Pon Integration and Development Guidelines

Other formats: PDF, EPUB3

Table of Contents

1. Introduction

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 engineering principles. Microservices development begins with 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.

1.1. 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.

1.2. Pon specific information

The purpose of these guidelines is to define standards to successfully establish "consistent integration and development look and feel" quality. The integration guild drafted and owns this document. Teams are responsible to fulfill these guidelines during API development and are encouraged to contribute to guideline evolution via pull requests.

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

In case guidelines are changing, following rules apply:

- existing APIs 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- clients of existing APIs have to cope with these APIs based on outdated rules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 public externally available, it has to be re-reviewed and changed according to current guidelines - for sake of overall consistency.

2. Integration guidelines

2.1. Principles

2.1.1. 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

2.1.2. 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.

2.1.3. 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.

2.2. General guidelines

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

2.2.1. (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.

2.2.2. (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.

2.2.3. (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'`.

2.2.4. (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.

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

All resources, APIs, documentation, commments etc. must be written in U.S. English.

2.2.5.1. Audience

- **Development guild**
- **Integration guild**

2.3. Meta information

2.3.1. (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)

2.3.2. (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:

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

2.3.3. (RFP) MAY provide API identifiers

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

```yaml
/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:

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

2.3.4. (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:

```yaml
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

2.4. Security

2.4.1. (RFP) MUST secure endpoints

Every API endpoint should be secured, peferably using OAuth 2.0.

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

```yaml
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.

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

2.4.2. (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.

2.4.3. (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.

2.5. Compatibility

2.5.1. (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.

2.5.2. (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).

2.5.3. (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).

2.5.4. (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.

2.5.5. (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 [161] for an example.

Maps (see [216]), even though technically objects, are also forbidden as top level data structures, since they don't support compatible, future extensions.

{TODO} Add example

2.5.6. (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 [240] about enum value naming conventions.

2.5.7. (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.

2.5.8. {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.

2.5.9. (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.

2.6. 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.

2.6.1. (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 **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.

2.6.2. (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.

2.6.3. 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.

2.6.4. 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 [193].

2.6.5. 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 **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 **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.

2.6.6. 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 **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 **SHOULD** add `Deprecation` and `Sunset` header to responses).

2.6.7. MUST not start using deprecated APIs

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

3. Development guidelines

3.1. General development guidelines

> Progress is made by lazy men looking for easier ways to do things.
>
> — Robert A. Heinlein

3.1.1. Introduction

This chapter is about coding, developing or anything related to using a programming language to solve a problem. 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 previous 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 unlogical code will in itself be a source of new bugs.

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

3.1.2. 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

3.1.3. 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.

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.

Once and only once / don't repeat yourself (DRY)

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.

DRY code is good, but it is always a choice.

3.1.4. 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.

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

3.1.5. 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

3.1.6. 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.

3.2. File structure and naming

3.2.1. (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

3.2.2. (RFP) MUST filenames are either CamelCase or snake_case

Filenames

3.3. Version control

3.4. Testing code

3.5. Development environment

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

```
<!-- Adds rule id as a postfix to all rule titles -->
<script>
var ruleIdRegEx = /(\d)+/;
var ruleheaders = document.getElementsByTagName("h4");
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 + "]";
    }
}
</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 h2headers = document.getElementsByTagName('h3');
for (var i = 1; i < h2headers.length; i++) {
    var a = h2headers[i].getElementsByTagName("a")[0];
    if (a !== undefined) {
        link = document.createElement('a');
        link.innerHTML = a.innerHTML;
        link.href = a.hash;
        li = document.createElement('li');
        li.append(link);
        ul.append(li);
    }
}

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

<!-- Cookies Consent -->
<link rel="stylesheet" type="text/css"
href="https://cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookieconsent.min.cs
s" />
<script
src="https://cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookieconsent.min.js"
></script>
```

```
<script>
window.addEventListener("load", function(){
window.cookieconsent.initialise({
  "palette": {
    "popup": {
      "background": "#eaf7f7",
      "text": "#5c7291"
    },
    "button": {
      "background": "#56cbdb",
      "text": "#ffffff"
    }
  },
  "content": {
    "message": "This web site uses cookies to analyze the general behavior of
visitors."
  }
})});
</script>
```

[1] 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.