# Pon Integration and Development Guidelines

Document source files: Github

Other formats: PDF

# Table of Contents


Pon Integration and Development Guidelines . . . 1
1. Introduction . . . 10
1.1. Software engineering . . . 10
1.1.1. Software requirements . . . 11
1.1.2. Software architecture and design . . . 11
1.1.3. Software development . . . 11
1.1.4. Software testing . . . 11
1.1.5. Software maintenance and support . . . 11
1.2. Document management . . . 12
1.3. Attribution . . . 12
1.4. Conventions used in these guidelines . . . 12
1.5. Pon specific information . . . 12
1.5.1. Automated code style checking (linting) . . . 13
1.5.2. Guideline or standard . . . 13
1.5.3. **MUST** comply with standards and guidelines . . . 13
1.5.4. **(RFP) MUST** write all resources using U.S. English . . . 13
2. Generic . . . 13
2.1. Guilds . . . 13
2.1.1. Communication . . . 13
2.1.1.1. Ad-hoc . . . 13
2.1.1.2. New rules updates . . . 13
2.2. Monitoring . . . 13

2.2.1. **(RFP) MUST** have predefined monitoring based KPIs . . . 13
2.2.2. **(RFP) MUST** be able monitor operational state using automated tools . . . 14
2.2.3. **(RFP) SHOULD** be able to report detailed relevant operational states . . . 14
2.3. Quality . . . 14
2.3.1. **(RFP) SHOULD** have seperate environments for development, testing, acceptance and production . . . 14
2.3.1.1. Development . . . 14
2.3.1.2. Testing . . . 14
2.3.1.3. Acceptance . . . 14
2.3.1.4. Production . . . 15
2.3.2. Code Quality . . . 15
2.3.2.1. **(RFP) MUST** have code duplication checks . . . 15
2.3.2.2. **(RFP) MUST** have vulnerability checks . . . 15
2.3.3. GUI and websites quality . . . 15
2.3.3.1. **(RFP) SHOULD** have predefined tests . . . 15
2.3.4. API quality . . . 15
2.3.4.1. **(RFP) MUST** have predefined tests . . . 15
2.3.4.2. **(RFP) MUST** have a pre-defined code-coverage percentage . . . 15
2.4. Security . . . 15
2.4.1. **(RFP) MUST** be compliant with the Pon Security Policy and Principles . . . 16
2.4.2. **(RFP) MUST** be compliant with Binding Corporate Rules and local privacy legislation . . . 16
2.4.3. **(RFP) MUST** have performed a Security & Privacy intake . . . 16
2.4.4. **(RFP) MUST** have the agreed security and privacy measures approved . . . 16
2.5. Privacy . . . 16
2.6. Documentation . . . 16
2.6.1. **MUST** include documentation comment saying what the tool is for . . . 16
2.6.2. **MUST** include monitoring documentation . . . 16
2.6.2.1. References . . . 17
2.6.3. **MUST** document deployment procedures . . . 17
2.7. Onboarding . . . 17
2.7.1. **(RFP) MUST** include guidelines in onboarding procedure . . . 17
2.7.2. **(RFP) SHOULD** have pre-defined development environment(s) . . . 17
2.7.2.1. Patron(s) □ . . . 17
2.7.3. Solution architecture repository . . . 18
2.8. Intellectual property (IP) . . . 18
2.8.1. **(RFP) MUST** include project wide license file . . . 18
2.8.2. **(RFP) MUST** include copyright notice in each source file . . . 18
2.9. Checklist . . . 18
3. Integration guidelines . . . 18
3.1. Principles . . . 18
3.1.1. API design principles . . . 18

3.1.2. API as a product . . . 19
3.1.3. API first . . . 20
3.2. Solution design . . . 21
3.2.1. Microservices . . . 21
3.2.2. Connectivity . . . 21
3.2.2.1. **(RFP) MUST** have self-healing connectivity . . . 21
3.2.2.2. **(RFP) SHOULD** have increasing reconnection intervals . . . 21
3.2.3. Loosely coupled . . . 21
3.2.4. Security . . . 21
3.2.5. Monitoring . . . 21
3.2.5.1. **(RFP) MUST** setup monitoring and alerting connections . . . 21
3.2.6. Documentation . . . 21
3.2.7. Security . . . 21
3.3. Generic . . . 21
3.3.1. Pagination . . . 21
3.3.1.1. **MUST** support pagination . . . 21
3.3.1.2. **SHOULD** prefer cursor-based pagination, avoid offset-based pagination . . . 22
3.3.1.3. **SHOULD** use pagination links where applicable . . . 24
3.4. Types . . . 24
3.4.1. FTP . . . 24
3.4.1.1. Monitoring . . . 24
3.4.1.2. Documentation . . . 24
3.4.1.3. Security . . . 24
3.5. General guidelines . . . 25
3.5.1. **(RFP) MUST** follow API first principle . . . 25
3.5.2. **(RFP) MUST** provide API specification . . . 25
3.5.3. **(RFP) MUST** only use durable and immutable remote references . . . 25
3.5.4. **(RFP) MAY** provide API user manual . . . 25
3.6. Meta information . . . 25
3.6.1. **(RFP) SHOULD** contain API meta information . . . 26
3.6.2. **(RFP) MAY** use semantic versioning . . . 26
3.6.3. **(RFP) MAY** provide API identifiers . . . 26
3.6.4. **(RFP) SHOULD** provide API audience . . . 27
3.7. Security . . . 29
3.7.1. **(RFP) MUST** secure endpoints . . . 29
3.7.1.1. References . . . 29
3.7.2. **(RFP) SHOULD** define and assign permissions (scopes) . . . 29
3.7.3. **(RFP) MAY** follow naming convention for permissions (scopes) . . . 30
3.8. Compatibility . . . 31
3.8.1. **(RFP) MUST** not break backward compatibility . . . 31
3.8.2. **(RFP) SHOULD** prefer compatible extensions . . . 31

3.8.3. **(RFP) MUST** prepare clients accept compatible API extensions . . . 32
3.8.4. **(RFP) SHOULD** design APIs conservatively . . . 32
3.8.5. **(RFP) MUST** always return JSON objects as top-level data structures if JSON is being used . . . 33
3.8.6. **(RFP) SHOULD** refrain from using enumerations . . . 33
3.8.7. **(RFP) SHOULD** avoid versioning . . . 34
3.8.8. {STATUS-TODO} **MUST** API Versioning Has No “Right Way” . . . 34
3.8.9. **(RFP) SHOULD** use URI versioning . . . 34
3.9. Deprecation . . . 35
3.9.1. **(RFP) MUST** obtain approval of clients before API shut down . . . 35
3.9.2. **(RFP) MUST** collect external partner consent on deprecation time span . . . 35
3.9.3. **(RFP) MUST** reflect deprecation in API specifications . . . 35
3.9.4. **(RFP) MUST** monitor usage of deprecated API scheduled for sunset . . . 35
3.9.5. **(RFP) SHOULD** add `Deprecation` and `Sunset` header to responses . . . 35
3.9.6. **(RFP) SHOULD** add monitoring for `Deprecation` and `Sunset` header . . . 36
3.9.7. **(RFP) MUST** not start using deprecated APIs . . . 36
3.10. Common data types . . . 36
3.10.1. **(RFP) MUST** use the common money object . . . 36
3.10.1.1. Cons . . . 37
3.10.1.2. Pros . . . 38
3.10.1.3. Notes . . . 38
3.10.2. **(RFP) MUST** use common field names and semantics . . . 38
3.10.2.1. Generic fields . . . 38
3.10.2.2. Link relation fields . . . 40
3.10.2.3. Address fields . . . 41
3.11. API naming . . . 43
3.11.1. **MUST/SHOULD** use functional naming schema . . . 43
3.11.2. **MUST** follow naming convention for hostnames . . . 44
3.11.3. **MUST** use lowercase separate words with hyphens for path segments . . . 44
3.11.4. **MUST** use snake_case (never camelCase) for query parameters . . . 44
3.11.5. **SHOULD** prefer hyphenated-pascal-case for HTTP header fields . . . 44
3.11.6. **MUST** pluralize resource names . . . 45
3.11.7. **SHOULD** not use /api as base path . . . 45
3.11.8. **MUST** avoid trailing slashes . . . 45
3.11.9. **MUST** stick to conventional query parameters . . . 45
3.12. Resources . . . 46
3.12.1. **MUST** avoid actions — think about resources . . . 46
3.12.2. **SHOULD** model complete business processes . . . 46
3.12.3. **SHOULD** define *useful* resources . . . 46
3.12.4. **MUST** keep URLs verb-free . . . 46
3.12.5. **MUST** use domain-specific resource names . . . 46

3.12.6. **MUST** use URL-friendly resource identifiers . . . 47
3.12.7. **MUST** identify resources and sub-resources via path segments . . . 47
3.12.8. **MAY** expose compound keys as resource identifiers . . . 47
3.12.9. **MAY** consider using (non-)nested URLs . . . 48
3.12.10. **SHOULD** only use UUIDs if necessary . . . 49
3.12.11. **SHOULD** limit number of resource types . . . 49
3.12.12. **SHOULD** limit number of sub-resource levels . . . 50
3.13. Performance . . . 50
3.13.1. **SHOULD** reduce bandwidth needs and improve responsiveness . . . 50
3.13.2. **SHOULD** use `gzip` compression . . . 51
3.13.3. **SHOULD** support partial responses via filtering . . . 51
3.13.3.1. Unfiltered . . . 51
3.13.3.2. Filtered . . . 52
3.13.4. **SHOULD** allow optional embedding of sub-resources . . . 53
3.13.5. **MUST** document cachable `GET`, `HEAD`, and `POST` endpoints . . . 53
3.14. Hypermedia . . . 56
3.14.1. **MUST** use REST maturity level 2 . . . 56
3.14.2. **MAY** use REST maturity level 3 - HATEOAS . . . 56
3.14.3. **MUST** use full, absolute URI . . . 57
3.14.4. **MUST** use common hypertext controls . . . 57
3.14.5. **SHOULD** use simple hypertext controls for pagination and self-references . . . 58
3.14.6. **MUST** not use link headers with JSON entities . . . 58
3.15. Common headers . . . 58
3.15.1. **MUST** use `Content-*` headers correctly . . . 58
3.15.2. **MAY** use standardized headers . . . 59
3.15.3. **MAY** use `Content-Location` header . . . 59
3.15.4. **SHOULD** use `Location` header instead of `Content-Location` header . . . 59
3.15.5. **MAY** consider to support `Prefer` header to handle processing preferences . . . 60
3.15.6. **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header . . . 61
3.15.7. **MAY** consider to support `Idempotency-Key` header . . . 62
3.16. Proprietary headers . . . 63
3.16.1. **MUST** use only the specified proprietary Pon headers . . . 63
3.16.2. **MUST** propagate proprietary headers . . . 65
3.16.3. **MUST** support `X-Flow-ID` . . . 65
3.16.3.1. Data Definition . . . 65
3.16.3.2. Service Guidance . . . 66
3.17. API Operation . . . 66
3.17.1. **MUST** publish Open API specification . . . 66
3.17.2. **SHOULD** monitor API usage . . . 67
3.18. Events . . . 67
3.18.1. Events, event types, and categories . . . 67

3.18.2. **MUST** treat events as part of the service interface . . . 67
3.18.3. **MUST** make event schema available for review . . . 68
3.18.4. **MUST** ensure event schema conforms to Open API schema object . . . 68
3.18.5. **MUST** ensure that events are registered as event types . . . 69
3.18.6. **MUST** ensure that events conform to a well-known event category . . . 73
3.18.6.1. The general event category . . . 73
3.18.6.2. The data change event category . . . 74
3.18.6.3. Event metadata . . . 75
3.18.7. **MUST** ensure that events define useful business resources . . . 76
3.18.8. **MUST** ensure that events do not provide sensitive data . . . 77
3.18.9. **MUST** use the general event category to signal steps and arrival points in business processes . . . 77
3.18.10. **MUST** use data change events to signal mutations . . . 77
3.18.11. **SHOULD** provide means for explicit event ordering . . . 78
3.18.12. **SHOULD** use the hash partition strategy for data change events . . . 78
3.18.13. **SHOULD** ensure that data change events match the APIs resources . . . 79
3.18.14. **MUST** indicate ownership of event types . . . 79
3.18.15. **MUST** define event payloads compliant with overall API guidelines . . . 79
3.18.16. **MUST** maintain backwards compatibility for events . . . 80
3.18.17. **SHOULD** avoid `additionalProperties` in event type definitions . . . 80
3.18.18. **MUST** use unique event identifiers . . . 81
3.18.19. **SHOULD** design for idempotent out-of-order processing . . . 81
3.18.20. **MUST** follow naming convention for event type names . . . 82
3.18.21. **MUST** prepare event consumers for duplicate events . . . 82
Appendix A: Tooling . . . 83
3.A.1. API first integrations . . . 83
3.A.2. Support libraries . . . 83
Appendix B: Best practices . . . 83
3.B.1. Optimistic locking in RESTful APIs . . . 83
3.B.1.1. Introduction . . . 83
3.B.1.2. `ETag` with `If-Match` header . . . 83
3.B.1.3. `ETags` in result entities . . . 84
3.B.1.4. Version numbers . . . 85
3.B.1.5. `Last-Modified` / `If-Unmodified-Since` . . . 86
3.B.1.6. Conclusion . . . 87
4. Development guidelines . . . 87
4.1. General development guidelines . . . 87
4.1.1. Introduction . . . 87
4.1.2. Rules and definitions . . . 88
4.1.3. Definition: code quality . . . 88
4.1.4. Coding rule: logical structured code . . . 88

4.1.5. Coding rule: code is simple and concise . . . 89
4.1.6. Coding rule: do not repeat yourself (DRY) . . . 89
4.1.7. Coding rule: code and code changes are self-explanatory . . . 90
4.1.8. Coding rule: solution design steps are template-based . . . 91
4.1.9. Coding rule: code quality is known . . . 91
4.1.10. Coding rule: cyclomatic complexity is low . . . 91
4.2. File structure and naming . . . 92
4.2.1. **(RFP) MUST** add comment to file . . . 92
4.2.2. **(RFP) MUST** filenames are either CamelCase or snake_case . . . 92
4.3. Version control . . . 92
4.3.1. **MUST** use review guidelines for version control . . . 92
4.3.1.1. Review guidelines . . . 92
4.4. Testing code . . . 92
4.4.1. **MUST** use automated linter based on approved style template . . . 92
4.4.2. **MUST** use automated tests based on approved testing template . . . 92
4.5. Monitoring & logging . . . 93
4.5.1. **SHOULD** use dedicated logging library and logging levels . . . 93
4.6. Development environment . . . 93
4.7. Development background . . . 93
4.8. Date and time handling . . . 93
4.8.1. **(RFP) MUST** use RFC 3339 for time and date encoding . . . 93
4.8.2. **(RFP) MUST** date time manipulation must be handled by a library . . . 94
4.8.3. **SHOULD** define time durations and intervals properties conform to RFC 3339 . . . 94
Appendix C: Pon Standard Style . . . 94
4.C.1. **MUST** encapsulate body of if or else . . . 94
4.C.1.1. Example 1 . . . 94
4.C.2. **SHOULD** order if statements by increased complexity . . . 95
4.C.2.1. Example 1 . . . 95
4.C.3. **MUST** use special quotes only to reduce complexity . . . 95
4.C.3.1. Example 1 . . . 95
4.C.3.2. Example 2 . . . 96
4.C.3.3. References . . . 96
4.C.4. **SHOULD** never use tabs for indentation . . . 96
4.C.4.1. References . . . 96
4.C.5. **MUST** use predefined spacing for indentation . . . 96
4.C.5.1. References . . . 96
4.C.6. **SHOULD** check return types of non-void functions . . . 97
4.C.7. References . . . 97
4.C.8. **SHOULD** check the validity of parameters inside each function . . . 97
4.C.8.1. Example 1 . . . 97
4.C.8.2. References . . . 97

4.C.9. **MUST** not have unused variables . . . 97
4.C.9.1. References . . . 98
4.C.10. **SHOULD** use < or > instead of <= or >= . . . 98
4.C.11. **SHOULD** use != instead of > or < when only a single value results in false . . . 98
4.C.11.1. Example 1 . . . 98
Appendix D: Pon Standard Style - Go . . . 98
4.D.1. **MUST** for linting we use golangci-lint in our CI/CD system . . . 98
4.D.1.1. Example linter implementation in Git Actions . . . 99
4.D.2. **SHOULD** go Vet is used to check go code for correctness in the development process . . . 99
4.D.3. **MUST** go Vet is used to check go code for correctness in the build pipeline . . . 99
4.D.4. **MUST** use tabs for indentation in Go . . . 99
4.D.5. **MUST** use gofmt in the IDE for automatic formatting . . . 99
4.D.6. {SHALL} every function is commented . . . 99
4.D.7. **MUST** single line multiple declarations are not used . . . 99
4.D.7.1. Example 1 Invalid declaration . . . 100
4.D.7.2. Example 2 Valid declaration . . . 100
4.D.8. **MUST** we do not try and catch exceptions. Errors are values and we handle errors . . . 100
4.D.9. **MUST** errors are handle only once. . . . 100
4.D.9.1. Don't do . . . 100
4.D.9.2. Better . . . 100
4.D.9.3. We can also include the stacktrace in the logging . . . 101
4.D.10. **SHOULD** add context to errors when they are meaningless in the context of the (final) receiver. . . . 101
4.D.10.1. Passing through context of the error with fmt.Errorf() . . . 101
4.D.10.2. Better → Passing through context of the error with errors.Wrap() from the "github.com/pkg/errors" package . . . 102
Appendix E: Pon Standard Style - Magento . . . 103
Appendix F: Pon Standard Style - WordPress . . . 103
5. Networking . . . 103
5.1. HTTP requests . . . 103
5.1.1. **MUST** use HTTP methods correctly . . . 103
5.1.1.1. GET . . . 103
5.1.1.2. GET with body . . . 103
5.1.1.3. PUT . . . 104
5.1.1.4. POST . . . 104
5.1.1.5. PATCH . . . 105
5.1.1.6. DELETE . . . 106
5.1.1.7. HEAD . . . 106
5.1.1.8. OPTIONS . . . 106
5.1.2. **MUST** fulfill common method properties . . . 107
5.1.3. **SHOULD** consider to design `POST` and `PATCH` idempotent . . . 107

5.1.4. **SHOULD** use secondary key for idempotent `POST` design . . . 108
5.1.5. **MUST** define collection format of header and query parameters . . . 109
5.1.6. **SHOULD** design simple query languages using query parameters . . . 109
5.1.7. **SHOULD** design complex query languages using JSON . . . 110
5.1.7.1. Example . . . 111
5.1.8. **MUST** document implicit filtering . . . 111
5.2. HTTP status codes and errors . . . 112
5.2.1. **MUST** specify success and error responses . . . 112
5.2.2. **MUST** use standard HTTP status codes . . . 113
5.2.2.1. Success codes . . . 113
5.2.2.2. Redirection codes . . . 114
5.2.2.3. Client side error codes . . . 114
5.2.2.4. Server side error codes: . . . 115
5.2.3. **MUST** use most specific HTTP status codes . . . 115
5.2.4. **MUST** use code 207 for batch or bulk requests . . . 115
5.2.5. **MUST** use code 429 with headers for rate limits . . . 116
5.2.6. **MUST** use problem JSON . . . 117
5.2.7. **MUST** not expose stack traces . . . 118
6. Data formats . . . 118
6.1. Data formats . . . 118
6.1.1. **MUST** use JSON to encode structured data . . . 118
6.1.2. **MAY** use non JSON media types for binary data or alternative content representations . . . 119
6.1.2.1. **SHOULD** encode embedded binary data in `base64url` . . . 119
6.1.3. **SHOULD** prefer standard media type name `application/json` . . . 119
6.1.4. **SHOULD** use standardized property formats . . . 119
6.1.5. **MUST** use standard date and time formats . . . 120
6.1.5.1. JSON payload . . . 120
6.1.5.2. HTTP headers . . . 120
6.1.6. **SHOULD** use standards for country, language and currency codes . . . 120
6.1.7. **MUST** define format for number and integer types . . . 121
6.2. JSON guidelines . . . 121
6.2.1. **MUST** property names must be ASCII snake_case (and never camelCase): `^[a-z_][a-z_0-9]*$` . . . 122
6.2.2. **MUST** declare enum values using UPPER_SNAKE_CASE format . . . 122
6.2.3. **SHOULD** define maps using `additionalProperties` . . . 122
6.2.4. **SHOULD** pluralize array names . . . 123
6.2.5. **MUST** not use `null` for boolean properties . . . 123
6.2.6. **MUST** use same semantics for `null` and absent properties . . . 123
6.2.7. **SHOULD** not use `null` for empty arrays . . . 124
6.2.8. **SHOULD** represent enumerations as strings . . . 124

6.2.9. **SHOULD** name date/time properties with `_at` suffix . . . 124
6.2.10. **SHOULD** define dates properties compliant with RFC 3339 . . . 125
7. Appendices . . . 125
Appendix G: Changelog . . . 125
7.G.1. Rule Changes . . . 125
7.1. Bibliography . . . 125
Generic . . . 126
Coding standards . . . 126
Open API specification . . . 126
Publications, specifications and standards . . . 126
Dissertations . . . 127
Books . . . 127

# 1. Introduction

> Progress is made by lazy persons looking for more efficient ways to do things.
>
> — paraphrased; Robert A. Heinlein

Software and software ecosystems are resolutions for business challenges, which in effect will result in a firm reliance of the business on the software and corresponding integrations. Moreover, our entrepreneurial mindset will drive updates, changes, and additions to the software; software is rarely "done" or "finished."

The business challenges and corresponding software updates are where these guidelines come in. To a certain degree, how can we guide the software development to ensure the software is stable, does what it should do, is secure, and is updateable with reasonable effort.

These guidelines and standards do not prescribe the "perfect" way to build software and software integrations since there is no "perfect" way. However, the consistency and structure of the code and integrations are specified. Summarized: solve the same problem with the same solution.

This document will enable the business to make informed decisions about software development: quantify the software's quality and robustness in an automated, consistent manner; be aware of the quality and robustness and decide upon the required level.

Moreover, this document contains several standards regarding intellectual property (IP) and security. To assure correct undisputed ownership of the software and maintain a security baseline.

In the following chapters, software refers both to software and software integrations.

## 1.1. Software engineering

In the mid-1960s, the field of software engineering emerged; building software is about more than just writing instructions for a computer. For the business to be able to use the software, these

guidelines address the following points [Wikipedia]:

- Software requirements
- Software architecture and design
- Software development
- Software testing
- Software maintenance

### 1.1.1. Software requirements

Establish the needs of the business to be resolved by software. This involves gathering the requirements and specifying the goals of the software to be met [Wikipedia].

Next to the business requirements, this document contains the minimal standards for software, including software license and source code management.

> If you want to build an application, don't herd people together to build code and don't assign them tasks and work, but rather teach them to long for the endless joy of happy, efficient users.
>
> — paraphrased; Antoine de Saint-Exupery

### 1.1.2. Software architecture and design

Architecture and design are where the Art of Building Software comes in; the architecture and design are the foundation of the software. This document will guide towards selecting the architecture and creating the design. There are some specific don'ts and do's; however, it is up to the architect to create the optimal design by considering all business requirements [Wikipedia].

### 1.1.3. Software development

How much effort is required for another developer of comparable experience to pick up where the previous developer left off to fix, enhance or build upon the source code - without involving the former developer and considering the lifetime, quality, security, and the business impact of the application.

### 1.1.4. Software testing

The proof of the pudding is in the eating; a good cook will sample the dish before serving the guests. The software builds upon requirements, and it is up to the software engineers to indicate the proven level of adherence to these requirements, preferably in an automated manner. It is up to the business to decide on the level of awareness regarding the software quality based on testing.

### 1.1.5. Software maintenance and support

When deploying the latest version of the software, there are two continuously ongoing tasks. Firstly: bug fixing, updates, and new features; require the software to be flexible, adaptable, and

logically structured. Note there is an overlap with software development: clean quality code will require a lower effort to adjust or fix instead of ill-structured low-quality code.

Secondly: the software's level of availability and performance; requires the software to be monitorable and report its well-being. There is overlap with software testing: checking if the software is adhering to requirements can also be reused in the live environment to check if the software is operational and responding within a predefined timeframe.

## 1.2. Document management

This document is managed using Git. Git allows for controlled change approval, versioning, and tracking of updates and changes. Moreover, it is an open system allowing for community change suggestions.

## 1.3. Attribution

These guidelines are based on the Zalando restful API guidelines.

## 1.4. Conventions used in these guidelines

The requirement level keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" used in this document (case insensitive) are to be interpreted as described in RFC 2119.

*Standards*

MUST is by default a standard and not a guideline.

## 1.5. Pon specific information

These guidelines apply to all software owned by Pon, built by Pon, or made for Pon, specifically, where Pon owns the IP. Note that this excludes SaaS solutions.

The purpose of these guidelines is to define standards to successfully establish "consistent integration and development look and feel" quality. The integration and development guilds draft and own this document. Teams are responsible for fulfilling these guidelines during development and are encouraged to contribute to guideline evolution via pull requests.

These guidelines will, to some extent, remain a work in progress as our profession evolves, but teams can confidently follow and trust them.

In case guidelines are changing, the following rules apply:

- existing software and software integrations don't have to be changed, but we recommend it
- clients of existing APIs have to cope with these APIs based on outdated rules
- new APIs have to respect the current guidelines

Furthermore, you should keep in mind that once an API becomes publicly available, it has to be re-

reviewed and changed according to current guidelines for overall consistency.

### 1.5.1. Automated code style checking (linting)

Digital Solutions will provide a preferred code style configuration per language or application that is retrievable from this document's repository.

Code styles are approved and maintained by the software development guild.

### 1.5.2. Guideline or standard

An additional "S" to the rule number and green left margin indicates a Pon standard.

### 1.5.3. MUST comply with standards and guidelines

Contracts relating to software development must indicate that the signing party has knowledge of the guidelines and agrees to comply with the standards as shown in this document.

### 1.5.4. (RFP) MUST write all resources using U.S. English

All resources, APIs, documentation, comments, etc., must be written in the U.S. English language.

# 2. Generic

## 2.1. Guilds

This section descibes the Guilds structure and the way of working.

### 2.1.1. Communication

This section describes the guilds WOW regarding communication

#### 2.1.1.1. Ad-hoc

Go / no-go

#### 2.1.1.2. New rules updates

How to handle rule updates and adding new rules.

## 2.2. Monitoring

Monitoring of systems is based on KPIs set by the business

### 2.2.1. (RFP) MUST have predefined monitoring based KPIs

All systems must have predifined KPIs on which the monitoring is to be based. These KPIs are the foundation for the monitoring of the infrastructure components.

For example: a website should have KPIs indicating response time and uptime. A low response time will require a fast scaling infrastructure and monitoring will have to be setup accordingly. A high uptime will require fast responding infrastructure monitoring.

### 2.2.2. (RFP) MUST be able monitor operational state using automated tools

All systems must have capabilities to allow itself to be monitored in an automated way and respond unambigous about the current operational state.

### 2.2.3. (RFP) SHOULD be able to report detailed relevant operational states

All systems should be able to report its operational state with enough detail for operational and support teams to take preventive measures. Basic reporting of "up" and "down" is not sufficient.

For example: system response time should be reported in milliseconds instead of "oke" and "not oke", allowing the business to set guidelines for the operational teams when to respond and take measures before the response time degrades to a level it will impact the effectiveness of its users.

## 2.3. Quality

### 2.3.1. (RFP) SHOULD have seperate environments for development, testing, acceptance and production

In order to maintain software or integration quality the setup of an DTAP environment is preferred. This setup allows for an atomized way of working on features and bug fixes. These environments should be available before starting development.

This environment consists of four parts:

#### 2.3.1.1. Development

The development environment is used for the actual development or for example bug fixing.

#### 2.3.1.2. Testing

The testing environment is the first step for quality assurance, in this environment the automated or manual tests are done to confirm new functionalities or bug fixes. Note there can be several testing environments, eg: an environment for the testing of an operational bug and an environment for the testing of a new feature.

#### 2.3.1.3. Acceptance

The acceptance environment is the second step for quality assurance, this environment allows for the customer to confirm new functionalities or bug fixes. In this environment several updates can be combined to prepare for a deployment to the production systems.

This environment allows for the customer to sign of on the development.

#### 2.3.1.4. Production

The primary environment for production. This environment can only be updated after a signoff of which the requirements are specified beforehand.

### 2.3.2. Code Quality

The quality of the code should be known, but we diffrentiate between frontends and api's. GUI are more difficult to test, because you'll need to automate the click flow through the frontend/GUI and therefor more labour intensive to maintain.

All tests and checks should be configured in the pipeline so the developer receives instant feedback of the commit.

#### 2.3.2.1. (RFP) MUST have code duplication checks

A software project must have something in place to detect duplicated code.

#### 2.3.2.2. (RFP) MUST have vulnerability checks

When running in the pipeline a vulnerability check must happen, so compromised libraries are not taken into production (because the pipeline is blocked).

### 2.3.3. GUI and websites quality

#### 2.3.3.1. (RFP) SHOULD have predefined tests

A GUI project should have some tests defined, for example; a webshop should be able to have an item in the cart and finalize the checkout.

### 2.3.4. API quality

API's need to have a certain quality level which can be observed.

#### 2.3.4.1. (RFP) MUST have predefined tests

An API must have Unit / feature / e2e tests available. This tells something about how well you can trust what to expect from your code.

#### 2.3.4.2. (RFP) MUST have a pre-defined code-coverage percentage

An API must have a pre-defined level of code-coverage. This tells something about how well the code is tested and if all code paths have been checked.

## 2.4. Security

This section descibes the generic guidelines for security.

Security and Privacy within Pon is being managed by the Pon Security & Privacy Office. All Pon Business Clusters have a dedicated Group Security and Privacy Manager.

Additional resources

- Pon Security & Privacy by Design Process - Step by Step
- Data classification and corresponding security measures

### 2.4.1. (RFP) MUST be compliant with the Pon Security Policy and Principles

All systems must be compliant with the Pon Security Policy and Principles

### 2.4.2. (RFP) MUST be compliant with Binding Corporate Rules and local privacy legislation

All systems must be compliant with Binding Corportate Rules and local privacy legislation such as GDPR or CCPA.

### 2.4.3. (RFP) MUST have performed a Security & Privacy intake

Must have performed a Security & Privacy intake. When the requirements for the new system are clear, the Pon Security & Privacy by Design Process must be triggered via completing the Security & Privacy by Design Intake Form. The intake process is there to help you create APIs that are secure and privacy compliant and friendly. Your Group Security and Privacy manager will contact you to discuss the intake.

### 2.4.4. (RFP) MUST have the agreed security and privacy measures approved

Must have the agreed security and privacy measures approved by Security and/or Privacy Manager before going live

## 2.5. Privacy

This section descibes the generic guidelines for privacy.

## 2.6. Documentation

This section descibes the generic guidelines for documentation.

### 2.6.1. MUST include documentation comment saying what the tool is for

Every tool, program or integration must include a comment, either in a file or in the source briefly indicating the purpose.

If the comment is included in the source, it is preferred to be included in the file containing the main function

### 2.6.2. MUST include monitoring documentation

Every tool, program or integration must include a comment, either in a file or in the source inidicating how the software operational state can be checked or monitored by automated systems.

#### 2.6.2.1. References

- [gnu-coding-standards], Chapter 5.2

### 2.6.3. MUST document deployment procedures

Deployment of the tool, program or integration must be documented. This documentation describes the deployment procedures and required resources. Enabling an engineer to deploy without any support from the developers.

## 2.7. Onboarding

Onboarding of new developers for Pon or onboarding for new developers in a project.

### 2.7.1. (RFP) MUST include guidelines in onboarding procedure

The onboarding procedure for new developers must include familiarization with these guidelines.

### 2.7.2. (RFP) SHOULD have pre-defined development environment(s)

There should be pre-defined development environments, including OS and IDE. These environments will ensure a smooth onboarding and will make teams of developers more efficient.

#### 2.7.2.1. Patron(s) 

- Zeger Knops (zeger.knops@pon.com) = Solution architecture

Pon's software architecture centers around decoupled microservices that provide functionality via APIs. Small engineering teams own, deploy and operate these microservices. Our APIs most purely express what our systems do, and are therefore highly valuable business assets.

With this in mind, we've adopted "API First" as one of our key engineerings principles. Microservices development begins with an API definition outside the code and ideally involves ample peer-review feedback to achieve high-quality APIs. API First encompasses a set of quality-related standards and fosters a peer-review culture, including a lightweight review procedure. We encourage our teams to follow them to ensure that our APIs:

- are easy to understand and learn
- are general and abstracted from specific implementation and use cases
- are robust and easy to use
- have a common look and feel
- follow a consistent RESTful style and syntax
- are consistent with other teams' APIs and our global architecture

Ideally, all Pon APIs will look like the same author created them.

### 2.7.3. Solution architecture repository

The Pon guilds supply a repository of approved tools, techniques and frameworks both to inspire and guide teams. It is essential this repository is consulted when designing a solution architecture.

## 2.8. Intellectual property (IP)

### 2.8.1. (RFP) MUST include project wide license file

The source code of each project must include a Pon approved license file.

**/** * Copyright © Pon Holding - All Rights Reserved */**

### 2.8.2. (RFP) MUST include copyright notice in each source file

All applicable source files must contain a Pon approved copyright notice at the start of the file and a reference to the project license file.

Example:

**/** * Copyright © Pon Holding - All Rights Reserved * Unauthorized copying of this file, via any medium is strictly prohibited * Proprietary and confidential * * This file is subject to the terms and conditions defined in * file 'LICENSE.txt', which is part of this source code package. * * Written by Pon Employee <pon.employee@pon.com>, September 2020 */**

## 2.9. Checklist

- License files checked and added to project **(RFP) MUST** include project wide license file?
- All resources in U.S. English **(RFP) MUST** write all resources using U.S. English?
- Documentation available for both code and deployment?
- Security guidelines and standards checked?
- Monitoring documented, KPIs discussed and setup with ops team?
- Quality assurance automated and results readily available?

# 3. Integration guidelines

## 3.1. Principles

### 3.1.1. API design principles

Comparing SOA web service interfacing style of SOAP vs. REST, the former tend to be centered around operations that are usually use-case specific and specialized. In contrast, REST is centered around business (data) entities exposed as resources that are identified via URIs and can be manipulated via standardized CRUD-like methods using different representations, and hypermedia. RESTful APIs tend to be less use-case specific and comes with less rigid client / server coupling and

are more suitable for an ecosystem of (core) services providing a platform of APIs to build diverse new business services. We apply the RESTful web service principles to all kind of application (micro-) service components, independently from whether they provide functionality via the internet or intranet.

- We prefer REST-based APIs with JSON payloads
- We prefer systems to be truly RESTful [1]

An important principle for API design and usage is Postel's Law, aka The Robustness Principle (see also RFC 1122):

- Be liberal in what you accept, be conservative in what you send

*Readings:* Some interesting reads on the RESTful API design style and service architecture:

- 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

### 3.1.2. API as a product

The design of our APIs should be based on the API as a Product principle:

- Treat your API as product and act like a product owner
- Put yourself into the place of your customers; be an advocate for their needs
- Emphasize simplicity, comprehensibility, and usability of APIs to make them irresistible for client engineers
- Actively improve and maintain API consistency over the long term
- Make use of customer feedback and provide service level support

Embracing 'API as a Product' facilitates a service ecosystem which can be evolved more easily, and used to experiment quickly with new business ideas by recombining core capabilities. It makes the difference between agile, innovative product service business built on a platform of APIs and ordinary enterprise integration business where APIs are provided as "appendix" of existing products to support system integration and optimised for local server-side realization.

Understand the concrete use cases of your customers and carefully check the trade-offs of your API design variants with a product mindset. Avoid short-term implementation optimizations at the expense of unnecessary client side obligations, and have a high attention on API quality and client developer experience.

API as a Product is closely related to our API First principle (see next chapter) which is more

focused on how we engineer high quality APIs.

### 3.1.3. API first

Refer to Pon achitecture principles

API First is one of our engineering and architecture principles. In a nutshell API First requires two aspects:

- define APIs first, before coding its implementation, using a standard specification language
- get early review feedback from peers and client developers

By defining APIs outside the code, we want to facilitate early review feedback and also a development discipline that focus service interface design on…

- profound understanding of the domain and required functionality
- generalized business entities / resources, i.e. avoidance of use case specific APIs
- clear separation of WHAT vs. HOW concerns, i.e. abstraction from implementation aspects — APIs should be stable even if we replace complete service implementation including its underlying technology stack

Moreover, API definitions with standardized specification format also facilitate…

- single source of truth for the API specification; it is a crucial part of a contract between service provider and client users
- infrastructure tooling for API discovery, API GUIs, API documents, automated quality checks

Elements of API First are also this API Guidelines and a standardized API review process as to get early review feedback from peers and client developers. Peer review is important for us to get high quality APIs, to enable architectural and design alignment and to supported development of client applications decoupled from service provider engineering life cycle.

It is important to learn, that API First is **not in conflict with the agile development principles** that we love. Service applications should evolve incrementally — and so its APIs. Of course, our API specification will and should evolve iteratively in different cycles; however, each starting with draft status and *early* team and peer review feedback. API may change and profit from implementation concerns and automated testing feedback. API evolution during development life cycle may include breaking changes for not yet productive features and as long as we have aligned the changes with the clients. Hence, API First does *not* mean that you must have 100% domain and requirement understanding and can never produce code before you have defined the complete API and get it confirmed by peer review. On the other hand, API First obviously is in conflict with the bad practice of publishing API definition and asking for peer review after the service integration or even the service productive operation has started. It is crucial to request and get early feedback — as early as possible, but not before the API changes are comprehensive with focus to the next evolution step and have a certain quality (including API Guideline compliance), already confirmed via team internal reviews.

# 3.2. Solution design

This section descibes the generic guidelines for integration solution design.

## 3.2.1. Microservices

## 3.2.2. Connectivity

#### 3.2.2.1. (RFP) MUST have self-healing connectivity

External connections (e.g. SFTP, Database, SAP or Email) need to be monitored and automatically reset and restored when connection issues occur. The frequency for these reconnections and after how many retries to send an alert needs to be agreed upon with the business.

However, if no such agreements are in place, we set as default to retry once every minute and send an alert when reconnection has failed for an hour. After the alert has been sent, reconnection attempts must continue until successful.

#### 3.2.2.2. (RFP) SHOULD have increasing reconnection intervals

When reconnection strategies are deployed they should have increasing reconnection intervals. For example 1s, 2s, 4s etc.

## 3.2.3. Loosely coupled

## 3.2.4. Security

## 3.2.5. Monitoring

#### 3.2.5.1. (RFP) MUST setup monitoring and alerting connections

All connections must be monitored and alerting has to be setup based on the monitoring.

## 3.2.6. Documentation

## 3.2.7. Security

# 3.3. Generic

## 3.3.1. Pagination

#### 3.3.1.1. MUST support pagination

Access to lists of data items must support pagination to protect the service against overload as well as for best client side iteration and batch processing experience. This holds true for all lists that are (potentially) larger than just a few hundred entries.

There are two well known page iteration techniques:

- Offset/Limit-based pagination: numeric offset identifies the first page entry
- Cursor/Limit-based — aka key-based — pagination: a unique key element identifies the first page entry (see also Facebook's guide)

The technical conception of pagination should also consider user experience related issues. As mentioned in this article, jumping to a specific page is far less used than navigation via `next`/`prev` page links (See **SHOULD** use pagination links where applicable). This favours cursor-based over offset-based pagination.

**Note:** To provide a consistent look and feel of pagination patterns, you must stick to the common query parameter names defined in **MUST** stick to conventional query parameters.

#### 3.3.1.2. SHOULD prefer cursor-based pagination, avoid offset-based pagination

Cursor-based pagination is usually better and more efficient when compared to offset-based pagination. Especially when it comes to high-data volumes and/or storage in NoSQL databases.

Before choosing cursor-based pagination, consider the following trade-offs:

- Usability/framework support:
  - Offset-based pagination is more widely known than cursor-based pagination, so it has more framework support and is easier to use for API clients
- Use case - jump to a certain page:
  - If jumping to a particular page in a range (e.g., 51 of 100) is really a required use case, cursor-based navigation is not feasible.
- 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.
- Performance considerations - efficient server-side processing using offset-based pagination is hardly feasible for:
  - Very big data sets, especially if they cannot reside in the main memory of the database.
  - Sharded or NoSQL databases.
- Cursor-based navigation may not work if you need the total count of results.

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 (see also Link relation fields):

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

**Note:** The support of the `cursor` set may be dropped in favor of **SHOULD** use pagination links where applicable.

Further reading:

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

#### 3.3.1.3. SHOULD use pagination links where applicable

To simplify client design, APIs should support simplified hypertext controls for pagination over collections whenever applicable. Beside `next` this may comprise the support for `prev`, `first`, `last`, and `self` as link relations (see also Link relation fields for details).

The page content is transported via `items`, while the `query` object may contain the query filters applied to the collection resource as follows:

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

**Note:** In case of complex search requests, e.g. when `GET With Body` is required, the `cursor` may not be able to encode all query filters. In this case, it is best practice to encode only page position and direction in the `cursor` and transport the query filter in the body - in the request as well as in the response. To protect the pagination sequence, in this case it is recommended, that the `cursor` contains a hash over all applied query filters for pagination request validation.

**Remark:** You should avoid providing a total count unless there is a clear need to do so. Very often, there are significant system and performance implications when supporting full counts. Especially, if the data set grows and requests become complex queries and filters drive full scans. While this is an implementation detail relative to the API, it is important to consider the ability to support serving counts over the life of a service.

# 3.4. Types

## 3.4.1. FTP

#### 3.4.1.1. Monitoring

#### 3.4.1.2. Documentation

#### 3.4.1.3. Security

# 3.5. General guidelines

The titles are marked with the corresponding labels: **MUST**, **SHOULD**, **MAY**.

### 3.5.1. (RFP) MUST follow API first principle

You must follow the API First Principle, more specifically:

- You must define APIs first, before coding its implementation.
- You must call for review feedback from peers and client developers.

### 3.5.2. (RFP) MUST provide API specification

Must provide API specification according to standards as specified for API platform.

The API specification files should be subject to version control using a source code management system - best together with the implementing sources.

You must / should publish the component external / internal API specification with the deployment of the implementing service, and, hence, make it discoverable.

### 3.5.3. (RFP) MUST only use durable and immutable remote references

Normally, API specification files must be **self-contained**, i.e. files should not contain references to local or remote content, e.g. `../fragment.yaml#/element` or `$ref: 'https://github.com/zalando/zally/blob/master/server/src/main/resources/api/zally-api.yaml#/schemas/LintingRequest'`.

### 3.5.4. (RFP) MAY provide API user manual

In addition to the API Specification, it is good practice to provide an API user manual to improve client developer experience, especially of engineers that are less experienced in using this API. A helpful API user manual typically describes the following API aspects:

- API scope, purpose, and use cases
- concrete examples of API usage
- edge cases, error situation details, and repair hints
- architecture context and major dependencies - including figures and sequence flows

The user manual must be published online, e.g. via our documentation hosting platform service, GHE pages, or specific team web servers. Please do not forget to include a link to the API user manual into the API specification using the `#/externalDocs/url` property.

# 3.6. Meta information

### 3.6.1. (RFP) SHOULD contain API meta information

API specifications must contain the following meta information to allow for API management:

- `#/info/title` as (unique) identifying, functional descriptive name of the API
- `#/info/version` to distinguish API specifications versions following semantic rules
- `#/info/description` containing a proper description of the API
- `#/info/contact/{name,url,email}` containing the responsible team
- `#/info/api-id` unique identifier of the API (see rule 215)
- `#/info/audience` intended target audience of the API (see rule 219)

### 3.6.2. (RFP) MAY use semantic versioning

Open API allows to specify the API specification version in `#/info/version`. To share a common semantic of version information we expect API designers to comply to Semantic Versioning 2.0 rules 1 to 8 and 11 restricted to the format <MAJOR>.<MINOR>.<PATCH> for versions as follows:

- Increment the **MAJOR** version when you make incompatible API changes after having aligned this changes with consumers,
- Increment the **MINOR** version when you add new functionality in a backwards-compatible manner, and
- Optionally increment the **PATCH** version when you make backwards-compatible bug fixes or editorial changes not affecting the functionality.

**Additional Notes:**

- **Pre-release** versions (rule 9) and **build metadata** (rule 10) must not be used in API version information.
- While patch versions are useful for fixing typos etc, API designers are free to decide whether they increment it or not.
- API designers should consider to use API version `0.y.z` (rule 4) for initial API design.

Example:

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

### 3.6.3. (RFP) MAY provide API identifiers

Each API specification may be provisioned with a globally unique and immutable API identifier.

```
/info/api-id:
  type: string
  format: urn
  pattern: ^[a-z0-9][a-z0-9-:.]{6,62}[a-z0-9]$
  description: |
    Mandatory globally unique and immutable API identifier. The API
    id allows to track the evolution and history of an API specification
    as a sequence of versions.
```

API specifications will evolve and any aspect of an API specification may change. We require API identifiers because we want to support API clients and providers with API lifecycle management features, like change trackability and history or automated backward compatibility checks. The immutable API identifier allows the identification of all API specification versions of an API evolution. By using API semantic version information or API publishing date as order criteria you get the **version** or **publication history** as a sequence of API specifications.

**Note**: While it is nice to use human readable API identifiers based on self-managed URNs, it is recommend to stick to UUIDs to relief API designers from any urge of changing the API identifier while evolving the API. Example:

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

### 3.6.4. (RFP) SHOULD provide API audience

Each API must be classified with respect to the intended target **audience** supposed to consume the API, to facilitate differentiated standards on APIs for discoverability, changeability, quality of design and documentation, as well as permission granting. We differentiate the following API audience groups with clear organisational and legal boundaries:

**component-internal**

This is often referred to as a *team internal API* or a *product internal API*. The API consumers with this audience are restricted to applications of the same **functional component** which typically represents a specific **product** with clear functional scope and ownership. All services of a functional component / product are owned by a specific dedicated owner and engineering team(s). Typical examples of component-internal APIs are APIs being used by internal helper and worker services or that support service operation.

**business-unit-internal**

The API consumers with this audience are restricted to applications of a specific product portfolio owned by the same business unit.

**company-internal**

The API consumers with this audience are restricted to applications owned by the business units of the same the company.

**external-partner**

The API consumers with this audience are restricted to applications of business partners of the company owning the API and the company itself.

**external-public**

APIs with this audience can be accessed by anyone with Internet access.

**Note:** a smaller audience group is intentionally included in the wider group and thus does not need to be declared additionally.

The API audience is provided as API meta information in the `info`-block of the Open API specification and must conform to the following specification:

```
/info/x-audience:
  type: string
  x-extensible-enum:
    - component-internal
    - business-unit-internal
    - company-internal
    - external-partner
    - external-public
  description: |
    Intended target audience of the API. Relevant for standards around
    quality of design and documentation, reviews, discoverability,
    changeability, and permission granting.
```

**Note:** Exactly **one audience** per API specification is allowed. For this reason a smaller audience group is intentionally included in the wider group and thus does not need to be declared additionally. If parts of your API have a different target audience, we recommend to split API specifications along the target audience.

Example:

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

For details and more information on audience groups see the Pon internal documentation

TODO: add link to internal documentation regarding API audience issue 2

# 3.7. Security

## 3.7.1. (RFP) MUST secure endpoints

Every API endpoint should be secured, also for anonymous access. The preferred authentication method is OAuth 2.0. For anonymous access the client credentials grant is preferred.

The following code snippet shows how to define the authorization scheme using a bearer token (e.g. JWT token).

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

The next code snippet applies this security scheme to all API endpoints. The bearer token of the client must have additionally the scopes "scope_1" and "scope_2".

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

### 3.7.1.1. References

- OAuth.net - Client Credentials Grant
- OKTA - Server to server auth
- IBM - Anonymous authentication
- RFC6749 - Client credentials grant

## 3.7.2. (RFP) SHOULD define and assign permissions (scopes)

APIs should define permissions to protect their resources. Thus, at least one permission must be assigned to each endpoint. Permissions are defined as shown in the previous section.

The naming schema for permissions corresponds to the naming schema for hostnames and event type names. Please refer to **(RFP) MAY** follow naming convention for permissions (scopes) for designing permission names.

APIs should stick to component specific permissions without resource extension to avoid governance complexity of too many fine grained permissions. For the majority of use cases, restricting access to specific API endpoints using read and write is sufficient for controlling access for client types like merchant or retailer business partners, customers or operational staff. However, in some situations, where the API serves different types of resources for different owners, resource specific scopes may make sense.

Some examples for standard and resource-specific permissions:

| 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 |

After permission names are defined and the permission is declared in the security definition at the top of an API specification, it should be assigned to each API operation by specifying a security requirement like this:

```
paths:
 /business-partners/{partner-id}:
    get:
      summary: Retrieves information about a business partner
      security:
        - BearerAuth: [ business-partner-service.read ]
```

In very rare cases a whole API or some selected endpoints may not require specific access control. However, to make this explicit you should assign the `uid` pseudo permission in this case. It is the user id and always available as OAuth2 default scope.

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

Hint: you need not explicitly define the "Authorization" header; it is a standard header so to say implicitly defined via the security section.

### 3.7.3. (RFP) MAY follow naming convention for permissions (scopes)

As long as the functional naming is not supported for permissions, permission names in APIs must conform to the following naming pattern:

```
<permission> ::= <standard-permission> |  -- should be sufficient for majority of use
cases
                 <resource-permission> |  -- for special security access
differentiation use cases
                 <pseudo-permission>      -- used to explicitly indicate that access
is not restricted
```

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

<application-id>      ::= [a-z][a-z0-9-]*  -- application identifier
<resource-name>       ::= [a-z][a-z0-9-]*  -- free resource identifier
<access-mode>         ::= read | write    -- might be extended in future
```

This pattern is compatible with the previous definition.

# 3.8. Compatibility

## 3.8.1. (RFP) MUST not break backward compatibility

Change APIs, but keep all consumers running. Consumers usually have independent release lifecycles, focus on stability, and avoid changes that do not provide additional value. APIs are contracts between service providers and service consumers that cannot be broken via unilateral decisions.

There are two techniques to change APIs without breaking them:

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

We strongly encourage using compatible API extensions and discourage versioning (see **(RFP) SHOULD** avoid versioning and {STATUS-TODO} **MUST** API Versioning Has No “Right Way” below). The following guidelines for service providers (**(RFP) SHOULD** prefer compatible extensions) and consumers (**(RFP) MUST** prepare clients accept compatible API extensions) enable us (having Postel’s Law in mind) to make compatible changes without versioning.

**Note:** There is a difference between incompatible and breaking changes. Incompatible changes are changes that are not covered by the compatibility rules below. Breaking changes are incompatible changes deployed into operation, and thereby breaking running API consumers. Usually, incompatible changes are breaking changes when deployed into operation. However, in specific controlled situations it is possible to deploy incompatible changes in a non-breaking way, if no API consumer is using the affected API aspects (see also Deprecation guidelines).

**Hint:** Please note that the compatibility guarantees are for the "on the wire" format. Binary or source compatibility of code generated from an API definition is not covered by these rules. If client implementations update their generation process to a new version of the API definition, it has to be expected that code changes are necessary.

## 3.8.2. (RFP) SHOULD prefer compatible extensions

API designers may apply the following rules to evolve APIs for services in a backward-compatible way:

- Add only optional, never mandatory fields.
- Never change the semantic of fields (e.g. changing the semantic from customer-number to customer-id, as both are different unique customer keys)
- Input fields may have (complex) constraints being validated via server-side business logic. Never change the validation logic to be more restrictive and make sure that all constraints are clearly defined in description.
- Enum ranges can be reduced when used as input parameters, only if the server is ready to accept and handle old range values too. Enum range can be reduced when used as output parameters.
- Enum ranges cannot be extended when used for output parameters — clients may not be prepared to handle it. However, enum ranges can be extended when used for input parameters.
- Use `x-extensible-enum`, if range is used for output parameters and likely to be extended with growing functionality. It defines an open list of explicit values and clients must be agnostic to new values.
- Support redirection in case an URL has to change 301 (Moved Permanently).

### 3.8.3. (RFP) MUST prepare clients accept compatible API extensions

Service clients should apply the robustness principle:

- Be conservative with API requests and data passed as input, e.g. avoid to exploit definition deficits like passing megabytes of strings with unspecified maximum length.
- Be tolerant in processing and reading data of API responses, more specifically…

Service clients must be prepared for compatible API extensions of service providers:

- Be tolerant with unknown fields in the payload (see also Fowler's "TolerantReader" post), i.e. ignore new fields but do not eliminate them from payload if needed for subsequent `PUT` requests.
- Be prepared that `x-extensible-enum` return parameter may deliver new values; either be agnostic or provide default behavior for unknown values.
- Be prepared to handle HTTP status codes not explicitly specified in endpoint definitions. Note also, that status codes are extensible. Default handling is how you would treat the corresponding 2xx code (see RFC 7231 Section 6).
- Follow the redirect when the server returns HTTP status code 301 (Moved Permanently).

### 3.8.4. (RFP) SHOULD design APIs conservatively

Designers of service provider APIs should be conservative and accurate in what they accept from clients:

- Unknown input fields in payload or URL should not be ignored; servers should provide error feedback to clients via an HTTP 400 response code.
- Be accurate in defining input data constraints (like formats, ranges, lengths etc.) — and check constraints and return dedicated error information in case of violations.

- Prefer being more specific and restrictive (if compliant to functional requirements), e.g. by defining length range of strings. It may simplify implementation while providing freedom for further evolution as compatible extensions.

Not ignoring unknown input fields is a specific deviation from Postel's Law (e.g. see also The Robustness Principle Reconsidered) and a strong recommendation. Servers might want to take different approach but should be aware of the following problems and be explicit in what is supported:

- Ignoring unknown input fields is actually not an option for `PUT`, since it becomes asymmetric with subsequent `GET` response and HTTP is clear about the `PUT` *replace* semantics and default roundtrip expectations (see RFC 7231 Section 4.3.4). Note, accepting (i.e. not ignoring) unknown input fields and returning it in subsequent `GET` responses is a different situation and compliant to `PUT` semantics.
- Certain client errors cannot be recognized by servers, e.g. attribute name typing errors will be ignored without server error feedback. The server cannot differentiate between the client intentionally providing an additional field versus the client sending a mistakenly named field, when the client's actual intent was to provide an optional input field.
- Future extensions of the input data structure might be in conflict with already ignored fields and, hence, will not be compatible, i.e. break clients that already use this field but with different type.

In specific situations, where a (known) input field is not needed anymore, it either can stay in the API definition with "not used anymore" description or can be removed from the API definition as long as the server ignores this specific parameter.

### 3.8.5. (RFP) MUST always return JSON objects as top-level data structures if JSON is being used

In a JSON response body, you must always return a JSON object (and not e.g. an array) as a top level data structure to support future extensibility. JSON objects support compatible extension by additional attributes. This allows you to easily extend your response and e.g. add pagination later, without breaking backwards compatibility. See **SHOULD** use pagination links where applicable for an example.

Maps (see **SHOULD** define maps using `additionalProperties`), even though technically objects, are also forbidden as top level data structures, since they don't support compatible, future extensions.

{TODO} Add example

### 3.8.6. (RFP) SHOULD refrain from using enumerations

Enumerations are per definition closed sets of values, that are assumed to be complete and not intended for extension. This closed principle of enumerations imposes compatibility issues when an enumeration must be extended. To avoid these issues, we strongly recommend to use an open-ended list of values instead of an enumeration unless:

1. the API has full control of the enumeration values, i.e. the list of values does not depend on any

external tool or interface, and

2. the list of value is complete with respect to any thinkable and unthinkable future feature.
3. the values must be enforced.

To specify an open-ended list of values use the marker `x-extensible-enum` as follows:

```
delivery_methods:
  type: string
  x-extensible-enum:
    - PARCEL
    - LETTER
    - EMAIL
```

**Note:** `x-extensible-enum` is not JSON Schema conform but will be ignored by most tools.

See **MUST** declare enum values using UPPER_SNAKE_CASE format about enum value naming conventions.

### 3.8.7. (RFP) SHOULD avoid versioning

When changing your APIs, do so in a compatible way and avoid generating additional API versions unless the API is non-functional or is degraded. Multiple versions can significantly complicate understanding, testing, maintaining, evolving, operating and releasing our systems (supplementary reading).

If changing an API can't be done in a compatible way, then proceed in one of these three ways:

- create a new resource (variant) in addition to the old resource variant
- create a new service endpoint — i.e. a new application with a new API (with a new domain name)
- create a new API version supported in parallel with the old API by the same microservice

As we discourage versioning by all means because of the manifold disadvantages, we strongly recommend to only use the first two approaches.

### 3.8.8. {STATUS-TODO} MUST API Versioning Has No "Right Way"

API Versioning Has No "Right Way" provides an overview on different versioning approaches to handle breaking changes without being opinionated.

### 3.8.9. (RFP) SHOULD use URI versioning

With URI versioning a (major) version number is included in the path, e.g. `/v1/customers`. The consumer has to wait until the provider has been released and deployed.

# 3.9. Deprecation

Sometimes 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 the necessary consumer changes and actions are well communicated and aligned using *deprecation* and *sunset* dates.

## 3.9.1. (RFP) MUST obtain approval of clients before API shut down

Before shutting down an API, version of an API, or API feature the producer must make sure, that all clients have given their consent on a sunset date. Producers should help consumers to migrate to a potential new API or API feature by providing a migration manual and clearly state the time line for replacement availability and sunset (see also **(RFP) SHOULD** add `Deprecation` and `Sunset` header to responses). Once all clients of a sunset API feature are migrated, the producer may shut down the deprecated API feature.

## 3.9.2. (RFP) MUST collect external partner consent on deprecation time span

If the API is consumed by any external partner, the API owner must define a reasonable time span that the API will be maintained after the producer has announced deprecation. All external partners must state consent with this after-deprecation-life-span, i.e. the minimum time span between official deprecation and first possible sunset, **before** they are allowed to use the API.

## 3.9.3. (RFP) MUST reflect deprecation in API specifications

The API deprecation must be part of the API specification.

If an API endpoint (operation object), an input argument (parameter object), an in/out data object (schema object), or on a more fine grained level, a schema attribute or property should be deprecated, the producers must set `deprecated: true` for the affected element and add further explanation to the `description` section of the API specification. If a future shut down is planned, the producer must provide a sunset date and document in details what consumers should use instead and how to migrate.

## 3.9.4. (RFP) MUST monitor usage of deprecated API scheduled for sunset

Owners of an API, API version, or API feature used in production that is scheduled for sunset must monitor the usage of the sunset API, API version, or API feature in order to observe migration progress and avoid uncontrolled breaking effects on ongoing consumers. See also **SHOULD** monitor API usage.

## 3.9.5. (RFP) SHOULD add `Deprecation` and `Sunset` header to responses

During the deprecation phase, the producer should add a `Deprecation: <date-time>` (see draft: RFC Deprecation HTTP Header) and - if also planned - a `Sunset: <date-time>` (see RFC 8594) header on

each response affected by a deprecated element (see **(RFP) MUST** reflect deprecation in API specifications).

The `Deprecation` header can either be set to `true` - if a feature is retired -, or carry a deprecation time stamp, at which a replacement will become/became available and consumers must not on-board any longer (see **(RFP) MUST** not start using deprecated APIs). The optional `Sunset` time stamp carries the information when consumers latest have to stop using a feature. The sunset date should always offer an eligible time interval for switching to a replacement feature.

```
Deprecation: Sun, 31 Dec 2024 23:59:59 GMT
Sunset: Sun, 31 Dec 2025 23:59:59 GMT
```

If multiple elements are deprecated the `Deprecation` and `Sunset` headers are expected to be set to the earliest time stamp to reflect the shortest interval consumers are expected to get active.

**Note:** adding the `Deprecation` and `Sunset` header is not sufficient to gain client consent to shut down an API or feature.

**Hint:** In earlier guideline versions, we used the `Warning` header to provide the deprecation info to clients. However, `Warning` header has a less specific semantics, will be obsolete with draft: RFC HTTP Caching, and our syntax was not compliant with RFC 7234 — Warning header.

### 3.9.6. (RFP) SHOULD add monitoring for `Deprecation` and `Sunset` header

Clients should monitor the `Deprecation` and `Sunset` headers in HTTP responses to get information about future sunset of APIs and API features (see **(RFP) SHOULD** add `Deprecation` and `Sunset` header to responses). We recommend that client owners build alerts on this monitoring information to ensure alignment with service owners on required migration task.

**Hint:** In earlier guideline versions, we used the `Warning` header to provide the deprecation info (see hint in **(RFP) SHOULD** add `Deprecation` and `Sunset` header to responses).

### 3.9.7. (RFP) MUST not start using deprecated APIs

Clients must not start using deprecated APIs, API versions, or API features.

## 3.10. Common data types

Definitions of data objects that are good candidates for wider usage:

### 3.10.1. (RFP) MUST use the common money object

Use the following common money structure:

```yaml
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
```

APIs are encouraged to include a reference to the global schema for Money.

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

Please note that APIs have to treat Money as a closed data type, i.e. it's not meant to be used in an inheritance hierarchy. That means the following usage is not allowed:

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

#### 3.10.1.1. Cons

- Violates the Liskov Substitution Principle
- Breaks existing library support, e.g. Jackson Datatype Money
- Less flexible since both amounts are coupled together, e.g. mixed currencies are impossible

A better approach is to favor composition over inheritance:

```
{
    "price": {
      "amount": 19.99,
      "currency": "EUR"
```

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

#### 3.10.1.2. Pros

- No inheritance, hence no issue with the substitution principle
- Makes use of existing library support
- No coupling, i.e. mixed currencies is an option
- Prices are now self-describing, atomic values

#### 3.10.1.3. Notes

Please be aware that some business cases (e.g. transactions in Bitcoin) call for a higher precision, so applications must be prepared to accept values with unlimited precision, unless explicitly stated otherwise in the API specification.

Examples for correct representations (in EUR):

- `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

Make sure that you don't convert the "amount" field to `float` / `double` types when implementing this interface in a specific language or when doing calculations. Otherwise, you might lose precision. Instead, use exact formats like Java's `BigDecimal`. See Stack Overflow for more info.

Some JSON parsers (NodeJS's, for example) convert numbers to floats by default. After discussing the pros and cons we've decided on "decimal" as our amount format. It is not a standard Open API format, but should help us to avoid parsing numbers as float / doubles.

### 3.10.2. (RFP) MUST use common field names and semantics

There exist a variety of field types that are required in multiple places. To achieve consistency across all API implementations, you must use common field names and semantics whenever applicable.

#### 3.10.2.1. Generic fields

There are some data fields that come up again and again in API data:

- `id`: the identity of the object. If used, IDs must be opaque strings and not numbers. IDs are

unique within some documented context, are stable and don't change for a given object once assigned, and are never recycled cross entities.

- `xyz_id`: an attribute within one object holding the identifier of another object must use a name that corresponds to the type of the referenced object or the relationship to the referenced object followed by `_id` (e.g. `partner_id` not `partner_number`, or `parent_node_id` for the reference to a parent node from a child node, even if both have the type `Node`). **Exception**: We use `customer_number` instead of `customer_id` for customer facing identification of customers due to legacy reasons.
- `created_at`: when the object was created. If used, this must be a `date-time` construct. Originally named `created` before adding the naming conventions for date/time properties.
- `modified_at`: when the object was updated. If used, this must be a `date-time` construct. Originally named `modified` before adding the naming conventions for date/time properties.
- `type`: the kind of thing this object is. If used, the type of this field should be a string. Types allow runtime information on the entity provided that otherwise requires examining the Open API file.
- `ETag`: the ETag of an embedded sub-resource. It may be used to carry the `ETag` for subsequent `PUT`/`PATCH` calls (see `ETags` in result entities).

Example JSON schema:

```
tree_node:
  type: object
  properties:
    id:
      description: the identifier of this node
      type: string
    created_at:
      description: when got this node created
      type: string
      format: 'date-time'
    modified_at:
      description: when got this node last updated
      type: string
      format: 'date-time'
    type:
      type: string
      enum: [ 'LEAF', 'NODE' ]
    parent_node_id:
      description: the identifier of the parent node of this node
      type: string
  example:
    id: '123435'
    created_at: '2017-04-12T23:20:50.52Z'
    modified_at: '2017-04-12T23:20:50.52Z'
    type: 'LEAF'
    parent_node_id: '534321'
```

These properties are not always strictly necessary, but making them idiomatic allows API client developers to build up a common understanding of Pon's resources. There is very little utility for API consumers in having different names or value types for these fields across APIs.

#### 3.10.2.2. Link relation fields

To foster a consistent look and feel using simple hypertext controls for paginating and iterating over collection values the response objects should follow a common pattern using the below field semantics:

- `self`:the link or cursor in a pagination response or object pointing to the same collection object or page.
- `first`: the link or cursor in a pagination response or object pointing to the first collection object or page.
- `prev`: the link or cursor in a pagination response or object pointing to the previous collection object or page.
- `next`: the link or cursor in a pagination response or object pointing to the next collection object or page.
- `last`: the link or cursor in a pagination response or object pointing to the last collection object or page.

Pagination responses should contain the following additional array field to transport the page content:

- `items`: array of resources, holding all the items of the current page (`items` may be replaced by a resource name).

To simplify user experience, the applied query filters may be returned using the following field (see also `GET With Body`):

- `query`: object containing the query filters applied in the search request to filter the collection resource.

As Result, the standard response page using pagination links is defined as follows:

```
ResponsePage:
  type: object
  properties:
    self:
      description: Pagination link pointing to the current page.
      type: string
      format: uri
    first:
      description: Pagination link pointing to the first page.
      type: string
      format: uri
    prev:
      description: Pagination link pointing to the previous page.
```

```
      type: string
      format: uri
    next:
      description: Pagination link pointing to the next page.
      type: string
      format: uri
    last:
      description: Pagination link pointing to the last page.
      type: string
      format: uri

     query:
       description: >
         Object containing the query filters applied to the collection resource.
       type: object
       properties: ...

     items:
       description: Array of collection items.
       type: array
       required: false
       items:
         type: ...
```

The response page may contain additional metadata about the collection or the current page.

#### 3.10.2.3. Address fields

Address structures play a role in different functional and use-case contexts, including country variances. All attributes that relate to address information should follow the naming and semantics defined below.

```
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
```

Grouping and cardinality of fields in specific data types may vary based on the specific use case (e.g. combining addressee and address fields into a single type when modeling an address label vs distinct addressee and address types when modeling users and their addresses).

# 3.11. API naming

## 3.11.1. MUST/SHOULD use functional naming schema

Functional naming is a powerful, yet easy way to align global resources as *host*, *permission*, and *event names* within an the application landscape. It helps to preserve uniqueness of names while giving readers meaningful context information about the addressed component. Besides, the most important aspect is, that it allows to keep APIs stable in the case of technical and organizational changes (Pon for example maintains an internal naming convention).

To make use of this advantages for APIs with a larger audience we strongly recommended to follow the functional naming schema for hostnames, permission names, and event names in APIs as follows:

| Functional Naming | Audience |
|---|---|
| **must** | external-public, external-partner |
| **should** | company-internal, business-unit-internal |
| **may** | component-internal |

To conduct the functional naming schema, a unique `functional-name` is assigned to each functional component. It is built of the domain name of the functional group the component is belonging to and a unique a short identifier for the functional component itself:

```
<functional-name>       ::= <functional-domain>-<functional-component>
<functional-domain>     ::= [a-z][a-z0-9]*  -- managed functional group of components
<functional-component>  ::= [a-z][a-z0-9-]* -- name of owning functional component
```

**Internal Hint**: Use the simple functional name registry (internal link) to register your functional name before using it. The registry is a centralized infrastructure service to ensure uniqueness of your functional names (and available domains) and to support hostname DNS resolution.

Please see the following rules for detailed functional naming patterns:

- **MUST** follow naming convention for hostnames
- **MUST** follow naming convention for event type names

### 3.11.2. MUST follow naming convention for hostnames

Hostnames in APIs must, respectively should conform to the functional naming depending on the audience as follows (see **MUST/SHOULD use functional naming schema** for details and `<functional-name>` definition):

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

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

The following application specific legacy convention is **only** allowed for hostnames of component-internal APIs:

```
<application-hostname> ::= <application-id>.<organization-unit>.zalan.do
<application-id>       ::= [a-z][a-z0-9-]*  -- application identifier
<organization-id>      ::= [a-z][a-z0-9-]*  -- organization unit identifier, e.g. team
identifier
```

### 3.11.3. MUST use lowercase separate words with hyphens for path segments

Example:

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

This applies to concrete path segments and not the names of path parameters. For example `{shipment_order_id}` would be ok as a path parameter.

### 3.11.4. MUST use snake_case (never camelCase) for query parameters

Examples:

```
customer_number, order_id, billing_address
```

### 3.11.5. SHOULD prefer hyphenated-pascal-case for HTTP header fields

This is for consistency in your documentation (most other headers follow this convention). Avoid camelCase (without hyphens). Exceptions are common abbreviations like "ID."

Examples:

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

See also: HTTP Headers are case-insensitive (RFC 7230).

See Common headers and Proprietary headers sections for more guidance on HTTP headers.

### 3.11.6. MUST pluralize resource names

Usually, a collection of resource instances is provided (at least API should be ready here). The special case of a resource singleton is a collection with cardinality 1.

### 3.11.7. SHOULD not use /api as base path

In most cases, all resources provided by a service are part of the public API, and therefore should be made available under the root "/" base path.

If the service should also support non-public, internal APIs — for specific operational support functions, for example — we encourage you to maintain two different API specifications and provide API audience. For both APIs, you should not use `/api` as base path.

We see API's base path as a part of deployment variant configuration. Therefore, this information has to be declared in the server object.

### 3.11.8. MUST avoid trailing slashes

The trailing slash must not have specific semantics. Resource paths must deliver the same results whether they have the trailing slash or not.

### 3.11.9. MUST stick to conventional query parameters

If you provide query support for searching, sorting, filtering, and paginating, you must stick to the following naming conventions:

- `q`: default query parameter, e.g. used by browser tab completion; should have an entity specific alias, e.g. sku.
- `sort`: comma-separated list of fields (as defined by **MUST** define collection format of header and query parameters) to define the sort order. To indicate sorting direction, fields may be prefixed with `+` (ascending) or `-` (descending), e.g. /sales-orders?sort=+id.
- `fields`: field name expression to retrieve only a subset of fields of a resource. See **SHOULD** support partial responses via filtering below.
- `embed`: field name expression to expand or embedded sub-entities, e.g. inside of an article entity, expand silhouette code into the silhouette object. Implementing `embed` correctly is difficult, so do it with care. See **SHOULD** allow optional embedding of sub-resources below.
- `offset`: numeric offset of the first element provided on a page representing a collection request. See Pagination section below.
- `cursor`: an opaque pointer to a page, never to be inspected or constructed by clients. It usually (encrypted) encodes the page position, i.e. the identifier of the first or last page element, the pagination direction, and the applied query filters to recreate the collection. See pagination section below.

- `limit`: client suggested limit to restrict the number of entries on a page. See Pagination section below.

# 3.12. Resources

## 3.12.1. MUST avoid actions — think about resources

REST is all about your resources, so consider the domain entities that take part in web service interaction, and aim to model your API around these using the standard HTTP methods as operation indicators. For instance, if an application has to lock articles explicitly so that only one user may edit them, create an article lock with `PUT` or `POST` instead of using a lock action.

Request:

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

The added benefit is that you already have a service for browsing and filtering article locks.

## 3.12.2. SHOULD model complete business processes

An API should contain the complete business processes containing all resources representing the process. This enables clients to understand the business process, foster a consistent design of the business process, allow for synergies from description and implementation perspective, and eliminates implicit invisible dependencies between APIs.

In addition, it prevents services from being designed as thin wrappers around databases, which normally tends to shift business logic to the clients.

## 3.12.3. SHOULD define *useful* resources

As a rule of thumb 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 great way to support the last 10% is to allow clients to specify their needs for more/less information by supporting filtering and embedding.

## 3.12.4. MUST keep URLs verb-free

The API describes resources, so the only place where actions should appear is in the HTTP methods. In URLs, use only nouns. Instead of thinking of actions (verbs), it's often helpful to think about putting a message in a letter box: e.g., instead of having the verb *cancel* in the url, think of sending a message to cancel an order to the *cancellations* letter box on the server side.

## 3.12.5. MUST use domain-specific resource names

API resources represent elements of the application's domain model. Using domain-specific nomenclature for resource names helps developers to understand the functionality and basic semantics of your resources. It also reduces the need for further documentation outside the API

definition. For example, "sales-order-items" is superior to "order-items" in that it clearly indicates which business object it represents. Along these lines, "items" is too general.

### 3.12.6. MUST use URL-friendly resource identifiers

To simplify encoding of resource IDs in URLs, their representation must only consist of ASCII strings using letters, numbers, underscore, minus, colon, period, and - on rare occasions - slash. The corresponding regular expression is

[a-zA-Z0-9:._\-/]*

**Note:** slashes are only allowed to build and signal resource identifiers consisting of compound keys.

### 3.12.7. MUST identify resources and sub-resources via path segments

Some API resources may contain or reference sub-resources. Embedded sub-resources, which are not top-level resources, are parts of a higher-level resource and cannot be used outside of its scope. Sub-resources should be referenced by their name and identifier in the path segments as follows:

```
/resources/{resource-id}/sub-resources/{sub-resource-id}
```

In order to improve the consumer experience, you should aim for intuitively understandable URLs, where each sub-path is a valid reference to a resource or a set of resources. E.g. if `/customers/12ev123bv12v/addresses/DE_100100101` is valid, then `/customers/12ev123bv12v/addresses`, `/customers/12ev123bv12v` and `/customers` must be valid as well in principle. E.g.:

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

**Note:** resource identifiers may be build of multiple other resource identifiers (see **MAY** expose compound keys as resource identifiers).

### 3.12.8. MAY expose compound keys as resource identifiers

If a resource is best identified by a *compound key* consisting of multiple other resource identifiers, it is allowed to reuse the compound key in its natural form containing slashes instead of *technical* resource identifier in the resource path without violating the above rule **MUST** identify resources and sub-resources via path segments as follows:

```
/resources/{compound-key-1}[delim-1]...[delim-n-1]{compound-key-n}
```

Example paths:

```
/shopping-carts/{country}/{session-id}
/shopping-carts/{country}/{session-id}/items/{item-id}
/api-specifications/{docker-image-id}/apis/{path}/{file-name}
/api-specifications/{repository-name}/{artifact-name}:{tag}
/article-size-advices/{sku}/{sales-channel}
```

**Warning:** Exposing a compound key as described above limits ability to evolve the structure of the resource identifier as it is no longer opaque.

To compensate for this drawback, APIs must apply a compound key abstraction consistently in all requests and responses parameters and attributes allowing consumers to treat these as *technical resource identifier* replacement. The use of independent compound key components must be limited to search and creation requests, as follows:

```
# compound key components passed as independent search query parameters
GET /article-size-advices?skus=sku-1,sku-2&sales_channel_id=sid-1
=> { "items": [{ "id": "id-1", ...  },{ "id": "id-2", ...  }] }

# opaque technical resource identifier passed as path parameter
GET /article-size-advices/id-1
=> { "id": "id-1", "sku": "sku-1", "sales_channel_id": "sid-1", "size": ... }

# compound key components passed as mandatory request fields
POST /article-size-advices { "sku": "sku-1", "sales_channel_id": "sid-1", "size": ...
}
=> { "id": "id-1", "sku": "sku-1", "sales_channel_id": "sid-1", "size": ... }
```

Where `id-1` is representing the opaque provision of the compound key `sku-1/sid-1` as technical resource identifier.

**Remark:** A compound key component may itself be used as another resource identifier providing another resource endpoint, e.g `/article-size-advices/{sku}`.

### 3.12.9. MAY consider using (non-)nested URLs

If a sub-resource is only accessible via its parent resource and may not exist without parent resource, consider using a nested URL structure, for instance:

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

However, if the resource can be accessed directly via its unique id, then the API should expose it as a top-level resource. For example, customer has a collection for sales orders; however, sales orders have globally unique id and some services may choose to access the orders directly, for instance:

```
/customers/1637asikzec1
```

```
/sales-orders/5273gh3k525a
```

### 3.12.10. SHOULD only use UUIDs if necessary

Generating IDs can be a scaling problem in high frequency and near real time use cases. UUIDs solve this problem, as they can be generated without collisions in a distributed, non-coordinated way and without additional server round trips.

However, they also come with some disadvantages:

- pure technical key without meaning; not ready for naming or name scope conventions that might be helpful for pragmatic reasons, e.g. we learned to use names for product attributes, instead of UUIDs
- less usable, because…
- cannot be memorized and easily communicated by humans
- harder to use in debugging and logging analysis
- less convenient for consumer facing usage
- quite long: readable representation requires 36 characters and comes with higher memory and bandwidth consumption
- not ordered along their creation history and no indication of used id volume
- may be in conflict with additional backward compatibility support of legacy ids

UUIDs should be avoided when not needed for large scale id generation. Instead, for instance, server side support with id generation can be preferred (`POST` on id resource, followed by idempotent `PUT` on entity resource). Usage of UUIDs is especially discouraged as primary keys of master and configuration data, like brand-ids or attribute-ids which have low id volume but widespread steering functionality.

Please be aware that sequential, strictly monotonically increasing numeric identifiers may reveal critical, confidential business information, like order volume, to non-privileged clients.

In any case, we should always use string rather than number type for identifiers. This gives us more flexibility to evolve the identifier naming scheme. Accordingly, if used as identifiers, UUIDs should not be qualified using a format property.

Hint: Usually, random UUID is used - see UUID version 4 in RFC 4122. Though UUID version 1 also contains leading timestamps it is not reflected by its lexicographic sorting. This deficit is addressed by ULID (Universally Unique Lexicographically Sortable Identifier). You may favour ULID instead of UUID, for instance, for pagination use cases ordered along creation time.

### 3.12.11. SHOULD limit number of resource types

To keep maintenance and service evolution manageable, we should follow "functional segmentation" and "separation of concern" design principles and do not mix different business functionalities in same API definition. In practice this means that the number of resource types exposed via an API should be limited. In this context a resource type is defined as a set of highly

related resources such as a collection, its members and any direct sub-resources.

For example, the resources below would be counted as three resource types, one for customers, one for the addresses, and one for the customers' related addresses:

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

Note that:

- We consider `/customers/id/preferences` part of the `/customers` resource type because it has a one-to-one relation to the customer without an additional identifier.
- We consider `/customers` and `/customers/id/addresses` as separate resource types because `/customers/id/addresses/{addr}` also exists with an additional identifier for the address.
- We consider `/addresses` and `/customers/id/addresses` as separate resource types because there's no reliable way to be sure they are the same.

Given this definition, our experience is that well defined APIs involve no more than 4 to 8 resource types. There may be exceptions with more complex business domains that require more resources, but you should first check if you can split them into separate subdomains with distinct APIs.

Nevertheless one API should hold all necessary resources to model complete business processes helping clients to understand these flows.

### 3.12.12. SHOULD limit number of sub-resource levels

There are main resources (with root url paths) and sub-resources (or *nested* resources with non-root urls paths). Use sub-resources if their life cycle is (loosely) coupled to the main resource, i.e. the main resource works as collection resource of the subresource entities. You should use ⇐ 3 sub-resource (nesting) levels — more levels increase API complexity and url path length. (Remember, some popular web browsers do not support URLs of more than 2000 characters.)

## 3.13. Performance

### 3.13.1. SHOULD reduce bandwidth needs and improve responsiveness

APIs should support techniques for reducing bandwidth based on client needs. This holds for APIs that (might) have high payloads and/or are used in high-traffic scenarios like the public Internet and telecommunication networks. Typical examples are APIs used by mobile web app clients with (often) less bandwidth connectivity.

Common techniques include:

- compression of request and response bodies (see **SHOULD** use `gzip` compression)
- querying field filters to retrieve a subset of resource attributes (see **SHOULD** support partial responses via filtering below)
- `ETag` and `If-Match`/`If-None-Match` headers to avoid re-fetching of unchanged resources (see **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header)
- `Prefer` header with `return=minimal` or `respond-async` to anticipate reduced processing requirements of clients (see **MAY** consider to support `Prefer` header to handle processing preferences)
- Pagination for incremental access of larger collections of data items
- caching of master data items, i.e. resources that change rarely or not at all after creation (see **MUST** document cachable `GET`, `HEAD`, and `POST` endpoints).

Each of these items is described in greater detail below.

### 3.13.2. SHOULD use `gzip` compression

Compress the payload of your API's responses with gzip, unless there's a good reason not to — for example, you are serving so many requests that the time to compress becomes a bottleneck. This helps to transport data faster over the network (fewer bytes) and makes frontends respond faster.

Though gzip compression might be the default choice for server payload, the server should also support payload without compression and its client control via `Accept-Encoding` request header — see also RFC 7231 Section 5.3.4. The server should indicate used gzip compression via the `Content-Encoding` header.

### 3.13.3. SHOULD support partial responses via filtering

Depending on your use case and payload size, you can significantly reduce network bandwidth need by supporting filtering of returned entity fields. Here, the client can explicitly determine the subset of fields he wants to receive via the `fields` query parameter. (It is analogue to GraphQL `fields` and simple queries, and also applied, for instance, for Google Cloud API's partial responses.)

#### 3.13.3.1. Unfiltered

```
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"
  } ]
}
```

#### 3.13.3.2. Filtered

```
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"
  } ]
}
```

The `fields` query parameter determines the fields returned with the response payload object. For instance, `(name)` returns `users` root object with only the `name` field, and `(name,friends(name))` returns the `name` and the nested `friends` object with only its `name` field.

Open API doesn't support you in formally specifying different return object schemes depending on a parameter. When you define the field parameter, we recommend to provide the following description: `Endpoint supports filtering of return object fields as described in #157

The syntax of the query `fields` value is defined by the following BNF grammar.

```
<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>          ::= "!"
```

**Note:** Following the principle of least astonishment, you should not define the `fields` query parameter using a default value, as the result is counter-intuitive and very likely not anticipated by the consumer.

### 3.13.4. SHOULD allow optional embedding of sub-resources

Embedding related resources (also know as *Resource expansion*) is a great way to reduce the number of requests. In cases where clients know upfront that they need some related resources they can instruct the server to prefetch that data eagerly. Whether this is optimized on the server, e.g. a database join, or done in a generic way, e.g. an HTTP proxy that transparently embeds resources, is up to the implementation.

See **MUST** stick to conventional query parameters for naming, e.g. "embed" for steering of embedded resource expansion. Please use the BNF grammar, as already defined above for filtering, when it comes to an embedding query syntax.

Embedding a sub-resource can possibly look like this where an order resource has its order items as sub-resource (/order/{orderId}/items):

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

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

### 3.13.5. MUST document cachable `GET`, `HEAD`, and `POST` endpoints

Caching has to take many aspects into account, e.g. general cacheability of response information, our guideline to protect endpoints using SSL and OAuth authorization, resource update and invalidation rules, existence of multiple consumer instances. As a consequence, caching is in best case complex, e.g. with respect to consistency, in worst case inefficient.

As a consequence, client side as well as transparent web caching should be avoided, unless the service supports and requires it to protect itself, e.g. in case of a heavily used and therefore rate limited master data service, i.e. data items that rarely or not at all change after creation.

As default, API providers and consumers should always set the `Cache-Control` header set to `Cache-Control: no-store` and assume the same setting, if no `Cache-Control` header is provided.

**Note:** There is no need to document this default setting. However, please make sure that your framework is attaching this header value by default, or ensure this manually, e.g. using the best

practice of Spring Security as shown below. Any setup deviating from this default must be sufficiently documented.

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

If your service really requires to support caching, please observe the following rules:

- Document all cacheable `GET`, `HEAD`, and `POST` endpoints by declaring the support of `Cache-Control`, `Vary`, and `ETag` headers in response. **Note:** you must not define the `Expires` header to prevent redundant and ambiguous definition of cache lifetime. A sensible default documentation of these headers is given below.
- Take care to specify the ability to support caching by defining the right caching boundaries, i.e. time-to-live and cache constraints, by providing sensible values for `Cache-Control` and `Vary` in your service. We will explain best practices below.
- Provide efficient methods to warm up and update caches, e.g. as follows:
  - In general, you should support `ETag` Together With `If-Match`/ `If-None-Match` Header on all cacheable endpoints.
  - For larger data items support `HEAD` requests or more efficient `GET` requests with `If-None-Match` header to check for updates.
  - For small data sets provide full collection `GET` requests supporting `ETag`, as well as `HEAD` requests or `GET` requests with `If-None-Match` to check for updates.
  - For medium sized data sets provide full collection `GET` requests supporting `ETag` together with Pagination and `<entity-tag>` filtering `GET` requests for limiting the response to changes since the provided `<entity-tag>`. **Note:** this is not supported by generic client and proxy caches on HTTP layer.

**Hint:** For proper cache support, you must return 304 without content on a failed `HEAD` or `GET` request with `If-None-Match: <entity-tag>` instead of 412.

```
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"
```

**Hint:** For `ETag` source see **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header.

The default setting for `Cache-Control` should contain the `private` directive for endpoints with standard OAuth authorization, as well as the `must-revalidate` directive to ensure, that the client does not use stale cache entries. Last, the `max-age` directive should be set to a value between a few seconds (`max-age=60`) and a few hours (`max-age=86400`) depending on the change rate of your master data and your requirements to keep clients consistent.

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

The default setting for `Vary` is harder to determine correctly. It highly depends on the API endpoint, e.g. whether it supports compression, accepts different media types, or requires other request specific headers. To support correct caching you have to carefully choose the value. However, a good first default may be:

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

Anyhow, this is only relevant, if you encourage clients to install generic HTTP layer client and proxy caches.

**Note:** generic client and proxy caching on HTTP level is hard to configure. Therefore, we strongly recommend to attach the (possibly distributed) cache directly to the service (or gateway) layer of your application. This relieves from interpreting the `Vary` header and greatly simplifies interpreting the `Cache-Control` and `ETag` headers. Moreover, is highly efficient with respect to caching performance and overhead, and allows to support more advanced cache update and warm up patterns.

Anyhow, please carefully read RFC 7234 before adding any client or proxy cache.

# 3.14. Hypermedia

## 3.14.1. MUST use REST maturity level 2

We strive for a good implementation of REST Maturity Level 2 as it enables us to build resource-oriented APIs that make full use of HTTP verbs and status codes. You can see this expressed by many rules throughout these guidelines, e.g.:

- **MUST** avoid actions — think about resources
- **MUST** keep URLs verb-free
- **MUST** use HTTP methods correctly
- **MUST** use standard HTTP status codes

Although this is not HATEOAS, it should not prevent you from designing proper link relationships in your APIs as stated in rules below.

## 3.14.2. MAY use REST maturity level 3 - HATEOAS

We do not generally recommend to implement REST Maturity Level 3. HATEOAS comes with additional API complexity without real value in our SOA context where client and server interact via REST APIs and provide complex business functions as part of our e-commerce SaaS platform.

Our major concerns regarding the promised advantages of HATEOAS (see also RESTistential Crisis over Hypermedia APIs, Why I Hate HATEOAS and others for a detailed discussion):

- We follow the API First principle with APIs explicitly defined outside the code with standard specification language. HATEOAS does not really add value for SOA client engineers in terms of API self-descriptiveness: a client engineer finds necessary links and usage description (depending on resource state) in the API reference definition anyway.
- Generic HATEOAS clients which need no prior knowledge about APIs and explore API capabilities based on hypermedia information provided, is a theoretical concept that we haven't seen working in practice and does not fit to our SOA set-up. The Open API description format (and tooling based on Open API) doesn't provide sufficient support for HATEOAS either.
- In practice relevant HATEOAS approximations (e.g. following specifications like HAL or JSON API) support API navigation by abstracting from URL endpoint and HTTP method aspects via link types. So, Hypermedia does not prevent clients from required manual changes when domain model changes over time.
- Hypermedia make sense for humans, less for SOA machine clients. We would expect use cases where it may provide value more likely in the frontend and human facing service domain.
- Hypermedia does not prevent API clients to implement shortcuts and directly target resources without 'discovering' them.

However, we do not forbid HATEOAS; you could use it, if you checked its limitations and still see clear value for your usage scenario that justifies its additional complexity. If you use HATEOAS please share experience and present your findings.

### 3.14.3. MUST use full, absolute URI

Links to other resource must always use full, absolute URI.

**Motivation**: Exposing any form of relative URI (no matter if the relative URI uses an absolute or relative path) introduces avoidable client side complexity. It also requires clarity on the base URI, which might not be given when using features like embedding subresources. The primary advantage of non-absolute URI is reduction of the payload size, which is better achievable by following the recommendation to use gzip compression

### 3.14.4. MUST use common hypertext controls

When embedding links to other resources into representations you must use the common hypertext control object. It contains at least one attribute:

- `href`: The URI of the resource the hypertext control is linking to. All our API are using HTTP(s) as URI scheme.

In API that contain any hypertext controls, the attribute name `href` is reserved for usage within hypertext controls.

The schema for hypertext controls can be derived from this model:

```
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
```

The name of an attribute holding such a `HttpLink` object specifies the relation between the object that contains the link and the linked resource. Implementations should use names from the IANA Link Relation Registry whenever appropriate. As IANA link relation names use hyphen-case notation, while this guide enforces snake_case notation for attribute names, hyphens in IANA names have to be replaced with underscores (e.g. the IANA link relation type `version-history` would become the attribute `version_history`)

Specific link objects may extend the basic link type with additional attributes, to give additional information related to the linked resource or the relationship between the source resource and the linked one.

E.g. a service providing "Person" resources could model a person who is married with some other person with a hypertext control that contains attributes which describe the other person (`id`, `name`) but also the relationship "spouse" between the two persons (`since`):

```
{
  "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"
  }
}
```

Hypertext controls are allowed anywhere within a JSON model. While this specification would allow HAL, we actually don't recommend/enforce the usage of HAL anymore as the structural separation of meta-data and data creates more harm than value to the understandability and usability of an API.

### 3.14.5. SHOULD use simple hypertext controls for pagination and self-references

For pagination and self-references a simplified form of the extensible common hypertext controls should be used to reduce the specification and cognitive overhead. It consists of a simple URI value in combination with the corresponding link relations, e.g. `next`, `prev`, `first`, `last`, or `self`.

See simple-hypertext-control-fields and **SHOULD** use pagination links where applicable for examples and more information.

### 3.14.6. MUST not use link headers with JSON entities

For flexibility and precision, we prefer links to be directly embedded in the JSON payload instead of being attached using the uncommon link header syntax. As a result, the use of the `Link` Header defined by RFC 8288 in conjunction with JSON media types is forbidden.

## 3.15. Common headers

This section describes a handful of headers, which we found raised the most questions in our daily usage, or which are useful in particular circumstances but not widely known.

### 3.15.1. MUST use `Content-*` headers correctly

Content or entity headers are headers with a `Content-` prefix. They describe the content of the body of the message and they can be used in both, HTTP requests and responses. Commonly used content headers include but are not limited to:

- `Content-Disposition` can indicate that the representation is supposed to be saved as a file, and the proposed file name.
- `Content-Encoding` indicates compression or encryption algorithms applied to the content.
- `Content-Length` indicates the length of the content (in bytes).

- `Content-Language` indicates that the body is meant for people literate in some human language(s).
- `Content-Location` indicates where the body can be found otherwise (**MAY** use `Content-Location` header for more details]).
- `Content-Range` is used in responses to range requests to indicate which part of the requested resource representation is delivered with the body.
- `Content-Type` indicates the media type of the body content.

### 3.15.2. MAY use standardized headers

Use this list and mention its support in your Open API definition.

### 3.15.3. MAY use `Content-Location` header

The `Content-Location` header is *optional* and can be used in successful write operations (`PUT`, `POST`, or `PATCH`) or read operations (`GET`, `HEAD`) to guide caching and signal a receiver the actual location of the resource transmitted in the response body. This allows clients to identify the resource and to update their local copy when receiving a response with this header.

The Content-Location header can be used to support the following use cases:

- For reading operations `GET` and `HEAD`, a different location than the requested URI can be used to indicate that the returned resource is subject to content negotiations, and that the value provides a more specific identifier of the resource.
- For writing operations `PUT` and `PATCH`, an identical location to the requested URI can be used to explicitly indicate that the returned resource is the current representation of the newly created or updated resource.
- For writing operations `POST` and `DELETE`, a content location can be used to indicate that the body contains a status report resource in response to the requested action, which is available at provided location.

**Note**: When using the `Content-Location` header, the `Content-Type` header has to be set as well. For example:

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

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

### 3.15.4. SHOULD use `Location` header instead of `Content-Location` header

As the correct usage of `Content-Location` with respect to semantics and caching is difficult, we *discourage* the use of `Content-Location`. In most cases it is sufficient to direct clients to the resource location by using the `Location` header instead without hitting the `Content-Location` specific ambiguities and complexities.

More details in RFC 7231 7.1.2 Location, 3.1.4.2 Content-Location

### 3.15.5. MAY consider to support `Prefer` header to handle processing preferences

The `Prefer` header defined in RFC 7240 allows clients to request processing behaviors from servers. It pre-defines a number of preferences and is extensible, to allow others to be defined. Support for the `Prefer` header is entirely optional and at the discretion of API designers, but as an existing Internet Standard, is recommended over defining proprietary "X-" headers for processing directives.

The `Prefer` header can defined like this in an API definition:

```
components:
  headers:
  - Prefer:
      description: >
        The RFC7240 Prefer header indicates that a particular server behavior
        is preferred by the client but is not required for successful completion
        of the request (see [RFC 7240](https://tools.ietf.org/html/rfc7240).
        The following behaviors are supported by this API:

        # (indicate the preferences supported by the API or API endpoint)
        * **respond-async** is used to suggest the server to respond as fast as
          possible asynchronously using 202 - accepted - instead of waiting for
          the result.
        * **return=<minimal|representation>** is used to suggest the server to
          return using 204 without resource (minimal) or using 200 or 201 with
          resource (representation) in the response body on success.
        * **wait=<delta-seconds>** is used to suggest a maximum time the server
          has time to process the request synchronously.
        * **handling=<strict|lenient>** is used to suggest the server to be
          strict and report error conditions or lenient, i.e. robust and try to
          continue, if possible.

      type: array
      items:
        type: string
      required: false
```

**Note:** Please copy only the behaviors into your `Prefer` header specification that are supported by your API endpoint. If necessary, specify different `Prefer` headers for each supported use case.

Supporting APIs may return the `Preference-Applied` header also defined in RFC 7240 to indicate whether a preference has been applied.

### 3.15.6. MAY consider to support ETag together with If-Match/If-None-Match header

When creating or updating resources it may be necessary to expose conflicts and to prevent the 'lost update' or 'initially created' problem. Following RFC 7232 "HTTP: Conditional Requests" this can be best accomplished by supporting the ETag header together with the If-Match or If-None-Match conditional header. The contents of an ETag: <entity-tag> header is either (a) a hash of the response body, (b) a hash of the last modified field of the entity, or (c) a version number or identifier of the entity version.

To expose conflicts between concurrent update operations via PUT, POST, or PATCH, the If-Match: <entity-tag> header can be used to force the server to check whether the version of the updated entity is conforming to the requested <entity-tag>. If no matching entity is found, the operation is supposed a to respond with status code 412 - precondition failed.

Beside other use cases, If-None-Match: * can be used in a similar way to expose conflicts in resource creation. If any matching entity is found, the operation is supposed a to respond with status code 412 - precondition failed.

The ETag, If-Match, and If-None-Match headers can be defined as follows in the API definition:

```
components:
  headers:
  - ETag:
      description: |
        The RFC 7232 ETag header field in a response provides the entity-tag of
        a selected resource. The entity-tag is an opaque identifier for versions
        and representations of the same 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 (see
        [RFC 7232 Section 2.3](https://tools.ietf.org/html/rfc7232#section-2.3).

      type: string
      required: false
      example: W/"xy", "5", "5db68c06-1a68-11e9-8341-68f728c1ba70"

  - 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 (see [RFC 7232 Section
        3.1](https://tools.ietf.org/html/rfc7232#section-3.1).

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

  - 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 (see [RFC 7232 Section
        3.2](https://tools.ietf.org/html/rfc7232#section-3.2).

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

Please see Optimistic locking in RESTful APIs for a detailed discussion and options.

### 3.15.7. MAY consider to support `Idempotency-Key` header

When creating or updating resources it can be helpful or necessary to ensure a strong idempotent behavior comprising same responses, to prevent duplicate execution in case of retries after timeout and network outages. Generally, this can be achieved by sending a client specific *unique request key* – that is not part of the resource – via `Idempotency-Key` header.

The *unique request key* is stored temporarily, e.g. for 24 hours, together with the response and the request hash (optionally) of the first request in a key cache, regardless of whether it succeeded or failed. The service can now look up the *unique request key* in the key cache and serve the response from the key cache, instead of re-executing the request, to ensure idempotent behavior. Optionally, it can check the request hash for consistency before serving the response. If the key is not in the key store, the request is executed as usual and the response is stored in the key cache.

This allows clients to safely retry requests after timeouts, network outages, etc. while receive the same response multiple times. **Note:** The request retry in this context requires to send the exact same request, i.e. updates of the request that would change the result are off-limits. The request hash in the key cache can protection against this misbehavior. The service is recommended to reject such a request using status code 400.

**Important:** To grant a reliable idempotent execution semantic, the resource and the key cache have to be updated with hard transaction semantics – considering all potential pitfalls of failures, timeouts, and concurrent requests in a distributed systems. This makes a correct implementation exceeding the local context very hard.

The `Idempotency-Key` header must be defined as follows, but you are free to choose your expiration time:

```
components:
  headers:
  - Idempotency-Key:
      description: |
        The idempotency key is a free identifier created by the client to
        identify a request. It is used by the service to identify subsequent
        retries of the same request and ensure idempotent behavior by sending
        the same response without executing the request a second time.
```

```
      Clients should be careful as any subsequent requests with the same key
      may return the same response without further check. Therefore, it is
      recommended to use an UUID version 4 (random) or any other random
      string with enough entropy to avoid collisions.

      Idempotency keys expire after 24 hours. Clients are responsible to stay
      within this limits, if they require idempotent behavior.

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

**Hint:** The key cache is not intended as request log, and therefore should have a limited lifetime, else it could easily exceed the data resource in size.

**Note:** The `Idempotency-Key` header unlike other headers in this section is not standardized in an RFC. Our only reference are the usage in the Stripe API. However, as it fit not into our section about Proprietary headers, and we did not want to change the header name and semantic, we decided to treat it as any other common header.

# 3.16. Proprietary headers

This section shares definitions of proprietary headers that should be named consistently because they address overarching service-related concerns. Whether services support these concerns or not is optional; therefore, the Open API API specification is the right place to make this explicitly visible. Use the parameter definitions of the resource HTTP methods.

## 3.16.1. MUST use only the specified proprietary Pon headers

As a general rule, proprietary HTTP headers should be avoided. Still they can be useful in cases where context needs to be passed through multiple services in an end-to-end fashion. As such, a valid use-case for a proprietary header is providing context information, which is not a part of the actual API, but is needed by subsequent communication.

From a conceptual point of view, the semantics and intent of an operation should always be expressed by URLs path and query parameters, the method, and the content. Headers are more often used to implement functions close to the protocol considerations, such as flow control, content negotiation, and authentication. Thus, headers are reserved for general context information (RFC 7231).

`X-` headers were initially reserved for unstandardized parameters, but the usage of `X-` headers is deprecated (RFC 6648). This complicates the contract definition between consumer and producer of an API following these guidelines, since there is no aligned way of using those headers. Because of this, the guidelines restrict which `X-` headers can be used and how they are used.

The Internet Engineering Task Force's states in RFC 6648 that company specific header' names should incorporate the organization's name. We aim for backward compatibility, and therefore

keep the X- prefix.

The following proprietary headers have been specified by this guideline for usage so far. Remember that HTTP header field names are not case-sensitive.

| Header field name | Type | Description | Header field value example |
|---|---|---|---|
| X-Flow-ID | String | For more information see **MUST** support X-Flow-ID. | GKY7oDhpSiKY_gAAAABZ_A |
| X-Tenant-ID | String | Identifies the tenant initiated the request to the multi tenant Pon Platform. The X-Tenant-ID must be set according to the Business Partner ID extracted from the OAuth token when a request from a Business Partner hits the Pon Platform. | 9f8b3ca3-4be5-436c-a847-9cd55460c495 |
| X-Sales-Channel | String | Sales channels are owned by retailers and represent a specific consumer segment being addressed with a specific product assortment that is offered via CFA retailer catalogs to consumers (see platform glossary (internal link)) | 52b96501-0f8d-43e7-82aa-8a96fab134d7 |
| X-Frontend-Type | String | Consumer facing applications (CFAs) provide business experience to their customers via different frontend application types, for instance, mobile app or browser. Info should be passed-through as generic aspect — there are diverse concerns, e.g. pushing mobiles with specific coupons, that make use of it. Current range is mobile-app, browser, facebook-app, chat-app | mobile-app |
| X-device-Type | String | There are also use cases for steering customer experience (incl. features and content) depending on device type. Via this header info should be passed-through as generic aspect. Current range is smartphone, tablet, desktop, other. | tablet |
| X-device-OS | String | On top of device type above, we even want to differ between device platform, e.g. smartphone Android vs. iOS. Via this header info should be passed-through as generic aspect. Current range is iOS, Android, Windows, Linux, MacOS. | Android |

| Header field name | Type | Description | Header field value example |
|---|---|---|---|
| `X-Mobile-Advertising-ID` | String | It is either the IDFA (Apple Identifier for mobile Advertising) for iOS, or the GAID (Google mobile Advertising Identifier) for Android. It is a unique, customer-resettable identifier provided by mobile device's operating system to faciliate personalized advertising, and usually passed by mobile apps via http header when calling backend services. Called services should be ready to pass this parameter through when calling other services. It is not sent if the customer disables it in the settings for respective mobile platform. | b89fadce-1f42-46aa-9c83-b7bc49e76e1f |

**Exception:** The only exception to this guideline are the conventional hop-by-hop `X-RateLimit-` headers which can be used as defined in **MUST** use code 429 with headers for rate limits.

### 3.16.2. MUST propagate proprietary headers

All Pon's proprietary headers are end-to-end headers. [2]

All headers specified above must be propagated to the services down the call chain. The header names and values must remain unchanged.

For example, the values of the custom headers like `X-Device-Type` can affect the results of queries by using device type information to influence recommendation results. Besides, the values of the custom headers can influence the results of the queries (e.g. the device type information influences the recommendation results).

Sometimes the value of a proprietary header will be used as part of the entity in a subsequent request. In such cases, the proprietary headers must still be propagated as headers with the subsequent request, despite the duplication of information.

### 3.16.3. MUST support `X-Flow-ID`

The *Flow-ID* is a generic parameter to be passed through service APIs and events and written into log files and traces. A consequent usage of the *Flow-ID* facilitates the tracking of call flows through our system and allows the correlation of service activities initiated by a specific call. This is extremely helpful for operational troubleshooting and log analysis. Main use case of *Flow-ID* is to track service calls of our SaaS fashion commerce platform and initiated internal processing flows (executed synchronously via APIs or asynchronously via published events).

#### 3.16.3.1. Data Definition

The *Flow-ID* must be passed through:

- RESTful API requests via `X-Flow-ID` proprietary header (see **MUST** propagate proprietary headers)

- Published events via `flow_id` event field (see metadata)

The following formats are allowed:

- `UUID` (RFC-4122)
- `base64` (RFC-4648)
- `base64url` (RFC-4648 Section 5)
- Random unique string restricted to the character set `[a-zA-Z0-9/+_-=]` maximal of 128 characters.

**Note:** If a legacy subsystem can only process *Flow-IDs* with a specific format or length, it must define this restrictions in its API specification, and be generous and remove invalid characters or cut the length to the supported limit.

**Hint:** In case distributed tracing is supported by OpenTracing (internal link) you should ensure that created *spans* are tagged using `flow_id` — see How to Connect Log Output with OpenTracing Using Flow-IDs (internal link) or Best practises (internal link).

#### 3.16.3.2. Service Guidance

- Services **must** support *Flow-ID* as generic input, i.e.
  - RESTful API endpoints **must** support `X-Flow-ID` header in requests
  - Event listeners **must** support the metadata `flow-id` from events.

  **Note:** API-Clients **must** provide *Flow-ID* when calling a service or producing events. If no *Flow-ID* is provided in a request or event, the service must create a new *Flow-ID*.
- Services **must** propagate *Flow-ID*, i.e. use *Flow-ID* received with API-Calls or consumed events as...
  - input for all API called and events published during processing
  - data field written for logging and tracing

**Hint:** This rule also applies to application internal interfaces and events not published via Nakadi (but e.g. via AWS SQS, Kinesis or service specific DB solutions).

# 3.17. API Operation

## 3.17.1. MUST publish Open API specification

All service applications must publish Open API specifications of their external APIs. While this is optional for internal APIs, i.e. APIs marked with the **component-internal** API audience group, we still recommend to do so to profit from the API management infrastructure.

An API is published by copying its **Open API specification** into the reserved **/pon-apis** directory of the **deployment artifact** used to deploy the provisioning service. The directory must only contain **self-contained YAML files** that each describe one API (exception see **(RFP) MUST** only use durable and immutable remote references). We prefer this deployment artifact-based method over the past

(now legacy) `.well-known/schema-discovery` service endpoint-based publishing process, that we only support for backward compatibility reasons.

Background: In our dynamic and complex service infrastructure, it is important to provide API client developers a central place with online access to the API specifications of all running applications. As a part of the infrastructure, the API publishing process is used to detect API specifications. The findings are published in the API Portal - the universal hub for all Pon APIs.

**Note:** To publish an API, it is still necessary to deploy the artifact successful, as we focus the discovery experience on APIs supported by running services.

### 3.17.2. SHOULD monitor API usage

Owners of APIs used in production should monitor API service to get information about its using clients. This information, for instance, is useful to identify potential review partner for API changes.

Hint: A preferred way of client detection implementation is by logging of the client-id retrieved from the OAuth token.

## 3.18. Events

Pon's architecture centers around decoupled microservices and in that context we favour asynchronous event driven approaches. The guidelines in this section focus on how to design and publish events intended to be shared for others to consume.

### 3.18.1. Events, event types, and categories

Events are defined using an item called an *Event Type*. The Event Type allows events to have their structure declared with a schema by producers and understood by consumers. An Event Type declares standard information, such as a name, an owning application (and by implication, an owning team), a schema defining the event's custom data, and a compatibility mode declaring how the schema will be evolved. Event Types also allow the declaration of validation and enrichment strategies for events, along with supplemental information such as how events can be partitioned in an event stream.

Event Types belong to a well known *Event Category* (such as a data change category), which provides extra information that is common to that kind of event.

Event Types can be published and made available as API resources for teams to use, typically in an *Event Type Registry*. Each event published can then be validated against the overall structure of its event type and the schema for its custom data.

The basic model described above was originally developed in the Nakadi project, which acts as a reference implementation of the event type registry, and as a validating publish/subscribe broker for event producers and consumers.

### 3.18.2. MUST treat events as part of the service interface

Events are part of a service's interface to the outside world equivalent in standing to a service's

REST API. Services publishing data for integration must treat their events as a first class design concern, just as they would an API. For example this means approaching events with the "API first" principle in mind as described in the Introduction.

### 3.18.3. MUST make event schema available for review

Services publishing event data for use by others must make the event schema as well as the event type definition available for review.

### 3.18.4. MUST ensure event schema conforms to Open API schema object

To align the event schema specifications to API specifications, we use the Schema Object as defined by the Open API Specifications to define event schemas. This is particularly useful for events that represent data changes about resources also used in other APIs.

The Open API Schema Object is an **extended subset** of JSON Schema Draft 4. For convenience, we highlight some important differences below. Please refer to the Open API Schema Object specification for details.

As the Open API Schema Object specification *removes* some JSON Schema keywords, the following properties **must not** be used in event schemas:

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

On the other side Schema Object *redefines* some JSON Schema keywords:

- `additionalProperties`: For event types that declare compatibility guarantees, there are recommended constraints around the use of this field. See the guideline **SHOULD** avoid `additionalProperties` in event type definitions for details.

Finally, the Schema Object *extends* JSON Schema with some keywords:

- `readOnly`: events are logically immutable, so `readOnly` can be considered redundant, but harmless.
- `discriminator`: to support polymorphism, as an alternative to `oneOf`.
- `^x-`: patterned objects in the form of vendor extensions can be used in event type schema, but it might be the case that general purpose validators do not understand them to enforce a validation check, and fall back to must-ignore processing. A future version of the guidelines may define well known vendor extensions for events.

### 3.18.5. MUST ensure that events are registered as event types

In Pon's architecture, events are registered using a structure called an *Event Type*. The Event Type declares standard information as follows:

- A well known event category, such as a general or data change category.
- The name of the event type.
- The definition of the event target audience.
- An owning application, and by implication, an owning team.
- A schema defining the event payload.
- The compatibility mode for the type.

Event Types allow easier discovery of event information and ensure that information is well-structured, consistent, and can be validated.

Event type owners must pay attention to the choice of compatibility mode. The mode provides a means to evolve the schema. The range of modes are designed to be flexible enough so that producers can evolve schemas while not inadvertently breaking existing consumers:

- `none`: Any schema modification is accepted, even if it might break existing producers or consumers. When validating events, undefined properties are accepted unless declared in the schema.
- `forward`: A schema `S1` is forward compatible if the previously registered schema, `S0` can read events defined by `S1` - that is, consumers can read events tagged with the latest schema version using the previous version as long as consumers follow the robustness principle described in the guideline's API design principles.
- `compatible`: This means changes are fully compatible. A new schema, `S1`, is fully compatible when every event published since the first schema version will validate against the latest schema. In compatible mode, only the addition of new optional properties and definitions to an existing schema is allowed. Other changes are forbidden.

The compatibility mode interact with revision numbers in the schema `version` field, which follows semantic versioning (MAJOR.MINOR.PATCH):

- Changing an event type with compatibility mode `compatible` can lead to a PATCH or MINOR version revision. MAJOR breaking changes are not allowed.
- Changing an event type with compatibility mode `forward` can lead to a PATCH or MINOR version revision. MAJOR breaking changes are not allowed.
- Changing an event type with compatibility mode `none` can lead to PATCH, MINOR or MAJOR level changes.

The following examples illustrate this relations:

- Changes to the event type's `title` or `description` are considered PATCH level.
- Adding new optional fields to an event type's schema is considered a MINOR level change.

- All other changes are considered MAJOR level, such as renaming or removing fields, or adding new required fields.

The core Event Type structure is shown below as an Open API object definition:

```
EventType:
  description: |
    An event type defines the schema and its runtime properties. The required
    fields are the minimum set the creator of an event type is expected to
    supply.
  required:
    - name
    - category
    - owning_application
    - schema
  properties:
    name:
      description: |
        Name of this EventType. The name must follow the functional naming
        pattern `<functional-name>.<event-name>` to preserve global
        uniqueness and readability.
      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: |
        Intended target audience of the event type, analogue to audience definition
for REST APIs
        in rule #219 https://github.com/PonDigitalSolutions/restful-api-guidelines
    owning_application:
      description: |
        Name of the application (eg, as would be used in infrastructure
        application or service registry) owning this `EventType`.
      type: string
      example: price-service
    category:
      description: Defines the category of this EventType.
      type: string
      x-extensible-enum:
        - data
        - general
```

```
    compatibility_mode:
      description: |
        The compatibility mode to evolve the schema.
      type: string
      x-extensible-enum:
        - compatible
        - forward
        - none
      default: forward
    schema:
      description: The most recent payload schema for this EventType.
      type: object
      properties:
        version:
          description: Values are based on semantic versioning (eg "1.2.1").
          type: string
          default: '1.0.0'
        created_at:
          description: Creation timestamp of the schema.
          type: string
          readOnly: true
          format: date-time
          example: '1996-12-19T16:39:57-08:00'
        type:
          description: |
             The schema language of schema definition. Currently only
             json_schema (JSON Schema v04) syntax is defined, but in the
             future there could be others.
          type: string
          x-extensible-enum:
            - json_schema
        schema:
          description: |
              The schema as string in the syntax defined in the field type.
          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: When this event type was created.
      type: string
      pattern: date-time
    updated_at:
      description: When this event type was last updated.
      type: string
      pattern: date-time
```

APIs such as registries supporting event types, may extend the model, including the set of supported categories and schema formats. For example the Nakadi API's event category registration also allows the declaration of validation and enrichment strategies for events, along with supplemental information, such as how events are partitioned in the stream (see **SHOULD** use the hash partition strategy for data change events).

### 3.18.6. MUST ensure that events conform to a well-known event category

An *event category* describes a generic class of event types. The guidelines define two such categories:

- General Event: a general purpose category.
- Data Change Event: a category used for describing changes to data entities used for data replication based data integration.

The set of categories is expected to evolve in the future.

A category describes a predefined structure that event publishers must conform to along with standard information about that kind of event (such as the operation for a data change event).

#### 3.18.6.1. The general event category

The structure of the *General Event Category* is shown below as an Open API Schema Object definition:

```
GeneralEvent:
  description: |
    A general kind of event. Event kinds based on this event define their
    custom schema payload as the top level of the document, with the
    "metadata" field being required and reserved for standard metadata. An
    instance of an event based on the event type thus conforms to both the
    EventMetadata definition and the custom schema definition. Previously
    this category was called the Business Category.
  required:
    - metadata
  properties:
    metadata:
        $ref: '#/definitions/EventMetadata'
```

Event types based on the General Event Category define their custom schema payload at the top-level of the document, with the metadata field being reserved for standard information (the contents of metadata are described further down in this section).

In the example fragment below, the reserved metadata field is shown with fields "a" and "b" being defined as part of the custom schema:

Note:

- The General Event in a previous version of the guidelines was called a *Business Event*. Implementation experience has shown that the category's structure gets used for other kinds of events, hence the name has been generalized to reflect how teams are using it.
- The General Event is still useful and recommended for the purpose of defining events that drive a business process.
- The Nakadi broker still refers to the General Category as the Business Category and uses the keyword "business" for event type registration. Other than that, the JSON structures are identical.

See **MUST** use the general event category to signal steps and arrival points in business processes for more guidance on how to use the category.

#### 3.18.6.2. The data change event category

The *Data Change Event Category* structure is shown below as an Open API Schema Object:

```
DataChangeEvent:
  description: |
    Represents a change to an entity. The required fields are those
    expected to be sent by the producer, other fields may be added
    by intermediaries such as a publish/subscribe broker. An instance
    of an event based on the event type conforms to both the
    DataChangeEvent's definition and the custom schema definition.
  required:
    - metadata
    - data_op
    - data_type
    - data
  properties:
    metadata:
      description: The metadata for this event.
      $ref: '#/definitions/EventMetadata'
    data:
      description: |
        Contains custom payload for the event type. The payload must conform
        to a schema associated with the event type declared in the metadata
        object's `event_type` field.
      type: object
    data_type:
      description: name of the (business) data entity that has been mutated
```

```
      type: string
      example: 'sales_order.order'
    data_op:
      type: string
      enum: ['C', 'U', 'D', 'S']
      description: |
        The type of operation executed on the entity:

        - C: Creation of an entity
        - U: An update to an entity.
        - D: Deletion of an entity.
        - S: A snapshot of an entity at a point in time.
```

The Data Change Event Category is structurally different to the General Event Category. It defines a field called `data` for placing the custom payload information, as well as specific information related to data changes in the `data_type`. In the example fragment below, the fields `a` and `b` are part of the custom payload housed inside the `data` field:

See the following guidelines for more guidance on how to use the Data Change Event Category:

- **SHOULD** ensure that data change events match the APIs resources
- **MUST** use data change events to signal mutations
- **SHOULD** use the hash partition strategy for data change events

#### 3.18.6.3. Event metadata

The General and Data Change event categories share a common structure for *metadata*. The metadata structure is shown below as an Open API Schema Object:

```
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: Identifier of this event.
      type: string
      format: uuid
      example: '105a76d8-db49-4144-ace7-e683e8f4ba46'
    event_type:
      description: The name of the EventType of this Event.
      type: string
      example: 'example.important-business-event'
    occurred_at:
```

```
      description: When the event was created according to the producer.
      type: string
      format: date-time
      example: '1996-12-19T16:39:57-08:00'
    received_at:
      description: |
        When the event was seen by an intermediary such as a broker.
      type: string
      readOnly: true
      format: date-time
      example: '1996-12-19T16:39:57-08:00'
    version:
      description: |
        Version of the schema used for validating this event. This may be
        enriched upon reception by intermediaries. This string uses semantic
        versioning.
      type: string
      readOnly: true
    parent_eids:
      description: |
        Event identifiers of the Event that caused the generation of
        this Event. Set by the producer.
      type: array
      items:
        type: string
        format: uuid
      example: '105a76d8-db49-4144-ace7-e683e8f4ba46'
    flow_id:
      description: |
        A flow-id for this event (corresponds to the X-Flow-Id HTTP header).
      type: string
      example: 'JAh6xH4OQhCJ9PutIV_RYw'
    partition:
      description: |
        Indicates the partition assigned to this Event. Used for systems
        where an event type's events can be sub-divided into partitions.
      type: string
      example: '0'
```

Please note than intermediaries acting between the producer of an event and its ultimate consumers, may perform operations like validation of events and enrichment of an event's metadata. For example brokers such as Nakadi, can validate and enrich events with arbitrary additional fields that are not specified here and may set default or other values, if some of the specified fields are not supplied. How such systems work is outside the scope of these guidelines but producers and consumers working with such systems should look into their documentation for additional information.

### 3.18.7. MUST ensure that events define useful business resources

Events are intended to be used by other services including business process/data analytics and

monitoring. They should be based around the resources and business processes you have defined for your service domain and adhere to its natural lifecycle (see also **SHOULD** model complete business processes and **SHOULD** define *useful* resources).

As there is a cost in creating an explosion of event types and topics, prefer to define event types that are abstract/generic enough to be valuable for multiple use cases, and avoid publishing event types without a clear need.

### 3.18.8. MUST ensure that events do not provide sensitive data

Similar to API permission scopes, there will be event type permissions passed via an OAuth token supported in near future. In the meantime, teams are asked to note the following:

- Sensitive data, such as (e-mail addresses, phone numbers, etc) are subject to strict access and data protection controls.
- Event type owners **must not** publish sensitive information unless it's mandatory or necessary to do so. For example, events sometimes need to provide personal data, such as delivery addresses in shipment orders (as do other APIs), and this is fine.

### 3.18.9. MUST use the general event category to signal steps and arrival points in business processes

When publishing events that represent steps in a business process, event types must be based on the General Event category.

All your events of a single business process will conform to the following rules:

- Business events must contain a specific identifier field (a business process id or "bp-id") similar to flow-id to allow for efficient aggregation of all events in a business process execution.
- Business events must contain a means to correctly order events in a business process execution. In distributed settings where monotonically increasing values (such as a high precision timestamp that is assured to move forwards) cannot be obtained, the `parent_eids` data structure allows causal relationships to be declared between events.
- Business events should only contain information that is new to the business process execution at the specific step/arrival point.
- Each business process sequence should be started by a business event containing all relevant context information.
- Business events must be published reliably by the service.

At the moment we cannot state whether it's best practice to publish all the events for a business process using a single event type and represent the specific steps with a state field, or whether to use multiple event types to represent each step. For now we suggest assessing each option and sticking to one for a given business process.

### 3.18.10. MUST use data change events to signal mutations

When publishing events that represents created, updated, or deleted data, change event types must

be based on the Data Change Event category.

- Change events must identify the changed entity to allow aggregation of all related events for the entity.
- Change events **SHOULD** provide means for explicit event ordering.
- Change events must be published reliably by the service.

### 3.18.11. SHOULD provide means for explicit event ordering

Some common error cases may require event consumers to reconstruct event streams or replay events from a position within the stream. Events *should* therefore contain a way to restore their partial order of occurrence.

This can be done – among other ways – by adding

- a strictly monotonically increasing entity version (e.g. as created by a database) to allow for partial ordering of all events for an entity, or
- a strictly monotonically increasing message counter.

In the event type definition, the `ordering_key_fields` property should be used to indicate which field(s) contains the ordering key, if any.

System timestamps are not necessarily a good choice, since exact synchronization of clocks in distributed systems is difficult, two events may occur in the same microsecond and system clocks may jump backward or forward to compensate drifts or leap-seconds. If you use system timestamps to indicate event ordering, you must carefully ensure that your designated event order is not messed up by these effects.

Also, if using timestamps, the producer **must** make sure that they are formatted for all events in the UTC time zone, to allow for a simple string-based comparison.

**Note** that basing events on data structures that can be converged upon in a distributed setting (such as CRDTs, logical clocks and vector clocks) is outside the scope of this guidance.

### 3.18.12. SHOULD use the hash partition strategy for data change events

The `hash` partition strategy allows a producer to define which fields in an event are used as input to compute a logical partition the event should be added to. Partitions are useful as they allow supporting systems to scale their throughput while provide local ordering for event entities.

The `hash` option is particulary useful for data changes as it allows all related events for an entity to be consistently assigned to a partition, providing a relative ordered stream of events for that entity. This is because while each partition has a total ordering, ordering across partitions is not assured by a supporting system, thus it is possible for events sent across partitions to appear in a different order to consumers that the order they arrived at the server.

When using the `hash` strategy the partition key in almost all cases should represent the entity being changed and not a per event or change identifier such as the `eid` field or a timestamp. This ensures data changes arrive at the same partition for a given entity and can be consumed effectively by

clients.

There may be exceptional cases where data change events could have their partition strategy set to be the producer defined or random options, but generally `hash` is the right option - that is while the guidelines here are a "should", they can be read as "must, unless you have a very good reason".

### 3.18.13. SHOULD ensure that data change events match the APIs resources

A data change event's representation of an entity should correspond to the REST API representation.

There's value in having the fewest number of published structures for a service. Consumers of the service will be working with fewer representations, and the service owners will have less API surface to maintain. In particular, you should only publish events that are interesting in the domain and abstract away from implementation or local details - there's no need to reflect every change that happens within your system.

There are cases where it could make sense to define data change events that don't directly correspond to your API resource representations. Some examples are -

- Where the API resource representations are very different from the datastore representation, but the physical data are easier to reliably process for data integration.
- Publishing aggregated data. For example a data change to an individual entity might cause an event to be published that contains a coarser representation than that defined for an API
- Events that are the result of a computation, such as a matching algorithm, or the generation of enriched data, and which might not be stored as entity by the service.

### 3.18.14. MUST indicate ownership of event types

Event definitions must have clear ownership - this can be indicated via the `owning_application` field of the EventType.

Typically there is one producer application, which owns the EventType and is responsible for its definition, akin to how RESTful API definitions are managed. However, the owner may also be a particular service from a set of multiple services that are producing the same kind of event.

### 3.18.15. MUST define event payloads compliant with overall API guidelines

Events must be consistent with other API data and the API Guidelines in general.

Everything expressed in the Introduction to these Guidelines is applicable to event data interchange between services. This is because our events, just like our APIs, represent a commitment to express what our systems do and designing high-quality, useful events allows us to develop new and interesting products and services.

What distinguishes events from other kinds of data is the delivery style used, asynchronous publish-subscribe messaging. But there is no reason why they could not be made available using a REST API, for example via a search request or as a paginated feed, and it will be common to base events on the models created for the service's REST API.

The following existing guideline sections are applicable to events:

- General guidelines
- API naming
- Data formats
- Common data types
- Hypermedia

### 3.18.16. MUST maintain backwards compatibility for events

Changes to events must be based around making additive and backward compatible changes. This follows the guideline, "Must: Don't Break Backward Compatibility" from the Compatibility guidelines.

In the context of events, compatibility issues are complicated by the fact that producers and consumers of events are highly asynchronous and can't use content-negotiation techniques that are available to REST style clients and servers. This places a higher bar on producers to maintain compatibility as they will not be in a position to serve versioned media types on demand.

For event schema, these are considered backward compatible changes, as seen by consumers -

- Adding new optional fields to JSON objects.
- Changing the order of fields (field order in objects is arbitrary).
- Changing the order of values with same type in an array.
- Removing optional fields.
- Removing an individual value from an enumeration.

These are considered backwards-incompatible changes, as seen by consumers -

- Removing required fields from JSON objects.
- Changing the default value of a field.
- Changing the type of a field, object, enum or array.
- Changing the order of values with different type in an array (also known as a tuple).
- Adding a new optional field to redefine the meaning of an existing field (also known as a co-occurrence constraint).
- Adding a value to an enumeration (note that `x-extensible-enum` is not available in JSON Schema)

### 3.18.17. SHOULD avoid `additionalProperties` in event type definitions

Event type schema should avoid using `additionalProperties` declarations, in order to support schema evolution.

Events are often intermediated by publish/subscribe systems and are commonly captured in logs or long term storage to be read later. In particular, the schemas used by publishers and consumers can drift over time. As a result, compatibility and extensibility issues that happen less frequently with

client-server style APIs become important and regular considerations for event design. The guidelines recommend the following to enable event schema evolution:

- Publishers who intend to provide compatibility and allow their schemas to evolve safely over time **must not** declare an `additionalProperties` field with a value of `true` (i.e., a wildcard extension point). Instead they must define new optional fields and update their schemas in advance of publishing those fields.
- Consumers **must** ignore fields they cannot process and not raise errors. This can happen if they are processing events with an older copy of the event schema than the one containing the new definitions specified by the publishers.

The above constraint does not mean fields can never be added in future revisions of an event type schema - additive compatible changes are allowed, only that the new schema for an event type must define the field first before it is published within an event. By the same turn the consumer must ignore fields it does not know about from its copy of the schema, just as they would as an API client - that is, they cannot treat the absence of an `additionalProperties` field as though the event type schema was closed for extension.

Requiring event publishers to define their fields ahead of publishing avoids the problem of *field redefinition*. This is when a publisher defines a field to be of a different type that was already being emitted, or, is changing the type of an undefined field. Both of these are prevented by not using `additionalProperties`.

See also rule 111 in the Compatibility section for further guidelines on the use of `additionalProperties`.

### 3.18.18. MUST use unique event identifiers

The `eid` (event identifier) value of an event must be unique.

The `eid` property is part of the standard metadata for an event and gives the event an identifier. Producing clients must generate this value when sending an event and it must be guaranteed to be unique from the perspective of the owning application. In particular events within a given event type's stream must have unique identifiers. This allows consumers to process the `eid` to assert the event is unique and use it as an idempotency check.

Note that uniqueness checking of the `eid` might be not enforced by systems consuming events and it is the responsibility of the producer to ensure event identifiers do in fact distinctly identify events. A straightforward way to create a unique identifier for an event is to generate a UUID value.

### 3.18.19. SHOULD design for idempotent out-of-order processing

Events that are designed for idempotent out-of-order processing allow for extremely resilient systems: If processing an event fails, consumers and producers can skip/delay/retry it without stopping the world or corrupting the processing result.

To enable this freedom of processing, you must explicitly design for idempotent out-of-order processing: Either your events must contain enough information to infer their original order during consumption or your domain must be designed in a way that order becomes irrelevant.

As common example similar to data change events, idempotent out-of-order processing can be supported by sending the following information:

- the process/resource/entity identifier,
- a monotonically increasing ordering key and
- the process/resource state after the change.

A receiver that is interested in the current state can then ignore events that are older than the last processed event of each resource. A receiver interested in the history of a resource can use the ordering key to recreate a (partially) ordered sequence of events.

### 3.18.20. MUST follow naming convention for event type names

Event type names must (or should, see **MUST/SHOULD** use functional naming schema for details and definition) conform to the functional naming depending on the audience as follows:

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

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

<event-name>            ::= [a-z][a-z0-9-]* -- free event name (functional name)
```

The following application specific legacy convention is **only** allowed for internal event type names:

```
<application-event-name> ::= [<organization-id>.]<application-id>.<event-name>
<organization-id>  ::= [a-z][a-z0-9-]* -- organization identifier, e.g. team
identifier
<application-id>   ::= [a-z][a-z0-9-]* -- application identifier
```

**Note:** consistent naming should be used whenever the same entity is exposed by a data change event and a RESTful API.

### 3.18.21. MUST prepare event consumers for duplicate events

Event consumers must be able to process duplicate events.

Most message brokers and data streaming systems offer "at-least-once" delivery. That is, one particular event is delivered to the consumers one or more times. Other circumstances can also cause duplicate events.

For example, these situations occur if the publisher sends an event and doesn't receive the acknowledgment (e.g. due to a network issue). In this case, the publisher will try to send the same event again. This leads to two identical events in the event bus which have to be processed by the consumers. Similar conditions can appear on consumer side: an event has been processed successfully, but the consumer fails to confirm the processing.

# Appendix A: Tooling

This is not a part of the actual guidelines, but might be helpful for following them. Using a tool mentioned here doesn't automatically ensure you follow the guidelines.

### 3.A.1. API first integrations

The following frameworks were specifically designed to support the API First workflow with Open API YAML files (sorted alphabetically):

- **Swagger Codegen:** template-driven engine to generate client code in different languages by parsing Swagger Resource Declaration

The Swagger/Open API homepage lists more Community-Driven Language Integrations, but most of them do not fit our API First approach.

### 3.A.2. Support libraries

These utility libraries support you in implementing various parts of our RESTful API guidelines (sorted alphabetically):

# Appendix B: Best practices

The best practices presented in this section are not part of the actual guidelines, but should provide guidance for common challenges we face when implementing RESTful APIs.

### 3.B.1. Optimistic locking in RESTful APIs

#### 3.B.1.1. Introduction

Optimistic locking might be used to avoid concurrent writes on the same entity, which might cause data loss. A client always has to retrieve a copy of an entity first and specifically update this one. If another version has been created in the meantime, the update should fail. In order to make this work, the client has to provide some kind of version reference, which is checked by the service, before the update is executed. Please read the more detailed description on how to update resources via `PUT` in the HTTP Requests Section.

A RESTful API usually includes some kind of search endpoint, which will then return a list of result entities. There are several ways to implement optimistic locking in combination with search endpoints which, depending on the approach chosen, might lead to performing additional requests to get the current version of the entity that should be updated.

#### 3.B.1.2. `ETag` with `If-Match` header

An `ETag` can only be obtained by performing a `GET` request on the single entity resource before the update, i.e. when using a search endpoint an additional request is necessary.

Example:

```
< 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
```

Or, if there was an update since the `GET` and the entity's `ETag` has changed:

```
> HTTP/1.1 412 Precondition failed
```

**Pros**

- RESTful solution

**Cons**

- Many additional requests are necessary to build a meaningful front-end

#### 3.B.1.3. `ETags` in result entities

The ETag for every entity is returned as an additional property of that entity. In a response containing multiple entities, every entity will then have a distinct `ETag` that can be used in subsequent `PUT` requests.

In this solution, the `etag` property should be `readonly` and never be expected in the `PUT` request payload.

Example:

```
< GET /orders

> HTTP/1.1 200 OK
```

```
> {
>   "items": [
>     { "id": "O0000042", "etag": "osjnfkjbnkq3jlnksjnvkjlsbf", "foo": 42, "bar": true
},
>     { "id": "O0000043", "etag": "kjshdfknjqlowjdsljdnfkjbkn", "foo": 24, "bar":
false }
>   ]
> }

< PUT /orders/O0000042
< If-Match: osjnfkjbnkq3jlnksjnvkjlsbf
< { "id": "O0000042", "foo": 43, "bar": true }

> HTTP/1.1 204 No Content
```

Or, if there was an update since the `GET` and the entity's `ETag` has changed:

```
> HTTP/1.1 412 Precondition failed
```

**Pros**

- Perfect optimistic locking

**Cons**

- Information that only belongs in the HTTP header is part of the business objects

#### 3.B.1.4. Version numbers

The entities contain a property with a version number. When an update is performed, this version number is given back to the service as part of the payload. The service performs a check on that version number to make sure it was not incremented since the consumer got the resource and performs the update, incrementing the version number.

Since this operation implies a modification of the resource by the service, a `POST` operation on the exact resource (e.g. `POST /orders/O0000042`) should be used instead of a `PUT`.

In this solution, the `version` property is not `readonly` since it is provided at `POST` time as part of the payload.

Example:

```
< GET /orders

> HTTP/1.1 200 OK
> {
>   "items": [
>     { "id": "O0000042", "version": 1,  "foo": 42, "bar": true },
>     { "id": "O0000043", "version": 42, "foo": 24, "bar": false }
```

```
>   ]
> }

< POST /orders/O0000042
< { "id": "O0000042", "version": 1, "foo": 43, "bar": true }

> HTTP/1.1 204 No Content
```

or if there was an update since the `GET` and the version number in the database is higher than the one given in the request body:

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

**Pros**

- Perfect optimistic locking

**Cons**

- Functionality that belongs into the HTTP header becomes part of the business object
- Using `POST` instead of PUT for an update logic (not a problem in itself, but may feel unusual for the consumer)

#### 3.B.1.5. `Last-Modified` / `If-Unmodified-Since`

In HTTP 1.0 there was no `ETag` and the mechanism used for optimistic locking was based on a date. This is still part of the HTTP protocol and can be used. Every response contains a `Last-Modified` header with a HTTP date. When requesting an update using a `PUT` request, the client has to provide this value via the header `If-Unmodified-Since`. The server rejects the request, if the last modified date of the entity is after the given date in the header.

This effectively catches any situations where a change that happened between `GET` and `PUT` would be overwritten. In the case of multiple result entities, the `Last-Modified` header will be set to the latest date of all the entities. This ensures that any change to any of the entities that happens between `GET` and `PUT` will be detectable, without locking the rest of the batch as well.

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
```

Or, if there was an update since the `GET` and the entities last modified is later than the given date:

```
> HTTP/1.1 412 Precondition failed
```

**Pros**

- Well established approach that has been working for a long time
- No interference with the business objects; the locking is done via HTTP headers only
- Very easy to implement
- No additional request needed when updating an entity of a search endpoint result

**Cons**

- If a client communicates with two different instances and their clocks are not perfectly in sync, the locking could potentially fail

#### 3.B.1.6. Conclusion

We suggest to either use the `ETag` *in result entities* or `Last-Modified` / `If-Unmodified-Since` approach.

# 4. Development guidelines

## 4.1. General development guidelines

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
>
> — M. Fowler (1999)

### 4.1.1. Introduction

This chapter is about coding, developing or anything related to using a programming language to solve a problem.

> As programmers, we collect, organize, maintain, and harness knowledge. We document knowledge in specifications, we make it come alive in running code, and we use it to provide the checks needed during testing.

> — Pragmatic Programmer, Page 61

A problem is defined as a resolution for a challenge the business is facing. The mission of a coder is to write ”good”, ”clean” and ”secure” code. These terms are ill-defined.

At Pon we consider code ”good”, ”clean” and ”secure” based on the following rule of thumb

> How much effort is required[244] for another developer of comparable experience to pick up where the previous developer left off to fix, enhance or build upon the source code - without involving the former developer[245] and taking into account the lifetime, quality, security and the business impact[246] of the application.

There is a clear relationship between the amount of bugs in the code and the development effort required, needlessly complex illogical code will in itself be a source of new bugs. Moreover clear documentation and requirements from the business are the only solid foundation for software development.

The following chapters will further explain the intricacies of this rule.

### 4.1.2. Rules and definitions

Rules and definition used in the development guidelines.

### 4.1.3. Definition: code quality

> - Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product. It is the degree to which the correct software was produced.
> - Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability. It has a lot more to do with the degree to which the software works as needed.
>
> — Wikipedia

### 4.1.4. Coding rule: logical structured code

Refers to “*How much effort*”.

The amount of effort required for a change should be on-par with the apparent complexity of the change.

A logical structure, both for files and in the code itself, is essential. This also applies to the layout of

the code itself, for example indents.

*Updating an IP address*

Updating the IP address of the database; small apparent complexity, small effort required.

Good code will result in a single place to be updated. Clean code will result in easy to find code to be updated.

*Code layout*

Code layout may differ greatly between projects, but within the same project all files strictly adhere to the same code layout standards. Automated tools based on accepted code style standards are essential for enforcement.

### 4.1.5. Coding rule: code is simple and concise

Refers to "*How much effort*".

Code is simple and concise in order to increase the readability [praeng], rule 4 [nasa-safety-code]. Increasing performance by sacrificing readability is only an option if the performance increase is in-line with the business requirements.

> Rule: No function should be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function.
>
> Rationale: Each function should be a logical unit in the code that is understandable and verifiable as a unit. It is much harder to understand a logical unit that spans multiple screens on a computer display or multiple pages when printed. Excessively long functions are often a sign of poorly structured code.
>
> — Nasa coding rules, Rule 4

*Simple and concise*

Based on the experience level of the developer simple and concise can easily shift to unreadable and complex. Code should not be overly simplified in order for all to understand, it is up to the lead developer to guide and train the less experienced developers in writing and understanding simple and concise code.

### 4.1.6. Coding rule: do not repeat yourself (DRY)

Refers to "*How much effort*".

> Don’t repeat yourself (DRY, or sometimes do not repeat yourself) is a principle of software development aimed at reducing repetition of software patterns, replacing it with abstractions or using data normalization to avoid redundancy.
>
> The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system". The principle has been formulated by Andy Hunt and Dave Thomas in their book The Pragmatic Programmer. They apply it quite broadly to include "database schemas, test plans, the build system, even documentation". When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements. Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync.
>
> — Wikipedia

Or, simply put:

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
>
> — Pragmatic Programmer, Page 62

However keep in mind that if two distinct pieces of knowledge result in the same code the DRY principle should be reviewed carefully as shown in [accidental-doppelganger]

> The once and only once rule is one of the most fundamental principles of software development, it can be seen as a hallmark of good and clean code and will greatly reduce developer effort when applied correctly. But it is not the goal of software development; the goal is resolving challenges of the business under their conditions.
>
> Ref: https://gtramontina.com/posts/do-repeat-yourself.html

### 4.1.7. Coding rule: code and code changes are self-explanatory

Refers to “*without involving the previous developer*”.

All code is sufficiently documented in order to reduce the effort[244] required for updates and changes. Comments must explain the why, not the how [praeng].

- Code changes are documented and should contain a reference to an issue tracking system
- Deviation from guidelines is always documented

When readability is sacrificed for performance[254] it is reflected in the comments.

Remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments

Ref: [googleStyleguideCpp], #Comments

### 4.1.8. Coding rule: solution design steps are template-based

Refers to "*taking into account the lifetime, quality, security and business impact*".

Solution design comes first, coding second. The solution design must address the following

- Software lifetime
- Required quality
- Required security
- Business impact

### 4.1.9. Coding rule: code quality is known

Based on the quality as discussed in the solution design steps the code quality must be known.

This rule does not state that code must be fully automatically tested and scoring 100/100 on quality. This rule states that the quality, as agreed upon beforehand with the business, is known and documented.

### 4.1.10. Coding rule: cyclomatic complexity is low

Refers to "*How much effort*".

Keep the number of conditional statements to a minimum; rule 1 of [nasa-safety-code].

> The cyclomatic complexity of a section of source code is the number of linearly independent paths within it—where "linearly independent" means that each path has at least one edge that is not in one of the other paths. For instance, if the source code contained no control flow statements (conditionals or decision points), the complexity would be 1, since there would be only a single path through the code. If the code had one single-condition IF statement, there would be two paths through the code: one where the IF statement evaluates to TRUE and another one where it evaluates to FALSE, so the complexity would be 2. Two nested single-condition IFs, or one IF with two conditions, would produce a complexity of 3.
>
> — Wikipedia

Only use an *else* statement if required. Prefer a switch statement over multiple if-then-else constructs.

## 4.2. File structure and naming

### 4.2.1. (RFP) MUST add comment to file

Each source file is self explanatory and must contain comments showing at least the following

- Purpose of the file in a concise description
- Author of the file

### 4.2.2. (RFP) MUST filenames are either CamelCase or snake_case

Filenames

## 4.3. Version control

### 4.3.1. MUST use review guidelines for version control

All code must be reviewed according to documented and approved guidelines. Relates to Coding rule: logical structured code.

#### 4.3.1.1. Review guidelines

- Start reviewing only if the author approved the pull request (PR) and the pipeline passed the linting and automated tests.
- A review will always result in either an approved or declined PR.
- A declined PR must be recreated, preferably by the author, when issues have been addressed.
- Write comments in a clear, concise, constructive and unambiguous manner as described in Coding rule: code and code changes are self-explanatory.

## 4.4. Testing code

### 4.4.1. MUST use automated linter based on approved style template

The linter configuration is selected from the solution architecture repository.

### 4.4.2. MUST use automated tests based on approved testing template

Code changes must be covered by automated tests. When choosing how to cover your changes, pick the most lightweight (execution time wise) test type that will provide sufficient coverage. If you encounter an existing test that insufficiently covers your changes, you can delete that test but you must write a proper test to replace it. For example, a method that interacts with database has a unit test. You can replace it with an integration test.

Be aware that while high-level tests may provide coverage to code, it is only indirect coverage. Tests with more direct usage of the changed code will likely still needs to be written to ensure that regardless of the which components are actually used in the black box, the individual components still have coverage.

These tests must test the concrete implementation's behavior in a way that can not be inadvertently changed outside of the test itself.

# 4.5. Monitoring & logging

Logging and monitoring is essential for software, they business should always be able to confirm software is functioning correctly.

### 4.5.1. SHOULD use dedicated logging library and logging levels

Logging directly to the console or command line is not recommended; it reduces the effectivess of debugging, monitoring and checking the operational status of the software. It increases the security risk because it is challenging to detect if senstive data is being logged.

To effectively use the library logging levels (DEBUG, ERROR, NOTICE) should be used.

# 4.6. Development environment

Which tools are required for a developer to be at peak efficiency?

TODO : https://github.com/pondigitalsolutions/restful-api-guidelines/issues/16

# 4.7. Development background

This chapter further explores what it is to be a developer, it does not contain guidelines or standards per-se but it can be of guidance for developers.

PP 25

Apollo / NASA

https://fermatslibrary.com/s/apollo-11-implementation-of-trigonometric-functions

# 4.8. Date and time handling

This chapter handles the date and the time guidelines, including time intervals and durations.

### 4.8.1. (RFP) MUST use RFC 3339 for time and date encoding

The international standard RFC-3339 must be used for time and date encoding unless not feasible due to technical constraints.

### 4.8.2. (RFP) MUST date time manipulation must be handled by a library

Date and time manipulation is very complex and involves many one-offs, and thus very error-prone. Date and time manipulation must be handled by a dedicated library, which is available for all main development languages.

### 4.8.3. SHOULD define time durations and intervals properties conform to RFC 3339

Schema-based JSON properties that are by design durations and intervals could be strings formatted as recommended by RFC-3339 (Appendix A of RFC 3339 contains a grammar for durations).

# Appendix C: Pon Standard Style

Pon standard style can be applied when linters are not available.

Based on [standardJs], [googleStyleguideJs], [phpStandards], [nasa-safety-code] and [gnu-coding-standards].

### 4.C.1. MUST encapsulate body of if or else

The body of an if or else must be encapsulated it increases readability and decreases the introduction of bugs.

For single-line statements a ternary is preferred.

#### 4.C.1.1. Example 1

```
if (a < b)
  test(c)
load(b)

if (a < b)
  //test(c)
load(b)
```

Commenting out one line for debugging purposes resulted in the addition of a new bug, this can easily be avoid by using curly braces:

```
if (a < b) {
  test(c)
}
load(b)

if (a < b) {
```

```
  //test(c)
}
load(b)
```

### 4.C.2. SHOULD order if statements by increased complexity

If using if statements where the conditionals are related, they should be ordered by increased complexity when possible. Using this ordering will significantly improve readability.

#### 4.C.2.1. Example 1

```
if (!invoicePaid && !email) {
  sendInvoiceByMail();
} else if (invoicePaid) {
  closeOrder();
}
```

By swapping the if statements the logic is more comfortable to follow:

```
if (invoicePaid) {
  closeOrder();
} else if (!invoicePaid && !email) {
  sendInvoiceByMail();
}
```

### 4.C.3. MUST use special quotes only to reduce complexity

In most programming languages several options are available for the quoting of text, in the following examples Javascript style quoting is used which is applicable to a variety of other programming languages.

For the following examples the key question is: can the readbility be improved to make the code more simple and concise?

#### 4.C.3.1. Example 1

```
text = "The quick brown fox jumped over the lazy dog"
```

Although the code is correct the double quotes have have no additional function, the following code is more concise by using single quotes:

```
text = 'The quick brown fox jumped over the lazy dog'
```

#### 4.C.3.2. Example 2

```
text = "The quick brown \"fox\" jumped over the lazy dog"
```

The code is correct, but readability is improved by switching to single quotes:

```
text = 'The quick brown "fox" jumped over the lazy dog'
```

#### 4.C.3.3. References

- [googleStyleguideJs], Use single quotes
- [standardJs], Use single quotes for strings
- Rules and definitions, Coding rule: logical structured code
- Rules and definitions, Coding rule: code is simple and concise

### 4.C.4. SHOULD never use tabs for indentation

Tabs are hould never be used for indentation. A notable exception is Go where tabs are a language default.

#### 4.C.4.1. References

- [googleStyleguideJs], whitespace characters (2)
- [standardJs], Tabs should not be used
- [phpStandards], indenting
- Rules and definitions, Coding rule: logical structured code
- Rules and definitions, Coding rule: code is simple and concise

### 4.C.5. MUST use predefined spacing for indentation

Using predefined spacing for indentation results in clear readable code without sacrificing too much screen real estate; **SHOULD** never use tabs for indentation.

When using multiple languages a single project it is preferred to use the same indent for all languages involved.

#### 4.C.5.1. References

- [googleStyleguideJs], whitespace characters (2)
- [standardJs], Use 2 spaces for indentation

- [phpStandards], indenting
- Rules and definitions, Coding rule: logical structured code

### 4.C.6. SHOULD check return types of non-void functions

Return values of functions should not be ignored, especially if error return values must be propagated up the function call chain. By checking return types exception justification is enforced, which will result in increased code stability.

### 4.C.7. References

- [nasa-safety-code], Rule 7

### 4.C.8. SHOULD check the validity of parameters inside each function

Input parameters should not be assumed to be valid; by checking the validity code stability is increased.

#### 4.C.8.1. Example 1

```
// Lodash - startsWith.js  - https://github.com/lodash/lodash
function startsWith(string, target, position) {
  const { length } = string
  position = position == null ? 0 : position
  if (position < 0) {
    position = 0
  }
  else if (position > length) {
    position = length
  }
  target = `${target}`
  return string.slice(position, position + target.length) == target
}
```

Note the majority of the code in example 1 is about checking the input parameters.

#### 4.C.8.2. References

- [nasa-safety-code], Rule 7
- The Robustness Principle

### 4.C.9. MUST not have unused variables

All variables are in use, unused variables have no function and are cluttering the code.

#### 4.C.9.1. References

- [standardJs], No unused variables
- Rules and definitions, Coding rule: logical structured code

### 4.C.10. SHOULD use < or > instead of <= or >=

Using < is preferred over using <=, using > is preferred over using >=. It improves readability and performance of code.

### 4.C.11. SHOULD use != instead of > or < when only a single value results in false

If a return value always results in true except for a single value using != is preferred over using > or >=.

#### 4.C.11.1. Example 1

```
languages = ['NL', 'FR', 'BE'];

if (languages.indexOf('BE') >= 0)
```

The code is correct, but readability is reduced by using the >=, especially since the return of the indexOf function is counterintuitive, a more readable solution is:

```
languages = ['NL', 'FR', 'BE'];

if (languages.indexOf('BE') != -1)
```

# Appendix D: Pon Standard Style - Go

Patron(s) □ Dennis Verweij

dennis.verweij@pon.com

### 4.D.1. MUST for linting we use golangci-lint in our CI/CD system

In our IDE and CI flow we use golangci-lint as linter. Follow the installation steps for you system. Golangci-lint has integrations with various CI systems and IDE's. golangci-lint runs the most important code checks by default. In our CI/CD the default lint settings of golangci-lint are mandatory.

#### 4.D.1.1. Example linter implementation in Git Actions

```
qacheck:
    name: Run go tests
    runs-on: ubuntu-latest
    steps:
    - name: Set up Go 1.x
      uses: actions/setup-go@v2
      with:
        go-version: ^1.13
      id: go
    - name: Check out code into the Go module directory
      uses: actions/checkout@v2
    - name: golangci-lint
      uses: golangci/golangci-lint-action@v1
      with:
        version: v1.29
```

### 4.D.2. SHOULD go Vet is used to check go code for correctness in the development process

We preferably configure our IDE to run go Vet on every save operation.

### 4.D.3. MUST go Vet is used to check go code for correctness in the build pipeline

No code may be build without a code check. This is done by default when rule 270 is applied.

### 4.D.4. MUST use tabs for indentation in Go

Unlike the development guidelines for other languages in Go we use tabs for indentation. To improve formatting of code we use gofmt to automatically format go code in de IDE.

### 4.D.5. MUST use gofmt in the IDE for automatic formatting

In Go you don't worry about formatting yourself and use gofmt for automatic formating. Gofmt directions are leading.

### 4.D.6. {SHALL} every function is commented

In Go exported functions must be commented and is enforced by linting. Comments on unexported functions are not enforced by linting, but improve readablity of the code.

### 4.D.7. MUST single line multiple declarations are not used

For readability only one variable is declared in a single line

#### 4.D.7.1. Example 1 Invalid declaration

```
var valid, found, required bool
```

#### 4.D.7.2. Example 2 Valid declaration

```
var valid bool
var found bool
var required bool
```

### 4.D.8. MUST we do not try and catch exceptions. Errors are values and we handle errors

In some languages try/catch statements are used. Go does not have that method and we do not try to mimic it. Instead we handle the errors and the errors are just values. You need to assess the value and act upon it.

### 4.D.9. MUST errors are handle only once.

We handle errors only once.

#### 4.D.9.1. Don't do

Double logs for the same error.

```
func doSomething(val string) (string, error){
    // Do something with val that results in a doneValue and an error value
    if err != nil {
        log.Error(err)
        return doneValue,err
    }
    return doneValue, nil
}

func something(){
    val := "some stuff"
    result, err := doSomething(val)
    if err != nil {
        log.Error(err)
        // Handle the error
    }
}
```

#### 4.D.9.2. Better

```
func doSomething(val string) (string, error){
```

```
    // Do something with val that results in a doneValue and an error value
    return doneValue, err
}

func something() {
    val := "some stuff"
    result, err := doSomething(val)
    if err != nil {
        log.Error(err)
        // Handle the error
    }
}
```

#### 4.D.9.3. We can also include the stacktrace in the logging

The package github.com/pkg/errors gives more options than the default errors package. You can log the stacktrace.

```
import (
    "github.com/pkg/errors"
    log "github.com/sirupsen/logrus"
)
func doSomething(val string) (string, error){
    // Do something with val that results in a doneValue and an error value
    return doneValue, err
}

func something() {
    val := "some stuff"
    result, err := doSomething(val)
    if err != nil {
        log.Errorf("%+v", err)
        // Handle the error
    }
}
```

### 4.D.10. SHOULD add context to errors when they are meaningless in the context of the (final) receiver.

When errors are passed it might eventualy be unclear what the origin of the error is. You can pass context to it, but be carefull with fmt.Errorf(), because that will override the initial error with just a string.

#### 4.D.10.1. Passing through context of the error with fmt.Errorf()

Using fmt.ErrorF() overwrites the error and returns just a string. Sometimes it's just fine, but be aware of the consequences

```
import (
    log "github.com/sirupsen/logrus"
)
func doSomething(val string) (string, error){
    // Do something with val that results in a doneValue and an error value
    if err != nil {
        err = fmt.Errorf("Something whent wrong processing %s: %v", val, err)
    }
    return doneValue, err
}

func something() {
    val := "some stuff"
    result, err := doSomething(val)
    if err != nil {
        log.Errorf("%+v", err)
        // Handle the error
    }
}
```

This will return : overwritten error: test The stacktrace is gone

#### 4.D.10.2. Better → Passing through context of the error with errors.Wrap() from the "github.com/pkg/errors" package

Using errors.Wrap() adds your context to the error stack

```
import (
    "github.com/pkg/errors"
    log "github.com/sirupsen/logrus"
)
func doSomething(val string) (string, error){
    // Do something with val that results in a doneValue and an error value
    if err != nil {
        err = errors.Wrap(err, "Something whent wrong processing")
    }
    return doneValue, err
}

func something() {
    val := "some stuff"
    result, err := doSomething(val)
    if err != nil {
        log.Errorf("%+v", err)
        // Handle the error
    }
}
```

# Appendix E: Pon Standard Style - Magento

Patron(s) □ ?

# Appendix F: Pon Standard Style - WordPress

Patron(s) □ Roy van der Loo

# 5. Networking

## 5.1. HTTP requests

### 5.1.1. MUST use HTTP methods correctly

Be compliant with the standardized HTTP method semantics summarized as follows:

#### 5.1.1.1. GET

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

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

**Note:** `GET` requests on collection resources should provide sufficient filter and Pagination mechanisms.

#### 5.1.1.2. GET with body

APIs sometimes face the problem, that they have to provide extensive structured request information with `GET`, that may conflict with the size limits of clients, load-balancers, and servers. As we require APIs to be standard conform (body in `GET` must be ignored on server side), API designers have to check the following two options:

1. `GET` with URL encoded query parameters: when it is possible to encode the request information in query parameters, respecting the usual size limits of clients, gateways, and servers, this should be the first choice. The request information can either be provided via multiple query parameters or by a single structured URL encoded string.
2. `POST` with body content: when a `GET` with URL encoded query parameters is not possible, a `POST` with body content must be used. In this case the endpoint must be documented with the hint `GET With Body` to transport the `GET` semantic of this call.

**Note:** It is no option to encode the lengthy structured request information using header parameters. From a conceptual point of view, the semantic of an operation should always be expressed by the resource names, as well as the involved path and query parameters. In other

words by everything that goes into the URL. Request headers are reserved for general context information (see **MUST** use only the specified proprietary Pon headers). In addition, size limits on query parameters and headers are not reliable and depend on clients, gateways, server, and actual settings. Thus, switching to headers does not solve the original problem.

**Hint:** As `GET With Body` is used to transport extensive query parameters, the `cursor` cannot any longer be used to encode the query filters in case of cursor-based pagination. As a consequence, it is best practice to transport the query filters in the body, while using pagination links containing the `cursor` that is only encoding the page position and direction. To protect the pagination sequence the `cursor` may contain a hash over all applied query filters (See also **SHOULD** use pagination links where applicable).

#### 5.1.1.3. PUT

`PUT` requests are used to **update** (in rare cases to create) **entire** resources – single or collection resources. The semantic is best described as *"please put the enclosed representation at the resource mentioned by the URL, replacing any existing resource."*.

- `PUT` requests are usually applied to single resources, and not to collection resources, as this would imply replacing the entire collection
- `PUT` requests are usually robust against non-existence of resources by implicitly creating before updating
- on successful `PUT` requests, the server will **replace the entire resource** addressed by the URL with the representation passed in the payload (subsequent reads will deliver the same payload)
- successful `PUT` requests will usually generate 200 or 204 (if the resource was updated – with or without actual content returned), and 201 (if the resource was created)

**Important:** It is best practice to prefer `POST` over `PUT` for creation of (at least top-level) resources. This leaves the resource ID under control of the service and allows to concentrate on the update semantic using `PUT` as follows.

**Note:** In the rare cases where `PUT` is although used for resource creation, the resource IDs are maintained by the client and passed as a URL path segment. Putting the same resource twice is required to be idempotent and to result in the same single resource instance (see **MUST** fulfill common method properties).

**Hint:** To prevent unnoticed concurrent updates and duplicate creations when using `PUT`, you **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header to allow the server to react on stricter demands that expose conflicts and prevent lost updates. See also Optimistic locking in RESTful APIs for details and options.

#### 5.1.1.4. POST

`POST` requests are idiomatically used to **create** single resources on a collection resource endpoint, but other semantics on single resources endpoint are equally possible. The semantic for collection endpoints is best described as *"please add the enclosed representation to the collection resource identified by the URL"*.

- on a successful `POST` request, the server will create one or multiple new resources and provide

their URI/URLs in the response

- successful `POST` requests will usually generate 200 (if resources have been updated), 201 (if resources have been created), 202 (if the request was accepted but has not been finished yet), and exceptionally 204 with `Location` header (if the actual resource is not returned).

The semantic for single resource endpoints is best described as *"please execute the given well specified request on the resource identified by the URL"*.

**Generally:** `POST` should be used for scenarios that cannot be covered by the other methods sufficiently. In such cases, make sure to document the fact that `POST` is used as a workaround (see `GET With Body`).

**Note:** Resource IDs with respect to `POST` requests are created and maintained by server and returned with response payload.

**Hint:** Posting the same resource twice is **not** required to be idempotent (check **MUST** fulfill common method properties) and may result in multiple resources. However, you **SHOULD** consider to design `POST` and `PATCH` idempotent to prevent this.

#### 5.1.1.5. PATCH

`PATCH` requests are used to **update parts** of single resources, i.e. where only a specific subset of resource fields should be replaced. The semantic is best described as *"please change the resource identified by the URL according to my change request"*. The semantic of the change request is not defined in the HTTP standard and must be described in the API specification by using suitable media types.

- `PATCH` requests are usually applied to single resources as patching entire collection is challenging
- `PATCH` requests are usually not robust against non-existence of resource instances
- on successful `PATCH` requests, the server will update parts of the resource addressed by the URL as defined by the change request in the payload
- successful `PATCH` requests will usually generate 200 or 204 (if resources have been updated with or without updated content returned)

**Note:** since implementing `PATCH` correctly is a bit tricky, we strongly suggest to choose one and only one of the following patterns per endpoint, unless forced by a backwards compatible change. In preference order:

1. use `PUT` with complete objects to update a resource as long as feasible (i.e. do not use `PATCH` at all).
2. 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.)
3. use `PATCH` with JSON Patch, a specialized media type `application/json-patch+json` that includes instructions on how to change the resource.
4. use `POST` (with a proper description of what is happening) instead of `PATCH`, if the request does not modify the resource in a way defined by the semantics of the media type.

In practice JSON Merge Patch quickly turns out to be too limited, especially when trying to update single objects in large collections (as part of the resource). In this cases JSON Patch can shown its full power while still showing readable patch requests (see also JSON patch vs. merge).

**Note:** Patching the same resource twice is **not** required to be idempotent (check **MUST** fulfill common method properties) and may result in a changing result. However, you **SHOULD** consider to design `POST` and `PATCH` idempotent to prevent this.

**Hint:** To prevent unnoticed concurrent updates when using `PATCH` you **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header to allow the server to react on stricter demands that expose conflicts and prevent lost updates. See Optimistic locking in RESTful APIs and **SHOULD** consider to design `POST` and `PATCH` idempotent for details and options.

#### 5.1.1.6. DELETE

`DELETE` requests are used to **delete** resources. The semantic is best described as *"please delete the resource identified by the URL"*.

- `DELETE` requests are usually applied to single resources, not on collection resources, as this would imply deleting the entire collection
- successful `DELETE` requests will usually generate 200 (if the deleted resource is returned) or 204 (if no content is returned)
- failed `DELETE` requests will usually generate 404 (if the resource cannot be found) or 410 (if the resource was already deleted before)

**Important:** After deleting a resource with `DELETE`, a `GET` request on the resource is expected to either return 404 (not found) or 410 (gone) depending on how the resource is represented after deletion. Under no circumstances the resource must be accessible after this operation on its endpoint.

#### 5.1.1.7. HEAD

`HEAD` requests are used to **retrieve** the header information of single resources and resource collections.

- `HEAD` has exactly the same semantics as `GET`, but returns headers only, no body.

**Hint:** `HEAD` is particular useful to efficiently lookup whether large resources or collection resources have been updated in conjunction with the `ETag`-header.

#### 5.1.1.8. OPTIONS

`OPTIONS` requests are used to **inspect** the available operations (HTTP methods) of a given endpoint.

- `OPTIONS` responses usually either return a comma separated list of methods in the `Allow` header or as a structured list of link templates

**Note:** `OPTIONS` is rarely implemented, though it could be used to self-describe the full functionality of a resource.

### 5.1.2. MUST fulfill common method properties

Request methods in RESTful services can be…

- safe - the operation semantic is defined to be read-only, meaning it must not have *intended side effects*, i.e. changes, to the server state.
- idempotent - the operation has the same *intended effect* on the server state, independently whether it is executed once or multiple times. **Note:** this does not require that the operation is returning the same response or status code.
- cacheable - to indicate that responses are allowed to be stored for future reuse. In general, requests to safe methods are cachable, if it does not require a current or authoritative response from the server.

**Note:** The above definitions, of *intended (side) effect* allows the server to provide additional state changing behavior as logging, accounting, pre- fetching, etc. However, these actual effects and state changes, must not be intended by the operation so that it can be held accountable.

Method implementations must fulfill the following basic properties according to RFC 7231:

| Method | Safe | Idempotent | Cacheable |
|---|---|---|---|
| `GET` | ✓ Yes | ✓ Yes | ✓ Yes |
| `HEAD` | ✓ Yes | ✓ Yes | ✓ Yes |
| `POST` | ✗ No | ✗ No, but **SHOULD** consider to design `POST` and `PATCH` idempotent | ✓ May, but only if specific `POST` endpoint is safe. **Hint:** not supported by most caches. |
| `PUT` | ✗ No | ✓ Yes | ✗ No |
| `PATCH` | ✗ No | ✗ No, but **SHOULD** consider to design `POST` and `PATCH` idempotent | ✗ No |
| `DELETE` | ✗ No | ✓ Yes | ✗ No |
| `OPTIONS` | ✓ Yes | ✓ Yes | ✗ No |
| `TRACE` | ✓ Yes | ✓ Yes | ✗ No |

**Note:** **MUST** document cachable `GET`, `HEAD`, and `POST` endpoints.

### 5.1.3. SHOULD consider to design `POST` and `PATCH` idempotent

In many cases it is helpful or even necessary to design `POST` and `PATCH` idempotent for clients to expose conflicts and prevent resource duplicate (a.k.a. zombie resources) or lost updates, e.g. if same resources may be created or changed in parallel or multiple times. To design an idempotent API endpoint owners should consider to apply one of the following three patterns.

- A resource specific **conditional key** provided via `If-Match` header in the request. The key is in general a meta information of the resource, e.g. a *hash* or *version number*, often stored with it. It allows to detect concurrent creations and updates to ensure idempotent behavior (see **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header).

- A resource specific **secondary key** provided as resource property in the request body. The *secondary key* is stored permanently in the resource. It allows to ensure idempotent behavior by looking up the unique secondary key in case of multiple independent resource creations from different clients (see **SHOULD** use secondary key for idempotent `POST` design).
- A client specific **idempotency key** provided via `Idempotency-Key` header in the request. The key is not part of the resource but stored temporarily pointing to the original response to ensure idempotent behavior when retrying a request (see **MAY** consider to support `Idempotency-Key` header).

**Note:** While **conditional key** and **secondary key** are focused on handling concurrent requests, the **idempotency key** is focused on providing the exact same responses, which is even a *stronger* requirement than the idempotency defined above. It can be combined with the two other patterns.

To decide, which pattern is suitable for your use case, please consult the following table showing the major properties of each pattern:

| | Conditional Key | Secondary Key | Idempotency Key |
|---|---|---|---|
| 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 |

**Note:** The patterns applicable to `PATCH` can be applied in the same way to `PUT` and `DELETE` providing the same properties.

If you mainly aim to support safe retries, we suggest to apply conditional key and secondary key pattern before the Idempotency Key pattern.

### 5.1.4. SHOULD use secondary key for idempotent `POST` design

The most important pattern to design `POST` idempotent for creation is to introduce a resource specific **secondary key** provided in the request body, to eliminate the problem of duplicate (a.k.a zombie) resources.

The secondary key is stored permanently in the resource as *alternate key* or *combined key* (if consisting of multiple properties) guarded by a uniqueness constraint enforced server-side, that is visible when reading the resource. The best and often naturally existing candidate is a *unique foreign key*, that points to another resource having *one-on-one* relationship with the newly created resource, e.g. a parent process identifier.

A good example here for a secondary key is the shopping cart ID in an order resource.

**Note:** When using the secondary key pattern without `Idempotency-Key` all subsequent retries should fail with status code 409 (conflict). We suggest to avoid 200 here unless you make sure, that the delivered resource is the original one implementing a well defined behavior. Using 204 without content would be a similar well defined option.

### 5.1.5. MUST define collection format of header and query parameters

Header and query parameters allow to provide a collection of values, either by providing a comma-separated list of values or by repeating the parameter multiple times with different values as follows:

| 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 |

As Open API does not support both schemas at once, an API specification must explicitly define the collection format to guide consumers as follows:

| 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` |

When choosing the collection format, take into account the tool support, the escaping of special characters and the maximal URL length.

### 5.1.6. SHOULD design simple query languages using query parameters

We prefer the use of query parameters to describe resource-specific query languages for the majority of APIs because it's native to HTTP, easy to extend and has excellent implementation support in HTTP clients and web frameworks.

Query parameters should have the following aspects specified:

- Reference to corresponding property, if any
- Value range, e.g. inclusive vs. exclusive
- Comparison semantics (equals, less than, greater than, etc)
- Implications when combined with other queries, e.g. *and* vs. *or*

How query parameters are named and used is up to individual API designers. The following examples should serve as ideas:

- `name=Pon`, to query for elements based on property equality

- `age=5`, to query for elements based on logical properties
  - Assuming that elements don't actually have an `age` but rather a `birthday`
- `max_length=5`, based on upper and lower bounds (`min` and `max`)
- `shorter_than=5`, using terminology specific e.g. to *length*
- `created_before=2019-07-17` or `not_modified_since=2019-07-17`
  - Using terminology specific e.g. to time: *before*, *after*, *since* and *until*

We don't advocate for or against certain names because in the end APIs should be free to choose the terminology that fits their domain the best.

### 5.1.7. SHOULD design complex query languages using JSON

Minimalistic query languages based on query parameters are suitable for simple use cases with a small set of available filters that are combined in one way and one way only (e.g. *and* semantics). Simple query languages are generally preferred over complex ones.

Some APIs will have a need for sophisticated and more complex query languages. Dominant examples are APIs around search (incl. faceting) and product catalogs.

Aspects that set those APIs apart from the rest include but are not limited to:

- Unusual high number of available filters
- Dynamic filters, due to a dynamic and extensible resource model
- Free choice of operators, e.g. `and`, `or` and `not`

APIs that qualify for a specific, complex query language are encouraged to use nested JSON data structures and define them using Open API directly. The provides the following benefits:

- Data structures are easy to use for clients
  - No special library support necessary
  - No need for string concatenation or manual escaping
- Data structures are easy to use for servers
  - No special tokenizers needed
  - Semantics are attached to data structures rather than text tokens
- Consistent with other HTTP methods
- API is defined in Open API completely
  - No external documents or grammars needed
  - Existing means are familiar to everyone

JSON-specific rules and most certainly needs to make use of the `GET`-with-body pattern.

#### 5.1.7.1. Example

The following JSON document should serve as an idea how a structured query might look like.

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

Feel free to also get some inspiration from:

- Elastic Search: Query DSL
- GraphQL: Queries

### 5.1.8. MUST document implicit filtering

Sometimes certain collection resources or queries will not list all the possible elements they have, but only those for which the current client is authorized to access.

Implicit filtering could be done on:

- the collection of resources being return on a parent `GET` request
- the fields returned for the resource's detail

In such cases, the implicit filtering must be in the API specification (in its description).

Consider caching considerations when implicitly filtering.

Example:

If an employee of the company *Foo* accesses one of our business-to-business service and performs a `GET /business-partners`, it must, for legal reasons, not display any other business partner that is not owned or contractually managed by her/his company. It should never see that we are doing business also with company *Bar*.

Response as seen from a consumer working at `FOO`:

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

Response as seen from a consumer working at BAR:

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

The API Specification should then specify something like this:

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

# 5.2. HTTP status codes and errors

## 5.2.1. MUST specify success and error responses

APIs should define the functional, business view and abstract from implementation aspects. Success and error responses are a vital part to define how an API is used correctly.

Therefore, you must 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.

**Hint:** In most cases it is not useful to document all technical errors, especially if they are not under control of the service provider. Thus unless a response code conveys application-specific functional semantics or is used in a none standard way that requires additional explanation, multiple error response specifications can be combined using the following pattern (see also **(RFP) MUST** only use durable and immutable remote references):

```
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 designers should also think about a **troubleshooting board** as part of the associated online API documentation. It provides information and handling guidance on application-specific errors and is referenced via links from the API specification. This can reduce service support tasks and contribute to service client and provider performance.

### 5.2.2. MUST use standard HTTP status codes

You must only use standardized HTTP status codes consistently with their intended semantics. You must not invent new HTTP status codes.

RFC standards define ~60 different HTTP status codes with specific semantics (mainly RFC7231 and RFC 6585) — and there are upcoming new ones, e.g. draft legally-restricted-status. See overview on all error codes on Wikipedia or via https://httpstatuses.com/) also inculding 'unofficial codes', e.g. used by popular web servers like Nginx.

Below we list the most commonly used and best understood HTTP status codes, consistent with their semantic in the RFCs. APIs should only use these to prevent misconceptions that arise from less commonly used HTTP status codes.

**Important:** As long as your HTTP status code usage is well covered by the semantic defined here, you should not describe it to avoid an overload with common sense information and the risk of inconsistent definitions. Only if the HTTP status code is not in the list below or its usage requires additional information aside the well defined semantic, the API specification must provide a clear description of the HTTP status code in the response.

#### 5.2.2.1. Success codes

| Code | Meaning | Methods |
|---|---|---|
| 200 | OK - this is the standard success response | `<all>` |
| 201 | Created - Returned on successful entity creation. You are free to return either an empty response or the created resource in conjunction with the Location header. (More details found in the Common headers.) *Always* set the Location header. | `POST`, `PUT` |
| 202 | Accepted - The request was successful and will be processed asynchronously. | `POST`, `PUT`, `PATCH`, `DELETE` |
| 204 | No content - There is no response body. | `PUT`, `PATCH`, `DELETE` |

| Code | Meaning | Methods |
|---|---|---|
| 207 | Multi-Status - The response body contains multiple status informations for different parts of a batch/bulk request (see **MUST** use code 207 for batch or bulk requests). | POST |

#### 5.2.2.2. Redirection codes

| Code | Meaning | Methods |
|---|---|---|
| 301 | Moved Permanently - This and all future requests should be directed to the given URI. | <all> |
| 303 | See Other - The response to the request can be found under another URI using a GET method. | POST, PUT, PATCH, DELETE |
| 304 | Not Modified - indicates that a conditional GET or HEAD request would have resulted in 200 response if it were not for the fact that the condition evaluated to false, i.e. resource has not been modified since the date or version passed via request headers If-Modified-Since or If-None-Match. | GET, HEAD |

#### 5.2.2.3. Client side error codes

| Code | Meaning | Methods |
|---|---|---|
| 400 | Bad request - generic / unknown error. Should also be delivered in case of input payload fails business logic validation. | <all> |
| 401 | Unauthorized - the users must log in (this often means "Unauthenticated"). | <all> |
| 403 | Forbidden - the user is not authorized to use this resource. | <all> |
| 404 | Not found - the resource is not found. | <all> |
| 405 | Method Not Allowed - the method is not supported, see OPTIONS. | <all> |
| 406 | Not Acceptable - resource can only generate content not acceptable according to the Accept headers sent in the request. | <all> |
| 408 | Request timeout - the server times out waiting for the resource. | <all> |
| 409 | Conflict - request cannot be completed due to conflict, e.g. when two clients try to create the same resource or if there are concurrent, conflicting updates. | POST, PUT, PATCH, DELETE |
| 410 | Gone - resource does not exist any longer, e.g. when accessing a resource that has intentionally been deleted. | <all> |
| 412 | Precondition Failed - returned for conditional requests, e.g. If-Match if the condition failed. Used for optimistic locking. | PUT, PATCH, DELETE |
| 415 | Unsupported Media Type - e.g. clients sends request body without content type. | POST, PUT, PATCH, DELETE |
| 423 | Locked - Pessimistic locking, e.g. processing states. | PUT, PATCH, DELETE |

| Code | Meaning | Methods |
|---|---|---|
| 428 | Precondition Required - server requires the request to be conditional, e.g. to make sure that the "lost update problem" is avoided (see **MAY** consider to support `Prefer` header to handle processing preferences). | `<all>` |
| 429 | Too many requests - the client does not consider rate limiting and sent too many requests (see **MUST** use code 429 with headers for rate limits). | `<all>` |

#### 5.2.2.4. Server side error codes:

| Code | Meaning | Methods |
|---|---|---|
| 500 | Internal Server Error - a generic error indication for an unexpected server execution problem (here, client retry may be sensible) | `<all>` |
| 501 | Not Implemented - server cannot fulfill the request (usually implies future availability, e.g. new feature). | `<all>` |
| 503 | Service Unavailable - service is (temporarily) not available (e.g. if a required component or downstream service is not available) — client retry may be sensible. If possible, the service should indicate how long the client should wait by setting the `Retry-After` header. | `<all>` |

### 5.2.3. MUST use most specific HTTP status codes

You must use the most specific HTTP status code when returning information about your request processing status or error situations.

### 5.2.4. MUST use code 207 for batch or bulk requests

Some APIs are required to provide either *batch* or *bulk* requests using `POST` for performance reasons, i.e. for communication and processing efficiency. In this case services may be in need to signal multiple response codes for each part of an batch or bulk request. As HTTP does not provide proper guidance for handling batch/bulk requests and responses, we herewith define the following approach:

- A batch or bulk request **always** responds with HTTP status code 207 unless a non-item-specific failure occurs.
- A batch or bulk request **may** return 4xx/5xx status codes, if the failure is non-item-specific and cannot be restricted to individual items of the batch or bulk request, e.g. in case of overload situations or general service failures.
- A batch or bulk response with status code 207 **always** returns as payload a multi-status response containing item specific status and/or monitoring information for each part of the batch or bulk request.

**Note:** These rules apply *even in the case* that processing of all individual parts *fail* or each part is

executed *asynchronously*!

The rules are intended to allow clients to act on batch and bulk responses in a consistent way by inspecting the individual results. We explicitly reject the option to apply 200 for a completely successful batch as proposed in Nakadi's `POST /event-types/{name}/events` as short cut without inspecting the result, as we want to avoid risks and expect clients to handle partial batch failures anyway.

The bulk or batch response may look as follows:

```
BatchOrBulkResponse:
  description: batch response object.
  type: object
  properties:
    items:
      type: array
      items:
        type: object
        properties:
          id:
            description: Identifier of batch or bulk request item.
            type: string
          status:
            description: >
              Response status value. A number or extensible enum describing
              the execution status of the batch or bulk request items.
            type: string
            x-extensible-enum: [...]
          description:
            description: >
              Human readable status description and containing additional
              context information about failures etc.
            type: string
        required: [id, status]
```

**Note**: while a *batch* defines a collection of requests triggering independent processes, a *bulk* defines a collection of independent resources created or updated together in one request. With respect to response processing this distinction normally does not matter.

### 5.2.5. MUST use code 429 with headers for rate limits

APIs that wish to manage the request rate of clients must use the 429 (Too Many Requests) response code, if the client exceeded the request rate (see RFC 6585). Such responses must also contain header information providing further details to the client. There are two approaches a service can take for header information:

- Return a `Retry-After` header indicating how long the client ought to wait before making a follow-up request. The Retry-After header can contain a HTTP date value to retry after or the number of seconds to delay. Either is acceptable but APIs should prefer to use a delay in

seconds.

- Return a trio of `X-RateLimit` headers. These headers (described below) allow a server to express a service level in the form of a number of allowing requests within a given window of time and when the window is reset.

The `X-RateLimit` headers are:

- `X-RateLimit-Limit`: The maximum number of requests that the client is allowed to make in this window.
- `X-RateLimit-Remaining`: The number of requests allowed in the current window.
- `X-RateLimit-Reset`: The relative time in seconds when the rate limit window will be reset. **Beware** that this is different to Github and Twitter's usage of a header with the same name which is using UTC epoch seconds instead.

The reason to allow both approaches is that APIs can have different needs. Retry-After is often sufficient for general load handling and request throttling scenarios and notably, does not strictly require the concept of a calling entity such as a tenant or named account. In turn this allows resource owners to minimise the amount of state they have to carry with respect to client requests. The 'X-RateLimit' headers are suitable for scenarios where clients are associated with pre-existing account or tenancy structures. 'X-RateLimit' headers are generally returned on every request and not just on a 429, which implies the service implementing the API is carrying sufficient state to track the number of requests made within a given window for each named entity.

### 5.2.6. MUST use problem JSON

RFC 7807 defines a Problem JSON object and the media type `application/problem+json`. Operations should return it (together with a suitable status code) when any problem occurred during processing and you can give more details than the status code itself can supply, whether it be caused by the client or the server (i.e. both for 4xx or 5xx error codes).

The Open API schema definition of the Problem JSON object can be found on github. You can reference it by using:

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

You may define custom problem types as extensions of the Problem JSON object if your API needs to return specific, additional and detailed error information.

Problem `type` identifiers in our APIs are not meant to be resolved. The RFC encourages that custom problem types are URI references that point to human-readable documentation, **but** we deliberately decided against that. URLs tend to be fragile and not very stable over a longer period.

Hosting documentation often requires to bind to a specific tool or have DNS records that contain volatile organization structures, e.g. team names. Another reason is that all the important parts of an API must be documented using OpenAPI anyway.

In order to stay compatible the proposed pattern for custom problem types is to use relative URI references:

- `/problems/out-of-stock`
- `/problems/insufficient-funds`
- `/problems/user-deactivated`

Examples of problem types that **do not** satisfy our criteria:

- `https://docs.team.company.org/out-of-stock`
- `https://en.wikipedia.org/wiki/Stockout`
- `http://www.businessdictionary.com/definition/stockout.html`

**Hint** for backward compatibility: A previous version of this guideline (before the publication of RFC 7807 and the registration of the media type) told to return custom variant of the media type `application/x.problem+json`. Servers for APIs defined before this change should pay attention to the `Accept` header sent by the client and set the `Content-Type` header of the problem response correspondingly. Clients of such APIs should accept both media types.

### 5.2.7. MUST not expose stack traces

Stack traces contain implementation details that are not part of an API, and on which clients should never rely. Moreover, stack traces can leak sensitive information that partners and third parties are not allowed to receive and may disclose insights about vulnerabilities to attackers.

# 6. Data formats

## 6.1. Data formats

### 6.1.1. MUST use JSON to encode structured data

Use JSON-encoded body payload for transferring structured data. The JSON payload must follow RFC 7159 using a JSON object as top-level data structure (if possible) to allow for future extension. This also applies to collection resources, where one naturally would assume an array. See also **(RFP) MUST** always return JSON objects as top-level data structures if JSON is being used.

Additionally, the JSON payload must comply to RFC 7493), particularly

- Section 2.1 on encoding of characters, and
- Section 2.3 on object constraints.

As a consequence, a JSON payload must

- use UTF-8 encoding
- consist of valid Unicode strings, i.e. must not contain non-characters or surrogates, and
- contain only unique member names (no duplicate names).

### 6.1.2. MAY use non JSON media types for binary data or alternative content representations

Other media types may be used in following cases:

- Transferring binary data or data whose structure is not relevant. This is the case if payload structure is not interpreted and consumed by clients as is. Example of such use case is downloading images in formats JPG, PNG, GIF.
- In addition to JSON version alternative data representations (e.g. in formats PDF, DOC, XML) may be made available through content negotiation.

#### 6.1.2.1. SHOULD encode embedded binary data in `base64url`

Exposing binary data using an alternative media type is generally preferred. See the rule above.

If an alternative content representation is not desired then binary data should be embedded into the JSON document as a `base64url`-encoded string property following RFC 7493 Section 4.4.

### 6.1.3. SHOULD prefer standard media type name `application/json`

Previously, this guideline allowed the use of custom media types like `application/x.pon.article+json`. This usage is not recommended anymore and should be avoided, except where it is necessary for cases of media type versioning. Instead, just use the standard media type name `application/json` (or `application/problem+json` for **MUST** use problem JSON).

Custom media types beginning with `x` bring no advantage compared to the standard media type for JSON, and make automated processing more difficult. They are also discouraged by RFC 6838.

### 6.1.4. SHOULD use standardized property formats

JSON Schema and Open API define several universally useful property formats. The following table contains some additional formats that are particularly useful in an e-commerce environment.

Please notice that the list is not exhaustive and everyone is encouraged to propose additions.

| type | format | Specification | Example |
|---|---|---|---|
| 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 |

| type | format | Specification | Example |
|---|---|---|---|
| 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" |
| string | email | RFC 5322 | "example@pon.com" |
| string | gtin-13 | GTIN | "5710798389878" |
| string | hostname | RFC 1034 | "www.pon.com" |
| 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.pon.com/" |
| string | uri-template | RFC 6570 | "/users/{id}" |
| string | uuid | RFC 4122 | "e2ab873e-b295-11e9-9c02-…" |

### 6.1.5. MUST use standard date and time formats

#### 6.1.5.1. JSON payload

Read more about date and time format in **SHOULD** define dates properties compliant with RFC 3339.

#### 6.1.5.2. HTTP headers

Http headers including the proprietary headers use the HTTP date format defined in RFC 7231.

### 6.1.6. SHOULD use standards for country, language and currency codes

Use the following standard formats for country, language and currency codes:

- ISO 3166-1-alpha2 country codes
  - (It is "GB", not "UK", even though "UK" has seen some use at Pon)
- ISO 639-1 language code
  - BCP 47 (based on ISO 639-1) for language variants

- ISO 4217 currency codes

### 6.1.7. MUST define format for number and integer types

Whenever an API defines a property of type `number` or `integer`, the precision must be defined by the format as follows to prevent clients from guessing the precision incorrectly, and thereby changing the value unintentionally:

| type | format | specified value range |
|---|---|---|
| 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 |

The precision must be translated by clients and servers into the most specific language types. E.g. for the following definitions the most specific language types in Java will translate to `BigDecimal` for `Money.amount` and `int` or `Integer` for the `OrderList.page_size`:

```
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
```

## 6.2. JSON guidelines

These guidelines provides recommendations for defining JSON data at Pon. JSON here refers to RFC 7159 (which updates RFC 4627), the "application/json" media type and custom JSON media types

defined for APIs. The guidelines clarifies some specific cases to allow Pon JSON data to have an idiomatic form across teams and services.

The first some of the following guidelines are about property names, the later ones about values.

### 6.2.1. MUST property names must be ASCII snake_case (and never camelCase): `^[a-z_][a-z_0-9]*$`

Property names are restricted to ASCII strings. The first character must be a letter, or an underscore, and subsequent characters can be a letter, an underscore, or a number.

(It is recommended to use `_` at the start of property names only for keywords like `_links`.)

Rationale: No established industry standard exists, but many popular Internet companies prefer snake_case: e.g. GitHub, Stack Exchange, Twitter. Others, like Google and Amazon, use both - but not only camelCase. It's essential to establish a consistent look and feel such that JSON looks as if it came from the same hand.

### 6.2.2. MUST declare enum values using UPPER_SNAKE_CASE format

Enum values (using `enum` or `x-extensible-enum`) need to consistently use the upper-snake case format, e.g. `VALUE` or `YET_ANOTHER_VALUE`. This approach allows to clearly distinguish values from properties or other elements.

### 6.2.3. SHOULD define maps using `additionalProperties`

A "map" here is a mapping from string keys to some other type. In JSON this is represented as an object, the key-value pairs being represented by property names and property values. In Open API schema (as well as in JSON schema) they should be represented using additionalProperties with a schema defining the value type. Such an object should normally have no other defined properties.

The map keys don't count as property names in the sense of rule 118, and can follow whatever format is natural for their domain. Please document this in the description of the map object's schema.

Here is an example for such a map definition (the `translations` property):

```
components:
  schemas:
    Message:
      description:
        A message together with translations in several languages.
      type: object
      properties:
        message_key:
          type: string
          description: The message key.
        translations:
          description:
```

```
            The translations of this message into several languages.
            The keys are [IETF BCP-47 language
tags](https://tools.ietf.org/html/bcp47).
          type: object
          additionalProperties:
            type: string
            description:
              the translation of this message into the language identified by the key.
```

An actual JSON object described by this might then look like this:

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

### 6.2.4. SHOULD pluralize array names

To indicate they contain multiple values prefer to pluralize array names. This implies that object names should in turn be singular.

### 6.2.5. MUST not use `null` for boolean properties

Schema based JSON properties that are by design booleans must not be presented as nulls. A boolean is essentially a closed enumeration of two values, true and false. If the content has a meaningful null value, strongly prefer to replace the boolean with enumeration of named values or statuses - for example accepted_terms_and_conditions with true or false can be replaced with terms_and_conditions with values yes, no and unknown.

### 6.2.6. MUST use same semantics for `null` and absent properties

Open API 3.x allows to mark properties as `required` and as `nullable` to specify whether properties may be absent (`{}`) or `null` (`{"example":null}`). If a property is defined to be not `required` and `nullable` (see 2nd row in Table below), this rule demands that both cases must be handled in the exact same manner by specification.

The following table shows all combinations and whether the examples are valid:

| required | nullable | {} | {"example":null} |
|---|---|---|---|
| `true` | `true` | ✗ No | ✓ Yes |
| `false` | `true` | ✓ Yes | ✓ Yes |

| required | nullable | {} | {"example":null} |
|---|---|---|---|
| `true` | `false` | ✗ No | ✗ No |
| `false` | `false` | ✓ Yes | ✗ No |

While API designers and implementers may be tempted to assign different semantics to both cases, we explicitly decide **against** that option, because we think that any gain in expressiveness is far outweighed by the risk of clients not understanding and implementing the subtle differences incorrectly.

As an example, an API that provides the ability for different users to coordinate on a time schedule, e.g. a meeting, may have a resource for options in which every user has to make a `choice`. The difference between *undecided* and *decided against any of the options* could be modeled as *absent* and `null` respectively. It would be safer to express the `null` case with a dedicated Null object, e.g. `{}` compared to `{"id":"42"}`.

Moreover, many major libraries have somewhere between little to no support for a `null`/absent pattern (see Gson, Moshi, Jackson, JSON-B). Especially strongly-typed languages suffer from this since a new composite type is required to express the third state. Nullable `Option`/`Optional`/`Maybe` types could be used but having nullable references of these types completely contradicts their purpose.

The only exception to this rule is JSON Merge Patch RFC 7396) which uses `null` to explicitly indicate property deletion while absent properties are ignored, i.e. not modified.

### 6.2.7. SHOULD not use `null` for empty arrays

Empty array values can unambiguously be represented as the empty list, `[]`.

### 6.2.8. SHOULD represent enumerations as strings

Strings are a reasonable target for values that are by design enumerations.

### 6.2.9. SHOULD name date/time properties with `_at` suffix

Dates and date-time properties should end with `_at` to distinguish them from boolean properties which otherwise would have very similar or even identical names:

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

**Note:** `created` and `modified` were mentioned in an earlier version of the guideline and are therefore still accepted for APIs that predate this rule.

### 6.2.10. SHOULD define dates properties compliant with RFC 3339

Use the date and time formats defined by RFC 3339:

- for "date" use strings matching `date-fullyear "-" date-month "-" date-mday`, for example: `2015-05-28`
- for "date-time" use strings matching `full-date "T" full-time`, for example `2015-05-28T14:07:17Z`

Note that the Open API format "date-time" corresponds to "date-time" in the RFC) and `2015-05-28` for a date (note that the Open API format "date" corresponds to "full-date" in the RFC). Both are specific profiles, a subset of the international standard ISO 8601.

A zone offset may be used (both, in request and responses) — this is simply defined by the standards. However, we encourage restricting dates to UTC and without offsets. For example `2015-05-28T14:07:17Z` rather than `2015-05-28T14:07:17+00:00`. From experience we have learned that zone offsets are not easy to understand and often not correctly handled. Note also that zone offsets are different from local times that might be including daylight saving time. Localization of dates should be done by the services that provide user interfaces, if required.

When it comes to storage, all dates should be consistently stored in UTC without a zone offset. Localization should be done locally by the services that provide user interfaces, if required.

Sometimes it can seem data is naturally represented using numerical timestamps, but this can introduce interpretation issues with precision, e.g. whether to represent a timestamp as 1460062925, 1460062925000 or 1460062925.000. Date strings, though more verbose and requiring more effort to parse, avoid this ambiguity.

# 7. Appendices

## Appendix G: Changelog

This change log only contains major changes made after XXXX 2020.

Non-major changes are editorial-only changes or minor changes of existing guidelines, e.g. adding new error code. Major changes are changes that come with additional obligations, or even change an existing guideline obligation. The latter changes are additionally labeled with "Rule Change" here.

To see a list of all changes, please have a look at the commit list in Github.

### 7.G.1. Rule Changes

- `2020-XX-XX:`

## 7.1. Bibliography

## Generic

- **Lessons-learned blog: Thoughts on RESTful API Design**
- [praeng] **Pragmatic engineer blog: Readable Code**
- [accidental-doppelganger] **https://www.informit.com/articles/article.aspx?p=1313447**

## Coding standards

- [nasa-safety-code] **The Power of Ten – Rules for Developing Safety Critical Code**
- [gnu-coding-standards] **GNU Coding Standards**
- [standardJs] **standardJS**
- [googleStyleguideJs] **Google Javascript styleguide**
- [googleStyleguideCpp] **Google CPP styleguide**
- [phpStandards] **PHP Standards Recommendations**

## Open API specification

- **Open API specification**
- **Open API specification mind map**

## Publications, specifications and standards

- **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

## Dissertations

- **Roy Thomas Fielding - Architectural Styles and the Design of Network-Based Software Architectures**: This is the text which defines what REST is.

## Books

- **REST in Practice: Hypermedia and Systems Architecture**
- **Build APIs You Won't Hate**
- **InfoQ eBook - Web APIs: From Start to Finish**
- [fowler-refactoring] **Refactoring, Improving the Design of Existing Code, 2nd edition**
- [pragmatic-programmer] **The pragmatic programmer, 20th anniversary edition**
- [building-secure-and-reliable-systems] *https://sre.google/static/pdf/building_secure_and_reliable_systems.pdf[Building secure & reliable systems]

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

<!-- Adds rule id as a postfix to all rule titles -->
<script>
var ruleIdRegEx = /(\d){3}/;

// Rules are either in h4 or in h5
['h4','h5'].forEach(tag => {
  var ruleheaders = document.getElementsByTagName(tag)
  for (var i = 0; i < ruleheaders.length; i++) {
    var header = ruleheaders[i];
    if (header.id && header.id.match(ruleIdRegEx)) {
      var a = header.getElementsByTagName("a")[0]
      a.innerHTML += " [" + header.id + "]";

      /**
       * If a rule is a standard it is appended with an "S"
       */
      if (header.id.includes('S')) {
        $(a).parent().parent().addClass('rule-standard');
      }
    }
  }
});
</script>
```

```
<!-- Add last modified to header -->
<script>
$(function() {
  $('#footer-text')
    .clone()
    .attr('id', 'last-modified')
    .attr( 'class', 'last-modified')
    .insertAfter('#_pon_integration_and_development_guidelines');
})
</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 = 'Integration / Development 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 headers = document.getElementsByTagName('h2');
for (var i = 0; i < headers.length; i++) {
```

```
    var a = headers[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);
    }

    var subUl = document.createElement('ul');
    var subHeaders = headers[i].nextElementSibling.getElementsByTagName("h3");
    for (var subI = 0; subI < subHeaders.length; subI++) {
        var a = subHeaders[subI].getElementsByTagName("a")[0];

        link = document.createElement('a');
        link.innerHTML = a.innerHTML;
        link.href = a.hash;
        subLi = document.createElement('li');
        subLi.append(link);
        subUl.append(subLi);
    }

    li.append(subUl);
}

document.body.classList.add('toc2');
document.body.classList.add('toc-left');
nav.append(ul);

//$('ul.sectlevel1').clone().insertAfter($('#toctitle'));
//$('#toc ul.sectlevel3').hide();

// Add the left menu bar
header.prepend(nav);
</script>

<!-- Cookies Consent -->
<link rel="stylesheet" type="text/css"
href="https://cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookieconsent.min.css"
/>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookieconsent.min.js"></s
cript>

<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] Per definition of R.Fielding REST APIs have to support HATEOAS (maturity level 3). Our guidelines do not strongly advocate for full REST compliance, but limited hypermedia usage, e.g. for pagination (see Hypermedia). However, we still use the term "RESTful API", due to the absence of an alternative established term and to keep it like the very majority of web service industry that also use the term for their REST approximations — in fact, in today's industry full HATEOAS compliant APIs are a very rare exception.

[2] HTTP/1.1 standard (RFC 7230) defines two types of headers: end-to-end and hop-by-hop headers. End-to-end headers must be transmitted to the ultimate recipient of a request or response. Hop-by-hop headers, on the contrary, are meaningful for a single connection only.