET**, **POST**, **PUT**.
- Return **405** - Method not allowed if request not matching the whitelist.
- Make sure the caller is authorized to use the incoming HTTP method on the resource collection, action, and record

Validation of parameters

Typical best-practice guidelines for input validation apply:

- Treat all parameters, objects, and other input data as untrusted.
- Use built-in validation functionality where available.
- Check the request size and content length and type.
- Use strong typing for API parameters (if supported).
- To prevent SQL injection, avoid building queries manually – use parameterized queries instead.
- Whitelist parameter values and string inputs wherever possible.
- Log all input validation failures to detect credential stuffing attempts.

Management Endpoints

- Avoid exposing management endpoints via Internet.
- If management endpoints must be accessible via the Internet, make sure that users must use a strong authentication mechanism, e.g. multi-factor.
- Expose management endpoints via different HTTP ports
- Restrict access to these endpoints by firewall rules or use of access control lists.

Sensitive Information in HTTP requests

In order not to leak credentials. Passwords, security tokens, and API keys should not appear in the URL.

- In **POST**, **PUT** requests sensitive data should be transferred in the request body or request headers.
- In **GET** requests sensitive data should be transferred in an HTTP header.

GOOD

```
https://example.com/collection/{123}/action
```

BAD

```
https://example.com/collection/{123}/action?apiKey=a123
```

Documentation

API designers are required to provide a short but meaningful description about the purpose of the API. It is recommended to publish the API documentation with the deployment of the implementing service. In the following, **best practices** for API documentation are listed:

- describe the purpose of the API
- describe important use cases to give developers a quick head start
- enable a developer to use your API as quick as possible
- Use OpenAPI (aka Swagger) as **description and definition** language of your API.
- Use **OpenAPI** to create API **reference documentation**.
- Use Software Creation Chain interface documentation as help for documenting your API.

Performance

Caching

The goal of caching is never having to generate the same response twice. Furthermore, by doing this, we **gain speed** and **reduce server load**. The best way to cache your API is to put a gateway cache (or reverse proxy) in front of it. A very powerful open-source reverse proxy is Varnish.

When a **safe method** is used on a resource URL, the reverse proxy should cache the response that is returned from your API, and then will use this cached response for all subsequent requests for the same resource before they hit your API.

When an **unsafe method** is used on a resource URL, the cache ignores it and passes it to the API.

Cache-Control

Generally we assume that GET requests are cached and PUT, POST and DELETE requests are not. Unfortunately most APIs require additional caching rules for some requests. For these reasons, it's a good idea to set cache-control headers. These contain many different options for the appropriate handling of cached data. Refer to Overview of Cache-Control HTTP headers for additional controls.

For example use **max-age** to indicate after how many seconds the response should be considered out-of-date:

```
GET users/123

HTTP 1.1 200 OK
...
Cache-Control: max-age=600
Content-Type: text/json; charset=utf-8
Content-Length: ...
{
    "id": 123,
    "firstname": "Max",
    "surname":"Mustermann"
}
```

To disable caching completely use the **no-cache** and **no-store** directives:

```
GET users/123

HTTP 1.1 200 OK
...
Cache-Control: no-cache, no-store
Content-Type: text/json; charset=utf-8
Content-Length: ...
{
    "id": 123,
    "firstname": "Max",
    "surname":"Mustermann"
}
```

Compression

Compress the payload of your APIs responses using gzip, this helps to transport data faster over the network (fewer bytes) and makes frontends respond faster. But do **not** use gzip compression if you are serving so many requests that the time to compress becomes a bottleneck.

Next to requesting a particular resource, the client sends **Accept-Encoding header** that says what kind of compression algorithms the client understands. For example:

```
GET                 /employees          HTTP/1.1
Host:               www.domain.com
Accept:             text/html
Accept-Encoding:    gzip, compress
```

If the server understands one of the compression algorithms from Accept-Encoding, it can use that algorithm to compress the representation before serving it. When successfully compressed, server lets know the client of encoding scheme by the **Content-Encoding** header. For example:

```
200 OK
Content-Type:       text/html
Content-Encoding:   gzip
```

Asynchronicity

If an API operation is asynchronous you **must** return a **202-ACCEPTED** response code. This informs the client that the request has been accepted and understood by the server, but the resource is not yet completed. Instead of the URI of the actual resource, it would send a location to a status resource. For example a response like:

```
HTTP/1.1 202 Accepted
Location: /queue/621252
```

The location URI points to a (created) resource that will display the status of the asynchronous processing:

Request:

```
GET /queue/621252 HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK
{
   "status": "In progress",
   "eta": "3 minutes, 25 seconds"
}
```

As soon as processing is done, the server can create the original resource and delete the queue-resource. If the client wants to fetch the status again, the server will return a **303-SEE OTHER** code:

Request:

```
GET /queue/621252 HTTP/1.1
```

Response:

```
HTTP/1.1 303 See Other
Location: /blog/20010101-myblogarticle
```

Further-Concepts

Testability

Every API description (contract) using HTTP(S) protocol **MUST** be tested against its API implementation. The tests can be executed using the Dredd testing framework.

In addition to local runs, the tests should be an integral part the API implementation's CI/CD pipeline. The CI/CD pipeline should be configured to run the test whenever there is a change to either API description (contract) or its implementation.

Consumer Driven Contracts

In a micro service architecture many services are created in different programming languages. To ensure smooth communication between them, the interfaces must fit and remain stable over time. Consumer-Driven Contracts is an approach that additionally tests the interfaces and their callers.

Custom Headers

Naming

Custom headers have been used in the past with an X-prefix, but this convention was deprecated in June 2012 because of the inconveniences it caused when nonstandard fields became standard in RFC 6648. IANA also maintains a registry of proposed new HTTP headers.

Usage

Custom headers are often used in system to system connections where the system is acting on behalf of a third party (See also proprietary-headers).

For example, user U talks to server A. Server A presents credentials to server B with a custom header to say "Use my credentials to check that I'm authorized to perform this action on behalf of user U.

HATEOAS

What is HATEOAS?

HATEOAS stands for **H**yptertext **A**s **T**he **E**ngine **O**f **A**pplication **S**tate.It means that hypertext should be used to find your way through the API. An example:

```
GET /account/123 HTTP/1.1

HTTP 1.1 200 OK

{
    "account_number":123,
    "balance":100,
    "links": [
        {
            "href":"/account/123/deposit",
            "rel":"deposit"
        },
        {
            "href":"/account/123/withdraw",
            "rel":"withdraw"
        },
        {
            "href":"/account/123/transfer",
            "rel":"transfer"
        },
        {
            "href":"/account/123/close",
            "rel":"close"
        }
```

```
    ]
}
```

Apart from fact that we have 100 Dollars in our account we are provided with 4 options: deposit more money, withdraw money, transfer money to another account, or close our account. The links allow us to find out the URLs that are needed to the specific actions. Ow, let's suppose we didn't have 100 usd in the bank, but we actually are in the red:

```
GET /account/123 HTTP 1.1

HTTP 1.1 200 OK

{
    "account_number":123,
    "balance":-50,
    "links": [
        {
            "href":"/account/123/deposit",
            "rel":"deposit"
        }
    ]
}
```

Right now we have lost many of our options, and and only depositing money is valid? As long as we are in the red, we cannot close our account, nor transfer or withdraw any money from the account. The hypertext is actually telling us what is allowed and what not.

Relevant in Practice?

Similar to an interaction with a website, a REST client hits an initial API URI and uses the server-provided links to dynamically discover available actions and accesses the resources it needs. The client doesn't need prior knowledge of the service or the different steps involved in a workflow. Additionally, the clients no longer have to hard code the URI structures for different resources. HATEOAS allows the server to make URI changes as the API evolves without breaking the clients.

In practice HATEOAS is difficult to implement. This makes the API very complex and can lead to many requests when clients navigate through the links instead of reading the documentation and jumping directly to the required endpoints. On top of that, HATEOAS requires that clients use the API correctly, but in most cases a developer will read the documentation and access the required endpoints directly instead of navigating through links. Furthermore, the mere presence of the links is not sufficient to dissociate customers from the need to learn data required to create requests. An API **MUST** provide documentation to clearly describe all the links, link relation types and request response formats for each of the URIs.

However, you can use it, if you checked its limitations and still see clear value for your usage that justifies its additional complexity.

GraphQL

GraphQL is an **alternative** to the REST-API allowing clients to query the data in the structure they need, rather than relying on a predefined data structure of a fixed API. This helps to avoid over- and underfetching but brings its own implementation challenges.

GraphQL can not be used instead of a full REST service (HATEOAS). But can be a good alternative to Level 2 REST services of the Richardson Maturity Model.

In conclusion GraphQL is an interesting approach, but does not offer a solution for all problems. It depends on the specific use case, which tool is more suitable.

OData

OData (Open Data Protocol) is a HTTP based protocol which enables creation of REST-based services which allow resources identified using Uniform Resource Locators (URLs) and defined in a data model, to be published and edited by Web clients using simple HTTP messages.

OData is an **alternative** interface technology which **fully supports the REST principles**. It helps applications to focus on business logic without worrying about the various API approaches to define request and response headers, status codes, HTTP methods, URL conventions, media types, payload formats, query options, etc (For further reading: OData).

REST - more than just CRUD

Often the architectural style of REST APIs is limited to CRUD operations only. Resources are often mapped exactly to the internal domain-/entity-types using the HTTP verbs (POST, GET, PUT, DELETE). But sometimes you have to implement some kind of business logic that is not directly related to a CRUD operation.

For example, you might want to block or suspend a user, which is different from deleting them. It is often possible to map these to update actions, for example, updating a user with a status attribute set to suspended but in some cases, this could feel clunky and forced.

In those cases, it makes sense to include **actions as resources** and place them at URLs such as:

```
/users/{ID}/actions/suspend
```

It’s a great idea to denote them clearly, e.g., by prefixing them with /actions, although the use of a verb (like “suspend”) already gives it away that we’re breaking away from pure CRUD for better developer experience.

A detailed documentation is very important to ensure that the API is used correctly.

Checking for Existence

A standard way to check for existence is to use a HEAD request. The result of the HEAD request is either 200 OK if the resource exists or 404 NOT_FOUND if it does not. However, the usage of 4xx status codes implies an error situation. This might result in unintended triggering of error interceptors on the client side. And generally it is a bad practice to abuse exception handling for control flow.

So, what if non-existence is a valid result and not an error? For example checking the existence of the username before trying to create a user?

Bad

```
GET /user/exampleuser?exists
```

This is bad, because...
... the type of response and status code is inconsistent with the ordinary GET request
... the username is used as a path parameter (see section Identification of Resources)

```
GET /persons/exampleuser/exists
```

This is bad, because it introduces a new concept not compatible with REST principles (verb in the path).

Use filtering and selection for existence checks instead:

Good

```
GET /persons?username=exampleuser&fields=username

=>
// If the resource person with the given username exists:
200 OK, [{"username": "exampleuser"}]

// If person with the given username does not exist:
200 OK, []
```

For the check it is advisable to also use a specific query in the backend that only checks existence instead of fetching resource details.

Examples of good REST-APIs

- Stripe API Reference
- Twilio Docs

- GitHub API Documentation
- Twitter API Documentation

References

The following links and books are recommended for those who want to go deeper into the subject of good API design.

- Richardson Maturity Model - service-oriented-computing.de - good explaination of the Richardson Maturity Model *(article and video)*
- REST API Design Guideline - digitalchargingsolutions.com *(guideline)*
- REST API Tutorial by Lokesh Gupta (howtodoinjava.com)
- BMW Connected Vehicle API Guides *(tutorial)*
- Zalando RESTful API and Event Guidelines - zalando.com *(guideline)*
- Azure / Architecture / Best Practices: RESTful web API design - microsoft.com *(guideline)*
- Microsoft REST API Guidelines on github.com *(guideline)*
- The Web API Checklist - 43 Things To Think About When Designing, Testing, and Releasing your API by Mathieu Fenniak *(article)*
- Best practices in API documentation - Swagger.io
- API Stylebook - Design Guidelines - a collection of API design guidelines of various companies and government agencies *(collection of guidelines)*
- How to do stuff RESTful - Restcookbook.com *(cookbook)*
- API-Design – Wie man es besser (nicht) macht by Uwe Friedrichsen *(video)*
- The Design of Everyday APIs by Arnaud Lauret *(video)*
- List of HTTP status codes - Wikipedia *(definition)*
- RFC 7807: Problem Details for HTTP APIs *(RFC)*
- HATEOAS - Wikipedia *(definition)*
- A better way to implement HTTP PATCH operation in REST APIs on medium.com *(article)*
- RFC 6902: JavaScript Object Notation (JSON) Patch *(RFC)*
- Naming Conventions - api.gov.au *(naming convention)*
- REST API Mistakes Every Junior Developer should Avoid by Islem Maboud (aka CoderOne) *(video)*