API Guidelines

Description	UKHSA API Guidelines
Author(s)	UKSHA
Repository	https://github.com/ukhsa-collaboration/api-guidelines/

Table of Contents

1 API Guidelines

1.1 Introduction

> This documentation supplements the API Strategy to provide detailed guidance on patterns and standards.

Standardising API design reduces friction, making APIs easier to understand, use, and maintain. APIs designed with consistent patterns are more intuitive and user-friendly with a common set of expectations that will enable better collaboration between teams.

These guidelines will ensure that all APIs follow accepted design, security and governance models, thereby raising the bar on API quality across the organisation.

1.1.1 When to use these guidelines

These guidelines follow the principles of Representational State Transfer (REST), using HTTP methods and stateless communication between client and server. The guidelines cover these use cases:

- **Internal APIs (Private APIs)**: Used to communicate between different internal systems, services or applications.
- **Public APIs (Open APIs)**: Openly accessible to external developers and users.
- **Partner APIs**: shared with specific external partners but are not openly available to the public. These APIs are typically part of a business agreement, allowing partners to integrate with internal systems or access shared services.

All the above APIs are expected to apply the same guidelines, patterns and standards.

If your product API is based on a different API technology, such as GraphQL or gRPC, this guidance may only partially apply. Further guidance may be provided in future depending on demand.

1.1.2 How to read the guidelines

The CAPITALIZED words throughout these guidelines have a special meaning:

```
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED",  "MAY", and "OPTIONAL" in
this document are to be interpreted as described in RFC2119.
```

Refer to RFC2119 for details.

How to use these guidelines

Each section addresses key aspects of building APIs, including naming conventions, versioning, security, error handling, and documentation.

Here's how to navigate and use these guidelines effectively:

Review the sections on API design, naming conventions, versioning and error handling and create an OpenAPI definition that adheres to these patterns. Determine your security requirements and apply the recommended authorisation, authentication and security patterns, such as OAuth 2.0, JWTs, and Role-Based Access Control (RBAC). Ensure your API is well-documented including error scenarios and example responses within the OpenAPI definition. Use the recommended tools for linting, validating and testing your OpenAPI definition and other aspects of your API.

1.2 API Specification & Documentation

The OpenAPI specification (OAS; formerly known as Swagger) is a widely adopted standard for describing REST APIs. It provides a machine-readable format for defining API endpoints, request/response schemas, and security configurations.

Using the OpenAPI Specification to create an OpenAPI definition is an essential design output when developing your API. The definition can be created in a simple text editor, integrated development environment (IDE), or using a dedicated tool such as Swagger Editor.

A sample OpenAPI definition based on these guidelines can be viewed here.

1.2.1 Generating code from OpenAPI definitions

The OpenAPI definition can be used to generate client and server code, including data transfer (DTOs) and service objects (implementation stubs). The OpenAPI Generator project supports a number of programming languages, frameworks and toolchains for this purpose.

Consider using these tools to accelerate development and testing.

1.2.2 Validating API requests against OpenAPI definitions

Tools exist for validating JSON REST content against OpenAPI definitions as part of testing. For example, the swagger-request-validator is an open source (Apache 2.0 licensed) validator.

1.2.3 Example responses & mock responses

Provide example responses in the OpenAPI definition. Examples will allow clients see what a small sample of the data returned by the API looks like, which will avoid ambiguity and accelerate development.

1.3 Organisational Data Models

APIs **SHOULD** be built and categorised according to the business capabilities they support.

This approach will align the UKHSA API portfolio with the core business functions and value streams of the organisation (e.g. surveillance, monitoring etc), making it easier for internal and external users to find and integrate with APIs that expose known business capabilities.

Organising APIs by business capability provides a logical structure that facilitates consistent governance and makes it simpler to identify gaps and opportunities for development within the organisation's API ecosystem.

APIs **SHOULD NOT** be designed around specific application use cases that are not aligned to capabilities and defined organisational data models.

The consistency of the names of entities and attributes across the whole application **MUST** be maintained.

1.4 API Design

1.4.1 API-First

SHOULD follow the API-first approach to designing the API, as it prioritises the API contract from the very beginning, leading to the development of APIs which are more consistent, standardised and reusable.

1.4.2 RESTful API Style

SHOULD use the HTTP REST API style.

REST APIs **MUST** be stateless by design.

1.4.3 REST Maturity Levels

The Richardson Maturity Model defines levels of maturity for RESTful APIs, ranging from basic usage of HTTP to fully REST-compliant APIs.

APIs **MUST** use at least `Maturity Level 2`.

At this level, APIs not only define resources with URIs but also correctly use `HTTP methods` to perform actions on these resources. The interaction follows `REST` principles more closely, with specific verbs (`GET`, `POST`, `PUT`, `DELETE`) representing different actions.

- Resources **MUST** be identified with distinct URIs
- Standard HTTP methods **MUST** be used correctly
- The API **MUST** use standard HTTP status codes for responses

APIs **MAY** use `Maturity Level 3` - Hyper Media Controls (HATEOAS).

1.4.4 HTTP Methods & Semantics

APIs **MUST** use the appropriate HTTP method to perform operations on resources:

- `GET`: Retrieve a resource or a collection of resources.
- `POST`: Create a new resource.
- `PUT`: Update an existing resource, typically replacing it.
- `PATCH`: Partially update an existing resource.
- `DELETE`: Remove a resource.

APIs **SHOULD** observe standard method semantics:

- Safe methods have no side affects (i.e. using the method does not alter data).
- Idempotent methods can be be executed multiple times with the same result as executing once.
- Cacheable methods indicate responses can be cached / stored for future reuse.

Method implementations must fulfill the following basic properties according to RFC9110 Section 9.2:

Method	Safe?	Is idempotent?	Is cacheable?
`GET`	Yes	Yes	Optional
`HEAD`	Yes	Yes	Optional
`POST`	No	No	No
`PUT`	No	Yes	No
`PATCH`	No	No	No
`DELETE`	No	Yes	No
`OPTIONS`	Yes	Yes	No
`TRACE`	Yes	Yes	No

1.4.5 Response Format

APIs **SHOULD** accept and return valid JSON as the standard default data interchange format.

APIs **MAY** use the json.api specification.

APIs **MAY** use standard representations defined in specifications such as FHIR UK Core where required but **SHOULD** use the JSON formats where they are defined.

APIs **SHOULD** return JSON objects as top-level data structures and not return JSON arrays at the top level.

1.4.6 Content Negotiation

The API **MUST** indicate the format of the response using the Content-Type header.

```
Content-Type: application/json
```

APIs **MAY** return additional representations, such as XML, if supported and requested by content negotiation via the Accept header.

Supported content types **MUST** be documented in the OpenAPI specification.

1.4.7 REST HTTP Response Codes

APIs **MUST** use standard HTTP response codes

Use standard HTTP status codes to indicate the result of the operation. For example:

- **200 OK**, for a successful `GET` or `PUT` request.
- **201 Created**, for a successful `POST` request that results in resource creation.
- **204 No Content**, for a successful `DELETE` request.
- **400 Bad Request**, for a request with invalid data.
- **404 Not Found**, if the resource does not exist.
- **500 Internal Server Error**, for server-side issues.

`201 Created` responses to `POST` methods **SHOULD** have a `Location` header identifying the location of the newly created resource according to RFC9110 Section 10.2.2.

1.5 Naming Conventions

1.5.1 URI Structure

APIs **MUST** follow the defined hierarchical structure:

```
https://azgw.api.ukhsa.gov.uk/namespace/product/v1/users/12345?
sort=startDate
\____/  \__________________/ \_______________________________/
\____________/
  |            |                            |                        |
scheme     authority                      path
parameters
```

- **Scheme**: **MUST** always be `https://`
- **Authority**: Will be determined by the APIM platform
- **Path**: Will consist of components:
- **Namespace**: Predefined business area or capability
- **Product**: The business product name
- **Version (v1)**: the major API version with 'v' prefix
- **Collection (users)**: the REST collection
- **Resource (12345)**: The REST resource identifier

Each resource **MUST** be uniquely identifiable by a Uniform Resource Identifier (URI).

APIs **MUST** use lowercase for the entire URI.

APIs **SHOULD** limit the level of nesting to avoid overly complex URIs. Typically, two to three levels are sufficient.

> **Note**
>
> **Environments**
> Domain Names for various environments can be found in the API Management Low Level Design.

Namespaces

Namespaces and product names **MUST** be based on the Business Capability Model.

> **Note**
>
> Where applications supports multiple business capabilities then namespaces and product names should be based on the `Leading` one in LeanIX.

For example:

```
https://azgw.api.ukhsa.gov.uk/prevent/vaccine-management/v1/..
```

1.5.2 Resource Names

APIs **MUST** use **lowercase plural nouns** to represent collections (e.g., /orders, /customers, /products) not verbs.

Use:

> **Tip**
>
> ```
> /product/v1/orders
> /product/v1/orders/{orderId}/cancel
> ```

Avoid:

Caution

```
/product/v1/order
/product/v1/cancelOrder
```

1.5.3 Path Segments

APIs **MUST** use **kebab-case** for path segments.

Use:

Tip

```
/product/v1/user-accounts
```

Avoid:

Caution

```
/product/v1/userAccounts
/product/v1/user_accounts
```

1.5.4 Parameter Names

APIs **MUST** use lower camel case for query parameter names.

Use:

Tip

```
/product/v1/users?maxResults=10&startIndex=20
```

Not:

Caution

```
/product/v1/users?max_results=10&start_index=20
```

Terminology

APIs **MUST** use consistent names for query parameters having the same function across different endpoints.

Example:

Caution

```
/product/v1/orders?limit=10&offset=20
/product/v1/users?maxResults=10&startIndex=20
```

1.5.5 Property Names

APIs **MUST** use lower camel case for properties.

Example:

Use:

Tip

```
{
    "customerId": "12345",
    "userId" : "54321"
}
```

Not:

Caution

```
{
    "customer_id": "12345",
    "user_id" : "54321"
}
```

1.5.6 Terminology

Use consistent terminology across the API and in documentation. For instance, if you use `customer` in one part of your API, don't switch to `client` in another API if they represent the same concept.

Example query string:

Caution

```
/product/v1/orders?customerId=123
/product/v1/users?clientId=123
```

Example request/response model:

Caution

```
# order
{
    "orderId": "12345",
    "customerId" : "54321"
    ...
}

# user
{
    "userId": "12345",
    "clientId" : "54321"
    ...
}
```

1.6 Security

1.6.1 Data Protection

All APIs **MUST** be exposed using `HTTPS`. This is required to protect credentials and data in transit and applies to all API integrations.

Tokens are sensitive data and **MUST** be kept secret when communicated and stored in client applications.

API inputs **MUST** be validated.

1.6.2 Authentication

Authentication establishes the identity of a resource owner - i.e. either an end user or an application in system-to-system use cases.

Authentication **MUST** be handled with each request by providing a token along with the request.

APIs **MUST NOT** use HTTP Basic Authentication.

SHOULD use JWT (JSON Web Tokens), passed in the Authorization header using the Bearer scheme to convey authentication data.

> **Note**
>
> **OpenAPI Definition**
>
> Refer to OpenAPI documentation for the bearer scheme when designing the API.

When using JWTs as Bearer tokens, they **MUST** be included in the Authorization header as follows:

```
Authorization: Bearer <Base64 URL Encoded JWT content>
```

OpenID Connect

APIs **SHOULD** use OpenID Connect (OIDC) as the identity layer on top of OAuth 2.0 when authentication of end users is required.

JWT Validation

JWTs **MUST** be signed based on the JSON Web Signature (JWS) standard.

> **Note**
>
> **JWT Validation** JWT validation is a policy configurable on the APIM Platform that will perform some validation. However, APIs **MUST** still validate the JWT as specified below.

The API **MUST** validate the JWT `signature`, `expiry time`, `issuer`, `audience`, `subject` and `claims` in order to determine whether to grant access.

- **Issuer** (`iss`): Verify that the token was issued by a trusted authority. Check the `iss` claim against your expected issuer.
- **Audience** (`aud`): Ensure that the token was issued for your specific API or service by checking the `aud` claim.
- **Expiration Time** (`exp`)/**Not Before Time** (`nbf`): Ensure that the token has not expired by checking the `exp` and `nbf` claims, which are Unix timestamps.
- **Subject** (`sub`): Check the `sub` claim to ensure the token belongs to the expected user.
- **Custom Claims**: If the token includes any custom claims (e.g., roles, permissions), verify them according to your application's logic.

MUST NOT put secret information inside the JWT token that uses the JWS standard.

JWT expiration for interactive end-user applications **SHOULD** be between 1 and 60 minutes.

> ⚠ **Warning**
>
> **Security Note**
>
> Review the OWASP API guidelines on Broken Authentication and ensure relevant guidance is followed.

1.6.3 Authorisation

OAuth 2.0 provides authorisation of a client application via an access token. In end user use cases, authorisation is delegated from the user, whereas in system-to-system use cases client are authorised on their own behalf.

Authorisation **MUST** be handled with each request by providing a token along with the request.

APIs **SHOULD** use OAuth 2.0 for authorisation. Using OAuth 2.0 will provide the greatest compatibility with API consumers as it is a widely adopted standard. The following authorisation use cases are supported:

Use Case		Grant Type	Extensions
End User (confidential client)	For interactive authorisation where the authentication of a user is required and the client secret can be kept confidential within the backend service.	Authorization Code Refresh Token (**MAY**)	PKCE (**SHOULD**)
End User (public client)	For interactive authorisation where the authentication of a user is required and the client secret CANNOT be kept confidential in the client application.	Authorization Code Refresh Token (**MAY**)	PKCE (**MUST**)
System-to-System	For non-interactive authorisation outside of the context of a user	Client Credentials	

APIs **SHOULD NOT** use the "Resource Owner Password Credentials Grant" or "Implicit Grant", which are considered legacy and have been deprecated from OAuth 2.1 as they are considered weak from a security standpoint.

> **Note**
>
> Refer to OpenAPI documentation for OAuth 2.0 when designing the API.

OAuth 2.0 Authorization Code Grant Type

- **SHOULD** use OAuth 2.0 `Authorization Code` grant type for interactive authorisation where the authentication of a user is required.
- **SHOULD** use `PKCE` extension for enhanced security with confidential clients (e.g. backend service). **MAY** use refresh tokens with `Authorization Code` grant type.
- **MUST** use `Authorization Code` grant + `PKCE` with non-confidential (public) clients (e.g. single page web or mobile applications). Note that mobile applications can be reverse engineered to extract client secrets.

Client Credentials Grant Type

- **SHOULD** use OAuth 2.0 `Client Credentials` grant type for non-interactive (machine-to-machine) authorisation outside of the context of a user.
- **SHOULD** define permissions using OAuth 2.0 scopes.
- **SHOULD NOT** use refresh tokens with `Client Credentials` grant type.

TODO:

Scopes and permissions using OAuth 2.0 scopes

> ⚠ **Warning**
>
> **Security Note** Review the OWASP guidelines on Broken Function Level Authorization and ensure relevant guidance is followed.

1.6.4 Access Control

Role-Based Access Control (RBAC)

TODO

> ⚠ **Warning**
>
> **Security Note** Review the OWASP guidelines on Broken Function Level Authorization and ensure relevant guidance is followed.

1.6.5 Rate Limiting

TODO

1.7 Error Handling

1.7.1 Problem Details

APIs **MUST** conform to the RFC-9457 standard "Problem Details for HTTP APIs" which defines a structured way for expressing error details in HTTP APIs.

APIs **MUST** use the appropriate content type `application/problem+json` or `application/problem+xml` (depending on the format bring returned) when returning the Problem Details response object.

Problem Details responses **MUST** be described in the APIs open api specification.

A Problem Detail response **MUST NOT** contain a program stack trace or server log for debugging purposes, instead consider an extended member such as `traceId` as described bellow.

You **MUST** include all the base Problem Details members: `status`, `title`, `detail`, `type` and `instance`.

Extended Details

As per the RFC-9457 you **MAY** extend the Problem Details object to include additional context/information that are specific to the problem type.

There are some common extension members such as `traceId`, `errors` and `code` which are useful and so you should consider including them.

Extension Member	Include	Detail
`traceId`	**MAY**	Can be used to find any distributed traces and logs for the current request.
`errors`	**MAY**	When you want to respresent multiple validation errors from a single request.

Extension Member	Include	Detail
`code`	**MAY**	An API specific error code aiding the provider team understand the error based on their own potential taxonomy or registry.

If an API extends their Problem Details object to include API specific error codes i.e. adding `code` as an extension, then the API specific error codes **MUST** be documented in the OpenAPI definition.

The MOT history API and NHS Spine Core API Framework provides a useful example for standardising API specific error codes. While UKHSA's context may differ, a similar approach can be adopted while still adhering to RFC-9457.

1.7.2 Common Problems Registry

RFC-9457 has the concept of a registry for common problems, given that the intended use for these API Design Guidelines is for an APIM Platform, a **single** shared registry of Problem Details **MAY** be created or an existing registry adopted (as long as there aren't multiple registries) for common responses.

This should prevent redefining the same common responses for each new API and encourage consistency which is especially helpful for consumers of multiple APIs on the platform.

An example Problem Details registry with usage examples can be found here and the corresponding OpenAPI components file here; This can be achieved in OpenAPI through the use of the `$ref` keyword and referencing a URL which contains the shared OpenAPI components.

`$ref` usage example

```yaml
paths:
  /namespace/product/v1/patients:
    get:
      responses:
        '200':
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Result'
                title: GetResultsListOk
          description: A JSON array containing results objects.
        '404':
          $ref:
'https://developer.ukhsa.gov.uk/openApi/common#/components/responses/NotFo
und'
        default:
          $ref:
'https://developer.ukhsa.gov.uk/openApi/common#/#/components/responses/Une
xpectedError'
```

```yaml
# https://developer.ukhsa.gov.uk/openApi/common contents
...
components:
  responses:
    UnexpectedError:
      description: An unexpected error occurred.
      content:
        application/problem+json:
          schema:
            $ref: '#/components/schemas/ProblemDetails'
          examples:
            unauthorized:
              $ref: '#/components/examples/unauthorized'
            forbidden:
              $ref: '#/components/examples/forbidden'
            not-found:
              $ref: '#/components/examples/not-found'
            server-error:
              $ref: '#/components/examples/server-error'
...
```

> **Note**
>
> Refer to RFC-9547 standard for additional information.

1.7.3 Example Responses

400 Bad Request - Single Error

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

{
  "type": "https://datatracker.ietf.org/doc/html/rfc9110#section-15.5.1",
  "status": 400,
  "title": "Bad Request",
  "detail": "Invalid rquest, 'nhsNumber' is required.",
  "instance": "POST /namespace/product/v1/patients",
  "traceId": "00-63d4af1807586b0d98901ae47944192d-9a8635facb90bf76-01"
}
```

400 Bad Request - Multiple Errors

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

{
  "type": "https://datatracker.ietf.org/doc/html/rfc9110#section-15.5.1",
  "status": 400,
  "title": "Bad Request",
  "detail": "Invalid rquest, see errors.",
  "errors": [{
    "detail": "'nhsNumber' is required.",
    "pointer": "#/nhsNUmber"
   },
  {
    "detail": "'firstName' is required.",
    "pointer": "#/firstName"
  }]
  "instance": "POST /namespace/product/v1/patients",
  "traceId": "00-63d4af1807586b0d98901ae47944192d-9a8635facb90bf76-01"
}
```

401 Unauthorized

```
HTTP/1.1 401 Unauthorized
Content-Type: application/problem+json

{
  "type": "https://datatracker.ietf.org/doc/html/rfc9110#section-15.5.2",
  "status": 401,
  "title": "Unauthorized",
  "detail": "Access token not set or invalid. The requested resource could
not be returned",
  "instance": "GET /namespace/product/v1/patients/12345",
  "traceId": "00-63d4af1807586b0d98901ae47944192d-9a8635facb90bf76-01"
}
```

403 Forbidden

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json

{
  "type": "https://datatracker.ietf.org/doc/html/rfc9110#section-15.5.4",
  "status": 403,
  "title": "Forbidden",
  "detail": "The resource could not be returned as the requestor is not
authorized",
  "instance": "GET /namespace/product/v1/patients/12345",
  "traceId": "00-63d4af1807586b0d98901ae47944192d-9a8635facb90bf76-01"
}
```

404 Not Found

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{
  "type": "https://datatracker.ietf.org/doc/html/rfc9110#section-15.5.5",
  "status": 404,
  "title": "Not Found",
  "detail": "The requested resource was not found",
  "instance": "GET /namespace/product/v1/patients/12345",
  "traceId": "00-63d4af1807586b0d98901ae47944192d-9a8635facb90bf76-01"
}
```

500 Internal Server Error

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/problem+json

{
  "type": "https://datatracker.ietf.org/doc/html/rfc9110#section-15.6.1",
  "status": 500,
  "title": "Internal Server Error",
  "detail": "The server encountered an unexpected error",
  "traceId": "00-63d4af1807586b0d98901ae47944192d-9a8635facb90bf76-01"
}
```

1.8 Versioning and Deprecation

1.8.1 URI versioning

MUST use URI (path-based) versioning.

API versions **MUST** start with v1.

The version number **MUST** be placed consistently at the base of the api path.

Version numbers **MUST NOT** be passed as parameters.

Use:

Tip

```
/product/v1/users
/product/v2/users
```

Avoid:

Caution

```
/product/users/v2
/product/users?v=1
```

1.8.2 Semantic versioning

MUST use semantic versioning:

```
version = {MAJOR}.{MINOR}.{PATCH}
```

- `MAJOR` version when you make incompatible API changes
- `MINOR` version when you add functionality in a backward compatible manner
- `PATCH` version when you make backward compatible bug fixes

For example: `1.0.1 (MAJOR = 1, MINOR = 0, PATCH = 1)`

The semantic version represents the build version of the application.

MUST use only `MAJOR` version in URIs, formatter as the simple numeric `MAJOR` versions, prefixed with 'v' (e.g. `v1`, `v2`):

Use:

Tip

```
/product/v1/users
```

Avoid:

Caution

```
/product/v1.0.1/users
```

`MINOR` and `PATCH` versions **MUST NOT** be added to the URI as they do not affect compatibility.

1.8.3 API root endpoint

There **SHOULD** be an endpoint to return version metadata (typically the APIs root `/` endpoint) that is also documented in the OpenAPI definition, not only will this provide useful API metadata but will help API consumers know they're looking at the right place instead of getting a `404` or random `500` error as is common in some APIs.

```
GET /namespace/product/v1

{
  "name": "Product API",
  "version": "1.0.1"
  "status": "LIVE"
  "releaseDate": "2024-09-17"
  "documentation":
"https://developer.ukhsa.gov.uk/namespace/product/v1/docs"
  "releaseNotes":
"https://developer.ukhsa.gov.uk/namespace/product/v1/releaseNotes"
}
```

1.8.4 Compatibility

MUST provide a new API `MAJOR` version number for changes that alter the API contract, such as changes to resource structure, new required parameters, or significant behavioural changes.

Non-breaking changes, such as adding optional fields, new endpoints, or improving performance **MUST NOT** increment the version number.

SHOULD maintain backwards compatibility where possible.

1.8.5 Deprecation

MUST deprecate old API versions and document API deprecation status

MUST document when older API versions will be deprecated and eventually retired in the OpenAPI definition.

1.8.6 Communication

SHOULD notify API consumers of upcoming changes.

1.9 Pagination, Filtering & Sorting

1.9.1 Pagination

Offset-based pagination

SHOULD use offset-based pagination for *smaller* result sets.

Offset-based pagination uses `limit` and `offset` query parameters to specify the number of items to return and the starting position in the dataset.

Query Parameters

- `limit`: The maximum number of items to return.
- `offset`: The number of items to skip before starting to collect the result set.

```
GET /product/v1/orders?offset=10&limit=10
```

SHOULD provide sensible default values for the limit and offset parameters when not provided.

SHOULD enforce a maximum limit to prevent clients from requesting excessively large pages that could degrade server performance.

Cursor-based pagination

SHOULD use cursor-based pagination for *larger* result sets or when the underlying dataset changes frequently.

Cursor-based pagination uses a "cursor" that points to a specific item in the dataset, typically a unique identifier, to determine where to start the next page of results. The cursor is passed as a query parameter, often encoded, and allows precise navigation through the dataset.

Query Parameters

- `cursor`: The pointer to the position in the dataset to start the next page.
- `limit`: The maximum number of items to return.

```
GET /product/v1/orders?cursor=eyJvcmRlcklkIjoxMjN9&limit=10
```

SHOULD provide sensible default values for the limit parameter when not provided.

SHOULD enforce a maximum limit to prevent clients from requesting excessively large pages that could degrade server performance.

Pagination metadata

SHOULD return pagination metadata for larger result sets

The body of responses containing lists of results **SHOULD** contain pagination metadata for larger results sets:

- `total`: Used to inform the client of the total number of available items. This is useful for calculating the total number of pages or determining how much data remains.
- `offset` or `cursor`: Information about the current result set, `offset`, or `cursor` position.
- `limit`: The maximum number of items returned.

```
{
  "results": [
    {"id": 101, "item": "Item 1"},
    {"id": 102, "item": "Item 2"}
    // ... more results ...
  ],
  "metadata": {
    "total": 100,
    "offset": 10,
    "limit": 10
  }
}
```

1.9.2 Filtering

SHOULD use the `GET` `HTTP` method and ensure the filter is safe, idempotent and cacheable.

SHOULD use query parameters.

Example

```
GET /product/v1/results?type=Lateral%20Flow%20Test&result=POSITIVE
```

```yaml
paths:
  /results:
    get:
      summary: List all test results
      description: List all test results.
      operationId: getResults
      tags:
        - results
      parameters:
        - in: query
          name: type
          required: false
          description: The type of test to filter by.
          schema:
            type: string
            pattern: '^(eq|ne|gt|lt|gte|lte|in|nin|like|ilike)?:?
([^:&]+)$'
            description: "RHS filter expression in format '{operation}:
{value}'."
            example: "eq:Lateral%20Flow%20Test"
          example: Lateral Flow Test
        - in: query
          name: result
          required: false
          description: The result type of test to filter by.
          schema:
            type: string
            x-extensible-enum:
              - POSITIVE
              - NEGATIVE
              - UNREADABLE
            example: POSITIVE
```

Expressions

SHOULD use Right-Hand Side (`RHS`) operators to filter on specific fields in a resource.

`RHS` operators allow more sophisticated filtering than simple equality checks. Use these operators by appending them to the field name with a colon.

Operators

Available RHS operators include:

Operator	Description
`eq`	Equal to (default if no operator specified).
`ne`	Not equal to.
`gt`	Greater than.
`gte`	Greater than or equal to.
`lt`	Less than.
`lte`	Less than or equal to.
`in`	Matches any value in a comma-separated list.
`nin`	Does not match any value in a comma-separated list.
`like`	Pattern matching with wildcards (`*`).
`ilike`	Case-insensitive pattern matching with wildcards (`*`).

Examples

```
GET /product/v1/results?
type=Lateral%20Flow%20Test&result=in:POSITIVE,NEGATIVE
```

```
GET /product/v1/results?nhsNumber=like:485777*
```

```
GET /products?category=electronics&price=gte:100
```

You can combine multiple filters using the same URL by separating them with ampersands (&).

```
GET /products?category=electronics&price=gte:100&price=lte:500
```

Alternative Example

If there is a particularly common query parameter you **SHOULD** consider providing a new operation where the search parameter is embedded in the path as a path variable, but only where it makes sense from an API design perspective and aligns with RESTful resource / nested resources.

```
GET /product/v1/patients/4857773456/results?type=Lateral%20Flow%20Test
```

```yaml
paths:
  /patients/{nhsNumber}/results:
    get:
      summary: List all test results for given nhs number.
      description: List all test results for given nhs number.
      operationId: getResultsForNhs
      tags:
        - results
      parameters:
        - in: path
          name: nhsNumber
          required: true
          schema:
            type: string
            pattern: '^\d{3}(?:-| )?\d{3}(?:-| )?\d{4}$'
            description: The nhs number of patient
            example: '4857773456'
        - in: query
          name: type
          required: false
          description: The type of test to filter by.
          schema:
            type: string
            example: Lateral Flow Test
        - in: query
          name: result
          required: false
          description: The result type of test to filter by.
          schema:
            type: string
            x-extensible-enum:
              - POSITIVE
              - NEGATIVE
              - UNREADABLE
            example: POSITIVE
```

1.9.3 Sorting

Default sort order **SHOULD** be considered as undefined and non-deterministic.

If a explicit sort order is desired, the query parameter `sort` **SHOULD** be used with the following general syntax: `{fieldName}|{asc|desc},{fieldName}|{asc|desc}`.

Example

```
GET /product/v1/results?sort=nhsNumber|asc,type|desc
```

```yaml
components:
  parameters:
    sortParam:
      in: query
      name: sort
      description: How to sort the results.
      schema:
        type: string
        pattern: ^[a-z]+(?:[A-Z][a-z]+)+\|(?:asc|desc)(?:,[a-z]+(?:[A-Z]
[a-z]+)*\|(?:asc|desc))*$
      examples:
        sortBySingleField:
          value: nhsNumber|asc
          summary: Sort by a single field
        sortByMultipleField:
          value: nhsNumber|asc,type|desc
          summary: Sort by multiple fields
```

1.10 Testing

1.10.1 Validation / Linting

MUST validate the OpenAPI definition against the OpenAPI Specification and the UKSHA spectral ruleset.

1.10.2 Unit Testing

MUST perform appropriate **Unit Testing** to verify the functionality of various components within the API implementation.

1.10.3 Integration Testing

SHOULD perform **Contract Testing** to ensure that the API implementation adheres to the OpenAPI definition.

> This is especially important given the OpenAPI definition is the blueprint for onboarding APIs onto the APIM Platform.

SHOULD perform **End-to-End Testing** to ensure that all the components of the API work seamlessly together as a complete system.

	Contract Testing	End-to-End Testing
Purpose	To ensure that services communicate correctly by validating API contracts.	To verify that all the components of the API work seamlessly together as a complete system.
When to use	When microservices or APIs are involved, especially in a distributed system.	When you need to test workflows which might involve multiple API calls, integrations, and the overall system functionality.

	Contract Testing	End-to-End Testing
How it's done	By verifying the API implementation is communicating as defined in the APIs OpenAPI definition.	By simulating real user scenarios and executing tests through the application's UI or API.
What it reveals	Issues related to API compatibility and service interactions.	Overall application behaviour, including performance and user experiences.

1.10.4 Performance Testing

- **SHOULD** perform **Load Testing** to ensure the API can handle expected levels of load by simulating normal to high levels traffic to your API.
- **SHOULD** perform **Stress Testing** to understand the behaviour the API under extreme conditions and identify breaking points by simulating extreme levels of traffic to your API.

	Load Testing	Stress testing
Purpose	Ensures system can handle normal traffic and data volume	Determines system's breaking point and recovery
When to use	Before release or major updates	Before high-stress events or periodically
How it's done	Simulate normal to high levels of traffic	Simulate extreme levels of traffic
What it reveals	System lag, slow performing endpoints, or crashes	How the system fails or scales under extreme conditions

1.10.5 Security Testing

SHOULD conduct **Vulnerability Scanning** To identify known vulnerabilities in your API.

SHOULD conduct **Penetration Testing** manual and or automated, to identify security weaknesses in your API.

	Vulnerability Scanning	Penetration Testing
Purpose	To identify known vulnerabilities in an API or software application.	To simulate an attack on the API to exploit vulnerabilities and assess security measures.
When to use	During the development lifecycle or regularly to ensure ongoing security.	After significant changes to the API or before a major release to evaluate security posture.
How it's done	By using automated tools to scan code, configuration, and network settings for vulnerabilities.	Reconnaissance, vulnerability scanning, attempting to exploit vulnerabilities, and finally providing a detailed report. Typically performed by a security specialist/professional.
What it reveals	A list of known vulnerabilities, misconfigurations, and weaknesses in the API.	Specific vulnerabilities that can be exploited, their impact, and recommendations for remediation.

1.10.6 Recommended Testing Tools

Tool	Useful For	Open Source Licence
Spectral CLI	Validating your OpenAPI definitions against the OpenAPI Specification and the UKSHA spectral ruleset.	Apache 2.0
Prism	API Mock Servers from OpenAPI definition Contract Testing for API consumers and developers.	Apache 2.0
Pact	Consumer-Driven Contract Testing to ensure that your API meets the expectations of its consumers.	MIT
Zed Attack Proxy (ZAP)	Web application vulnerability scanner.	Apache 2.0

There is also a catalog of OpenAPI Tooling to support API development and validation.

1.11 Performance, Reliability & Monitoring

1.11.1 Performance, Reliability, and Monitoring

Overview

This section provides guidance on designing APIs for optimal performance, reliability, and observability. It includes best practices, patterns, and tools to ensure APIs meet high standards of quality and resilience.

1.11.2 Caching

Caching plays a crucial role in API performance, reducing latency and server load while improving response times.

APIs **SHOULD** leverage appropriate caching mechanisms.

APIs **SHOULD** use a multi-layered caching strategy i.e. implement caching at various layers (e.g., API Gateway / CDN, application layer, database/persistence layer) to optimise performance.

APIs **SHOULD** use distributed caching

Rule of Thumb

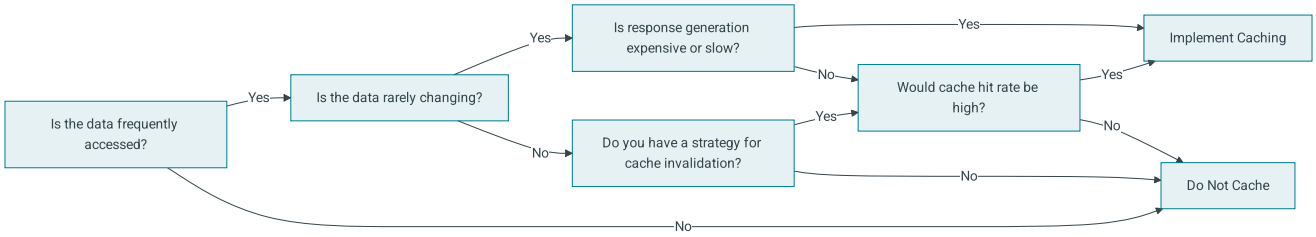
APIs **SHOULD** implement caching when:

- Serving frequently accessed, rarely changing data.
- Response generation is computationally expensive.
- Response times are slow or Requests are timing out.
- Handling high volumes of traffic.
- Serving static reference data or lookup tables.

APIs **SHOULD NOT** implement caching when:

- Serving volatile data that chances frequently, *unless* you have a strategy for cache invalidation.

Use the following flowchart to help you decide:



Server-Side vs Client-Side Caching

APIs **SHOULD** implement server-side response caching and **SHOULD** avoid implementing client-side caching except where there are specific requirements that necessitate it. This approach provides greater control, security, and consistency.

The HTTP caching specification RFC9111 is complex and often inconsistently implemented by clients, RESTful APIs frequently have nuanced caching requirements which may include handling data with mixed sensitivity levels that require careful cache management which generic client implementations might not handle correctly, furthermore client-side caching can lead to sensitive data being stored in *uncontrolled* environments.

Server-side caching reduces the risk of cached data leakage across different client contexts, ensures that caching policies are correctly applied regardless of client capabilities and that all clients receive consistent data.

In addition server-side caching provides greater control over what data is cached and for how long, offers fine-grained cache control which can be implemented based on resource type, authorisation level, or other factors and enables the ability to invalidate cached content when underlying data changes.

As such APIs **SHOULD** follow the best practice of defaulting the `Cache-Control` response header to the following value to prevent any default client caching behaviors.

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

Whilst this may seem inefficient, there are actually a good reasons for doing so, Preventing any accidental client caching of sensitive/secure data, meaning it's much safer to default to a posture of no client caching by default.

Server-Side Response Caching

APIs **SHOULD** implement server-controlled response caching that is independent of client-specified caching headers.

APIs **SHOULD** utilise their respective development ecosystem and take advantage of the available caching tools/libraries to support server-side response caching, for example if you are building your API with dotnet there is an output caching middleware specifically for sever controlled caching, and for python there is a framework agnostic caching library called cachews.

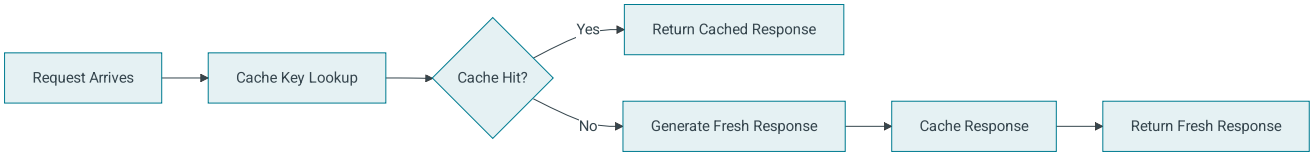
When utilising an API gateway, APIs **SHOULD** make use of any response caching functionality, as this helps to reduces the load on the backend API; Azure Api Management (APIM) provides this functionality through the use of policies.

Implementation Approaches

Basic Implementation Pattern

The general pattern for implementing server-side response caching is:

1. Check if request can be served from cache.
2. If cached, return cached response.
3. If not cached, generate response and store in cache.
4. Return fresh response.



Client-Side Caching

Client-side caching **SHOULD** be avoided. However in addition to server-side response caching, there are cases where client-side caching **MAY** be appropriate:

1. When offline capability is required (e.g., mobile applications)
2. For static resources that rarely change (e.g., images, stylesheets)
3. To reduce network traffic in bandwidth-constrained environments

If your API *really* requires supporting `HTTP caching`, please observe the following rules:

MAY responsibly enable `HTTP caching` explicitly for any operations that require it.

MUST document all cacheable `GET`, `HEAD`, and `POST` endpoints by declaring the support of `Cache-Control`, `Vary`, and `ETag` headers in response.

MUST NOT define the `Expires` header to prevent redundant and ambiguous definition of cache lifetime.

MUST 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.

APIs **SHOULD** use appropriate Cache-Control directives:

Directive	Purpose	Example
`max-age`	How long the response can be cached (in seconds)	`Cache-Control: max-age=3600`
`no-cache`	Must revalidate before using cached content	`Cache-Control: no-cache`
`no-store`	Don't cache the response at all	`Cache-Control: no-store`
`private`	Only browser can cache, not intermediaries	`Cache-Control: private, max-age=600`
`public`	Response can be cached by any cache	`Cache-Control: public, max-age=86400`

Operations which require the use of the `Authorization` Header i.e. OAuth protected endpoints, **SHOULD** also contain the `private` directive.

```
Cache-Control: private, must-revalidate, max-age=60
```

Example Implementations

Read-Only Reference Data

```
Cache-Control: public, max-age=86400
```

User-Specific Data (Non-Sensitive)

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

Time-Sensitive Data

```
Cache-Control: public, max-age=60
```

Sensitive Data

```
Cache-Control: no-store
```

Considerations for Special Cases

Pagination

Paginated responses **SHOULD** use cache control headers that decrease in duration for later pages:

```
# First page might be cached longer
Cache-Control: public, max-age=3600

# Later pages cached for shorter periods
Cache-Control: public, max-age=600
```

Search Results

Search endpoints **MAY** cache results for popular queries but **SHOULD** use shorter cache durations:

```
Cache-Control: public, max-age=300
```

Versioned APIs

Versioned API endpoints **MAY** use longer cache durations since their responses are stable by definition:

```
Cache-Control: public, max-age=604800
```

Multi-Layered Caching Strategy

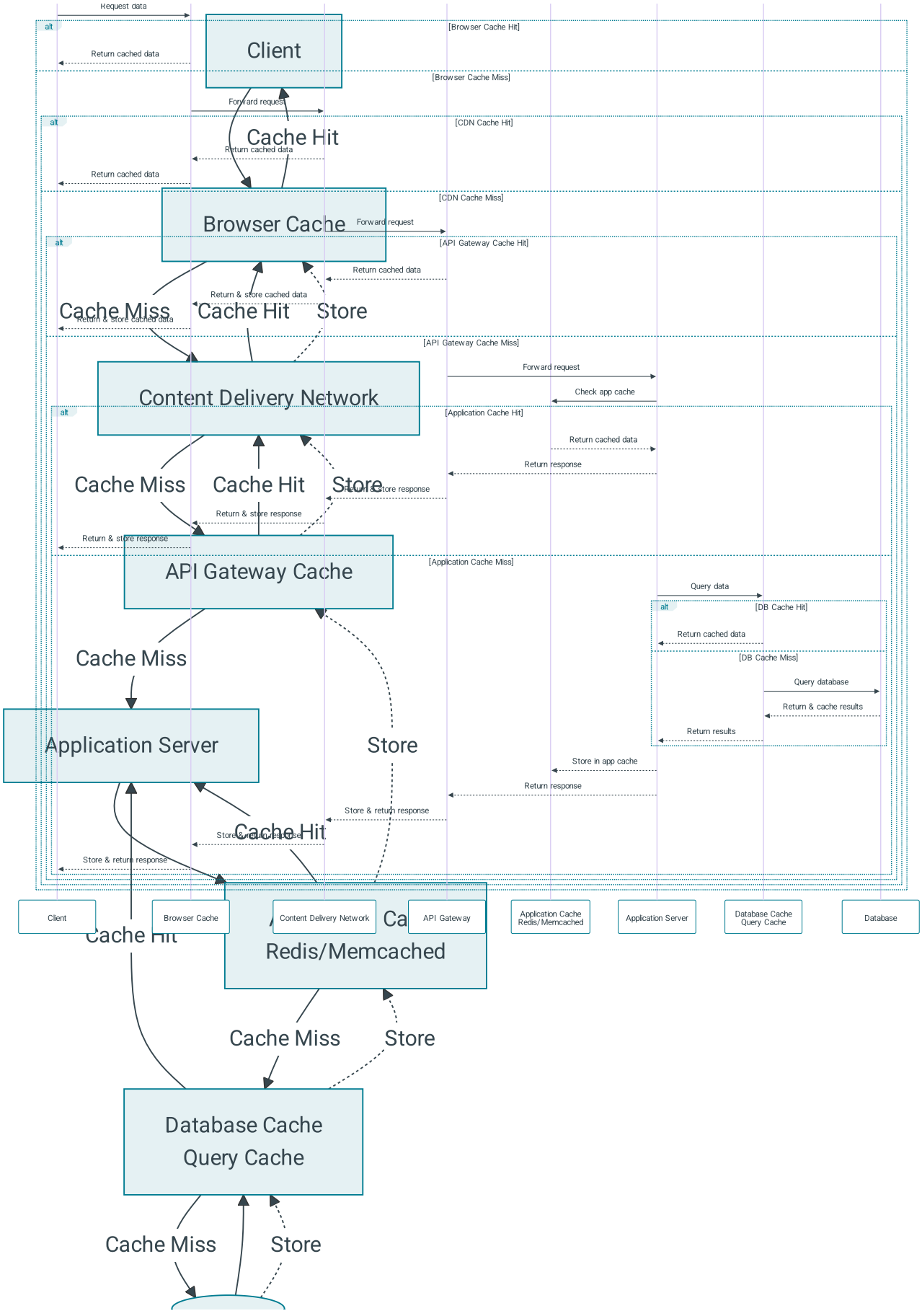
APIs **SHOULD** take a multi-layered approach to implement caching.

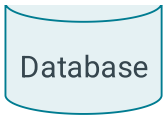
The diagrams below illustrates an example of multi-layered caching that provides performance benefits at different levels of your architecture.

> **Note**
>
> This is a generalised example and some of these layers might not be applicable for your application.

Flowchart Diagram

Sequence Diagram


Request data
alt
[Browser Cache Hit]
Client
Return cached data
[Browser Cache Miss]
Forward request
alt
[CDN Cache Hit]
Cache Hit
Return cached data
Return cached data
[CDN Cache Miss]
Browser Cache
Forward request
alt
[API Gateway Cache Hit]
Return cached data
Return & store cached data
Cache Miss
Cache Hit
Store
Return & store cached data
[API Gateway Cache Miss]
Forward request
Content Delivery Network
Check app cache
alt
[Application Cache Hit]
Return cached data
Return response
Cache Miss
Cache Hit
Store
Return & store response
Return & store response
Return & store response
API Gateway Cache
[Application Cache Miss]
Query data
alt
[DB Cache Hit]
Return cached data
[DB Cache Miss]
Query database
Return & cache results
Return results
Cache Miss
Application Server
Store
Store in app cache
Return response
Store & return response
Cache Hit
Store & return response
Store & return response
Client
Browser Cache
Content Delivery Network
API Gateway
Application Cache
Redis/Memcached
Application Server
Database Cache
Query Cache
Database
Cache Hit
Redis/Memcached
Cache Miss
Store
Database Cache
Query Cache
Cache Miss
Store



Benefits of Each Caching Layer

Browser Cache

- **Eliminates network requests** completely for repeat visits.
- **Instant response times** for cached resources.
- **Reduces bandwidth consumption** for the end user.

> ⚠ **Warning**
>
> **SHOULD** only be used for non-sensitive, static resources.
>
> **MUST NOT** be relied upon for critical application functionality.

Content Delivery Network (CDN)

- **Geographical distribution** reduces latency by serving from edge locations.
- **Massive scalability** to handle traffic spikes.
- **Offloads traffic** from origin servers.
- **Ideal for** static assets, public API responses, and infrequently changing data.

API Gateway Cache

- **Centralised caching** for all API endpoints.
- **Consistent policy enforcement** across all services.
- **Reduces load** on backend application servers.
- **Enables analytics** on cache performance at the API level.

Application Cache (Redis/Memcached)

- **High-speed data access** for frequently used data.
- **Flexible invalidation** based on application-specific logic.

- **Supports complex data structures** beyond simple key-value pairs.
- **Can be shared** across multiple application instances.

Database Cache/Query Cache

- **Optimises repeated queries** without application changes.
- **Reduces database load** for read-heavy workloads.
- **Often built into** database systems (e.g., MySQL Query Cache, PostgreSQL).
- **Transparent to the application** in many cases.

Multi-Layer Advantages

Using this approach, APIs can achieve significantly better performance and scalability while reducing infrastructure costs.

- **Defence in depth**: Even if one cache fails, others may still provide performance benefits.
- **Optimised resource usage**: Most expensive operations (i.e. database queries) are cached at multiple levels.
- **Improved resilience**: Distributed caching improves availability and fault tolerance.
- **Targeted optimisation**: Each layer can be optimised for specific types of data and access pattern.

Cache Invalidation Strategies

APIs **MUST** implement appropriate cache invalidation strategies to ensure data consistency while maintaining performance benefits. Effective cache invalidation is critical to prevent serving stale or incorrect data to clients.

Types of Cache Invalidation Strategies

Time-Based Expiration

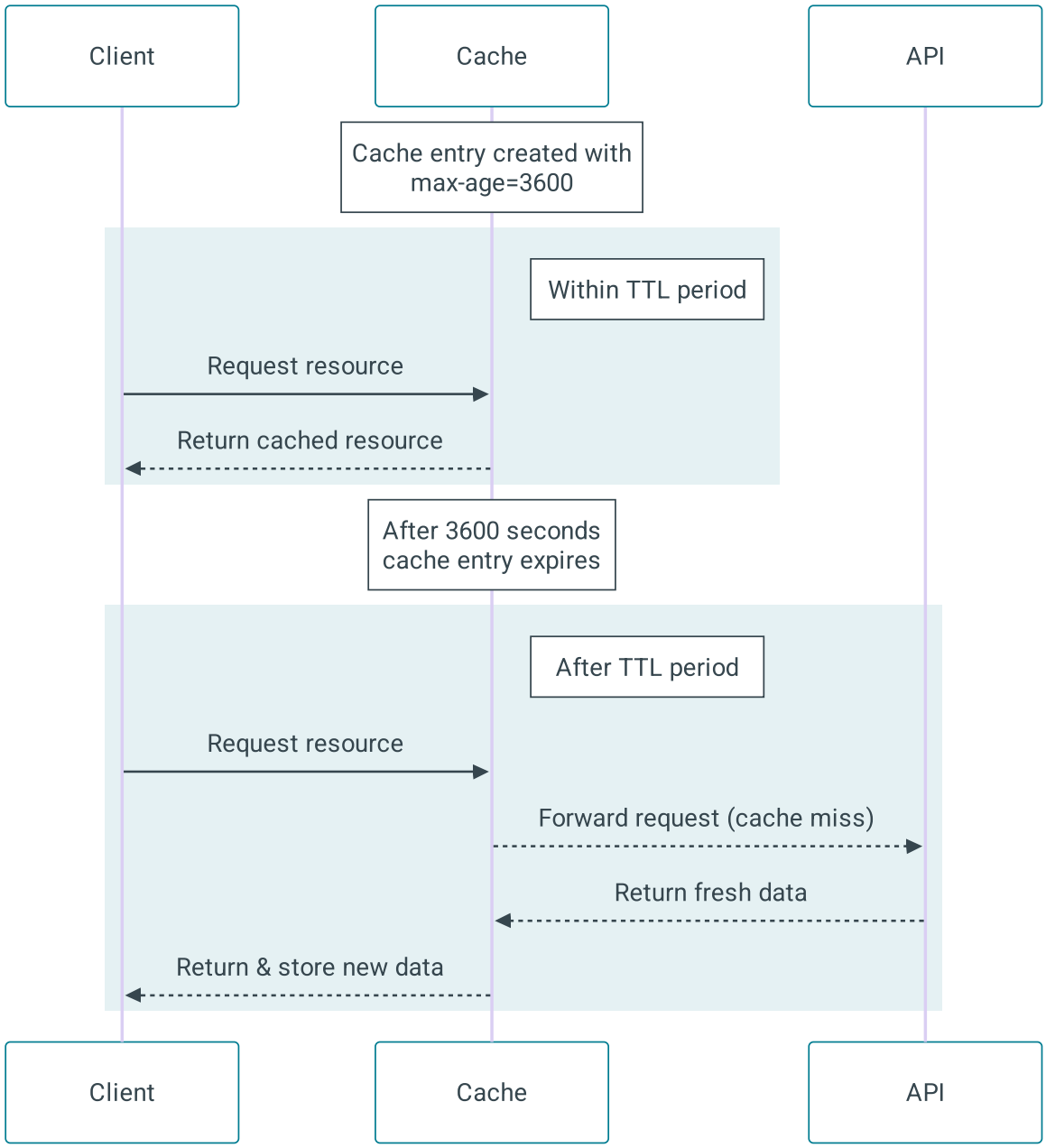
APIs **MUST** implement time-based expiration for all cacheable resources:

MUST set appropriate `max-age` values based on data volatility.

SHOULD use shorter expiration times for frequently changing data.

MAY use longer expiration times for static reference data.

MUST consider the business impact of serving stale data when setting expiration times.



Event-Based Invalidation

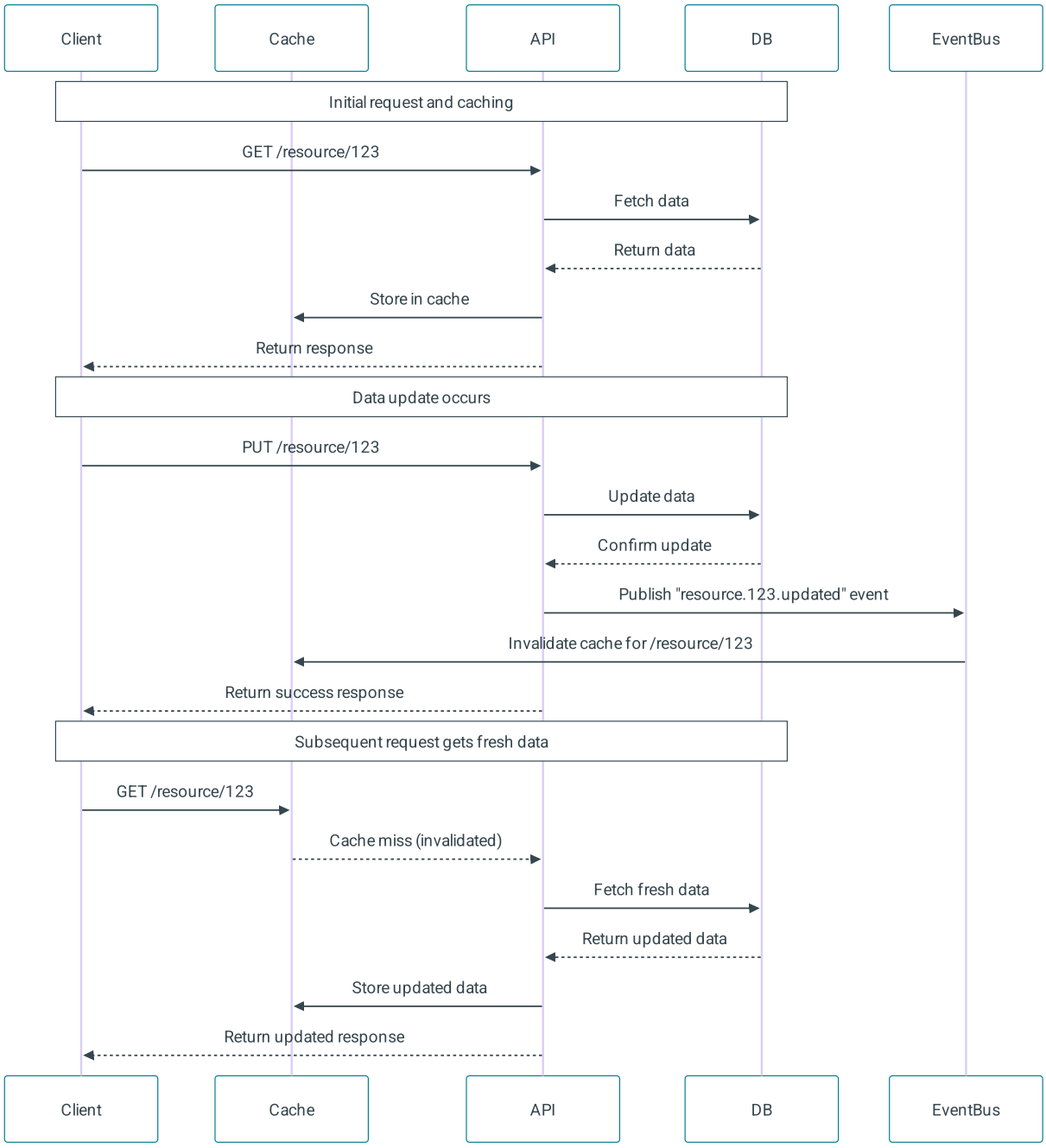
For data that changes unpredictably, APIs **SHOULD** implement event-based invalidation:

MUST trigger cache invalidation when the underlying data changes.

SHOULD use publish/subscribe mechanisms to notify cache systems of changes.

SHOULD implement targeted invalidation for specific resources rather than flushing entire caches.

MAY use message queues or webhooks for distributed cache invalidation.



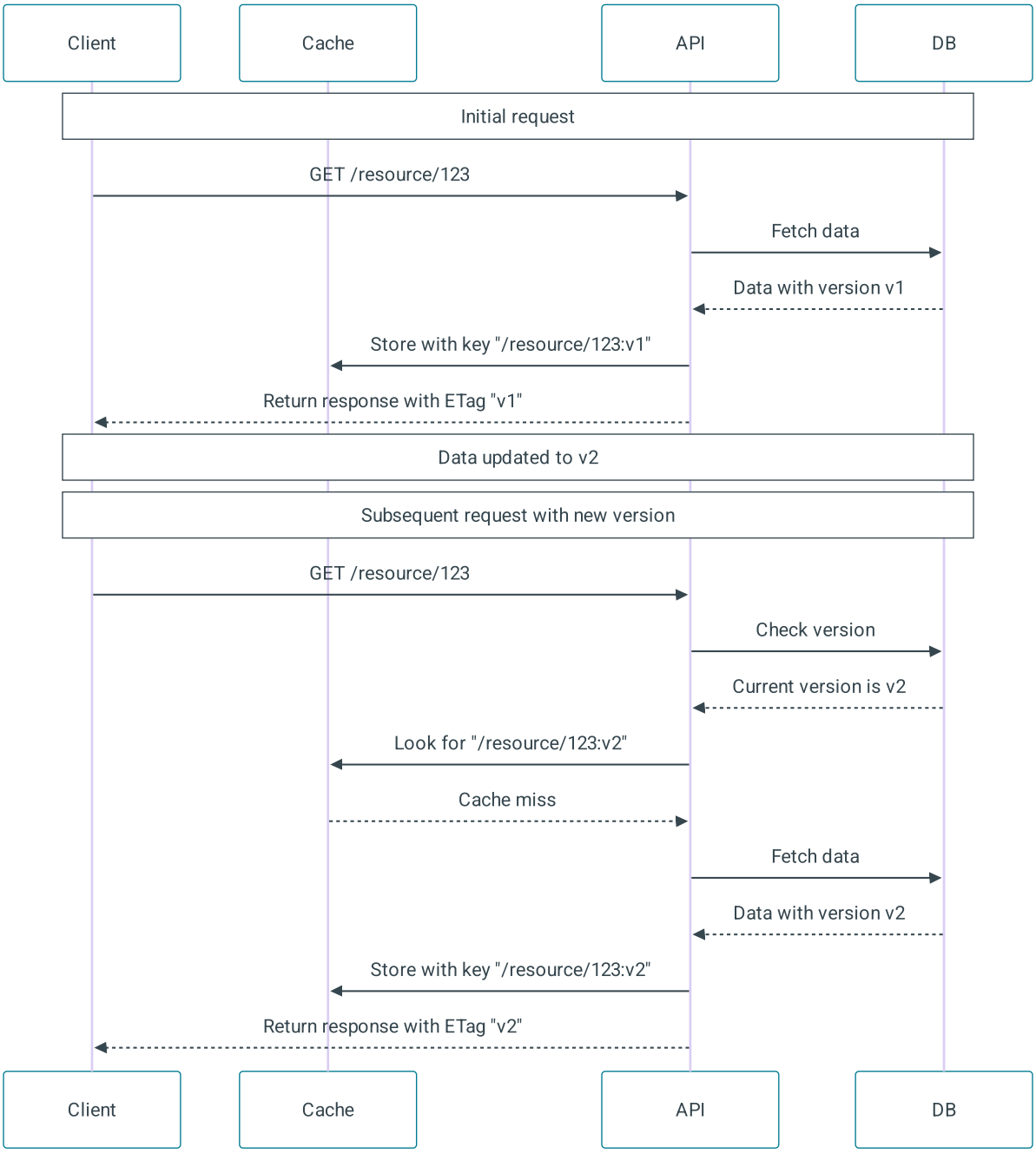
Resource Versioning

APIs **SHOULD** consider resource versioning as a complementary strategy:

MAY include version identifiers (e.g., `ETags`, timestamps) in cache keys.

MAY use content-based hashing for automatic versioning of static resources.

SHOULD NOT rely on versioning alone for frequently updated resources.



When to Use Each Strategy

Strategy	When to Use	When to Avoid
Time-Based Expiration	**MUST** use for all cacheable resources as a baseline strategy	**SHOULD NOT** rely solely on for critical, frequently changing data
Event-Based Invalidation	**SHOULD** use for dynamic data with unpredictable update patterns	**SHOULD NOT** use if update events cannot be reliably captured or propagated
Resource Versioning	**SHOULD** use for static assets and rarely changing resources	**SHOULD NOT** use as the only strategy for frequently updated resources

Hybrid Approaches

APIs **SHOULD** implement hybrid invalidation approaches for optimal results:

Time-Based + Event-Based

SHOULD set reasonable TTLs as a fallback.

MUST trigger invalidation on data changes.

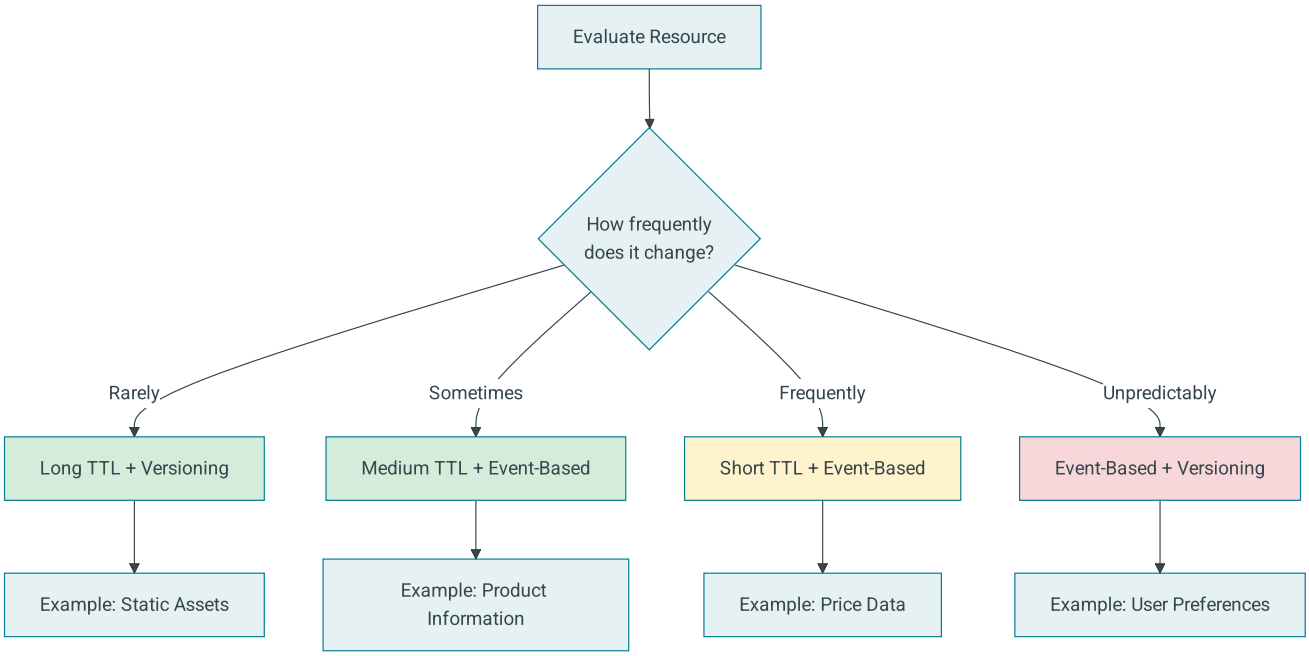
MUST ensure cache consistency in distributed environments.

Versioning + Time-Based

MAY version resources for major changes.

SHOULD set appropriate TTLs for minor variations.

SHOULD use conditional requests with `ETags`.



Cache Key Strategies

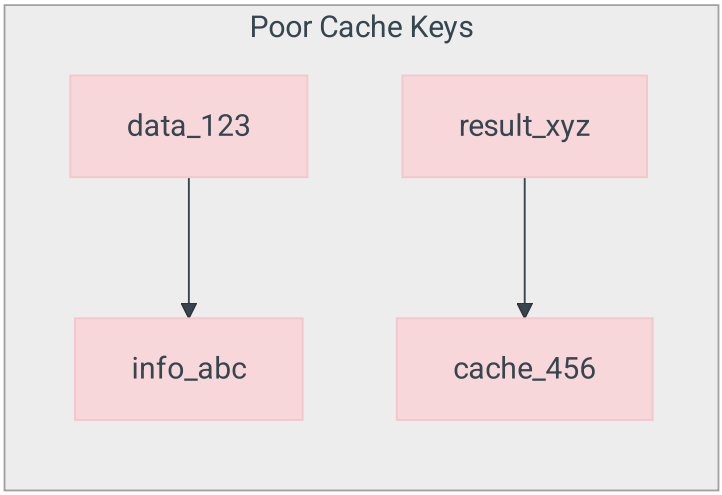
APIs **SHOULD** carefully design cache keys to support effective invalidation:

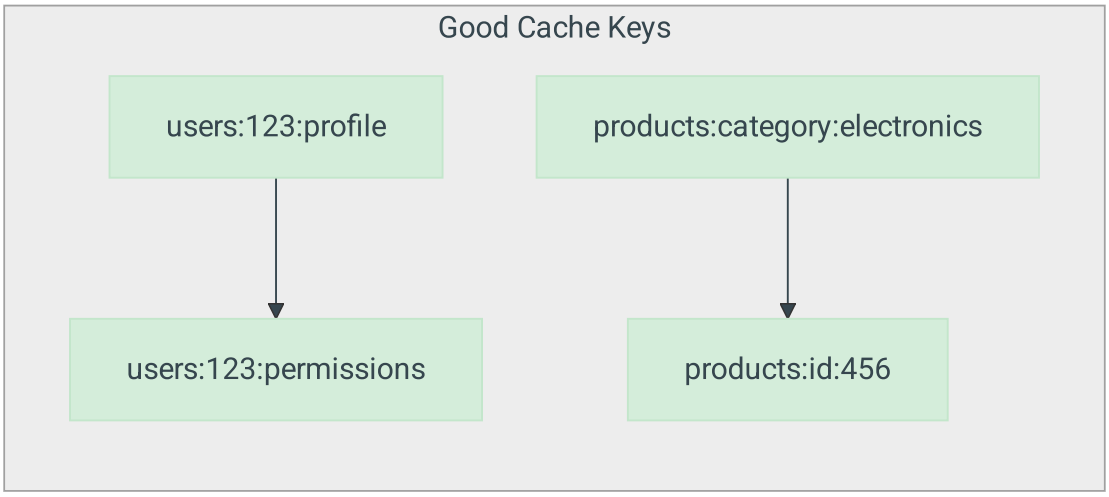
SHOULD use hierarchical keys to enable invalidation of related resources.

MAY include relevant parameters in cache keys (e.g., user roles for permission-dependent content)

MUST avoid including sensitive information in cache keys.

SHOULD document cache key formats to aid debugging and maintenance.





Performance Metrics

APIs using caching **SHOULD** monitor:

- **Cache Hit Rate**: Percentage of requests served from cache.
- **Cache Latency**: Time to retrieve data from cache.
- **Origin Latency**: Time to retrieve data from the origin.
- **Cache Size**: Memory/storage consumption by the cache.

APIs **SHOULD** aim for a cache hit rate of at least 80% for cacheable resources.

Response Headers for Monitoring

APIs **MAY** consider adding headers to help with debugging and monitoring:

```
X-Cache: HIT
X-Cache-TTL-Remaining: 286
X-Cache-Key: products:fec65fb3-1e5e-4ff2-a6e0-a423f77f0000
```

```
X-Cache: HIT
X-Cache-TTL-Remaining: 286
X-Cache-Key: products:list:limit=10:offset=0:sort=name|asc
```

Example metrics capture

The below pseudo python example shows how you could manually log caching statistics, however there might libraries that could collect this telemetry for you with OpenTelemetry instrumentation such as opentelemetry-instrumentation-fastapi.

```python
# Python example of cache monitoring
def get_cached_response(cache_key):
    start_time = time.time()
    cached_response = cache.get(cache_key)
    lookup_time = time.time() - start_time

    metrics.timing('cache.lookup_time', lookup_time)

    if cached_response:
        metrics.increment('cache.hit')
        return cached_response
    else:
        metrics.increment('cache.miss')
        return None
```

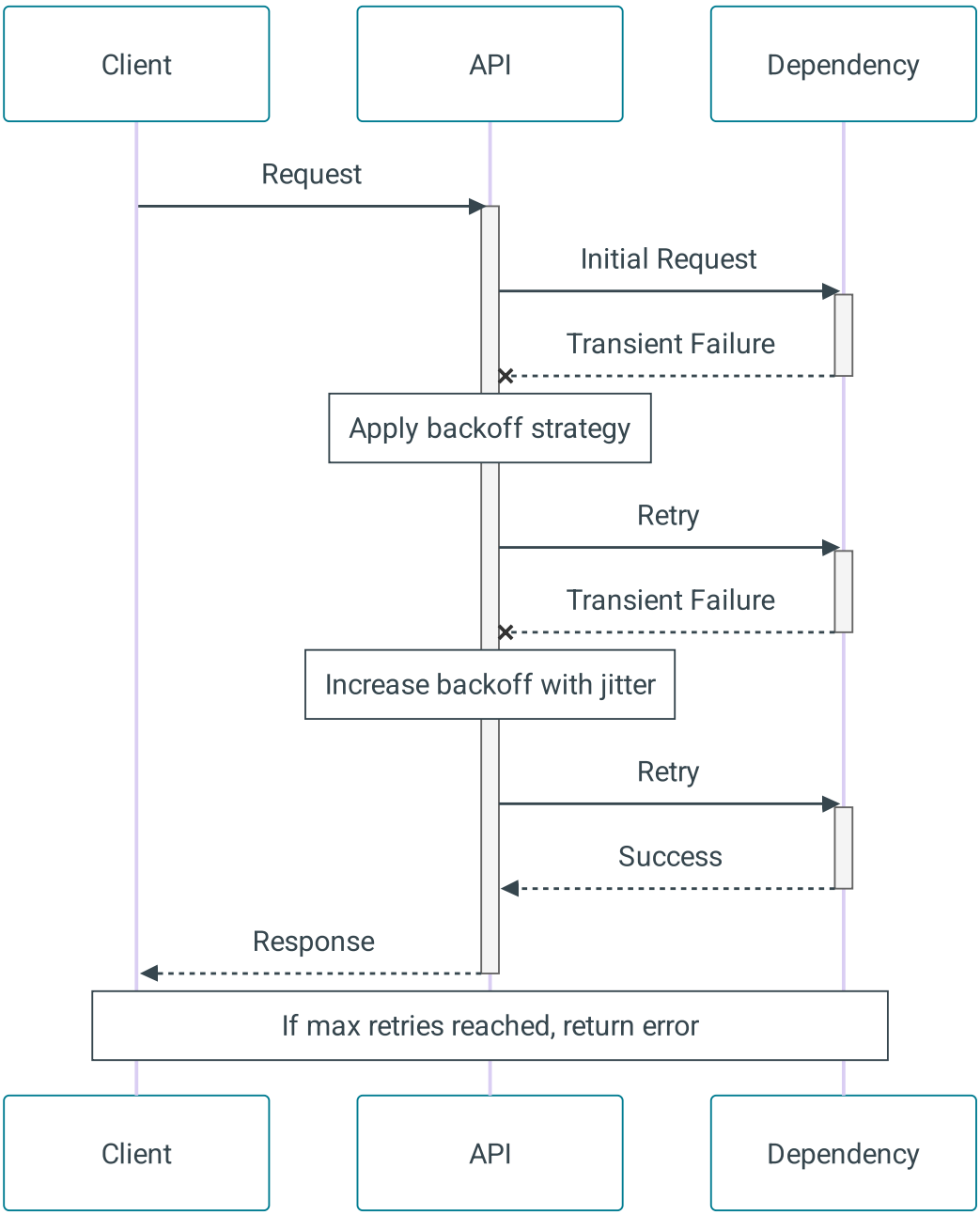
1.11.3 Resilience Patterns

Resilience patterns are essential for building robust APIs that can gracefully handle unexpected failures or delays in dependent systems. This section provides guidelines for implementing retries, timeouts, circuit breakers, bulkheads, and fallbacks in API design. These patterns **SHOULD** be applied where appropriate to ensure reliability, scalability, and user experience.

Retries

Retries allow APIs to recover from transient failures.

- APIs **SHOULD** implement retries for idempotent operations (e.g., `GET`, `PUT`, `DELETE`) where a transient failure is likely to succeed on subsequent attempts.
- Retries **SHOULD NOT** be used for non-idempotent operations (e.g., `POST`) unless specifically designed for retry safety.
- A backoff strategy (e.g., exponential backoff with jitter) **SHOULD** be used to prevent cascading failures.
- Retries **MUST** be capped with a maximum retry count to avoid infinite loops or unnecessary resource consumption.



Example

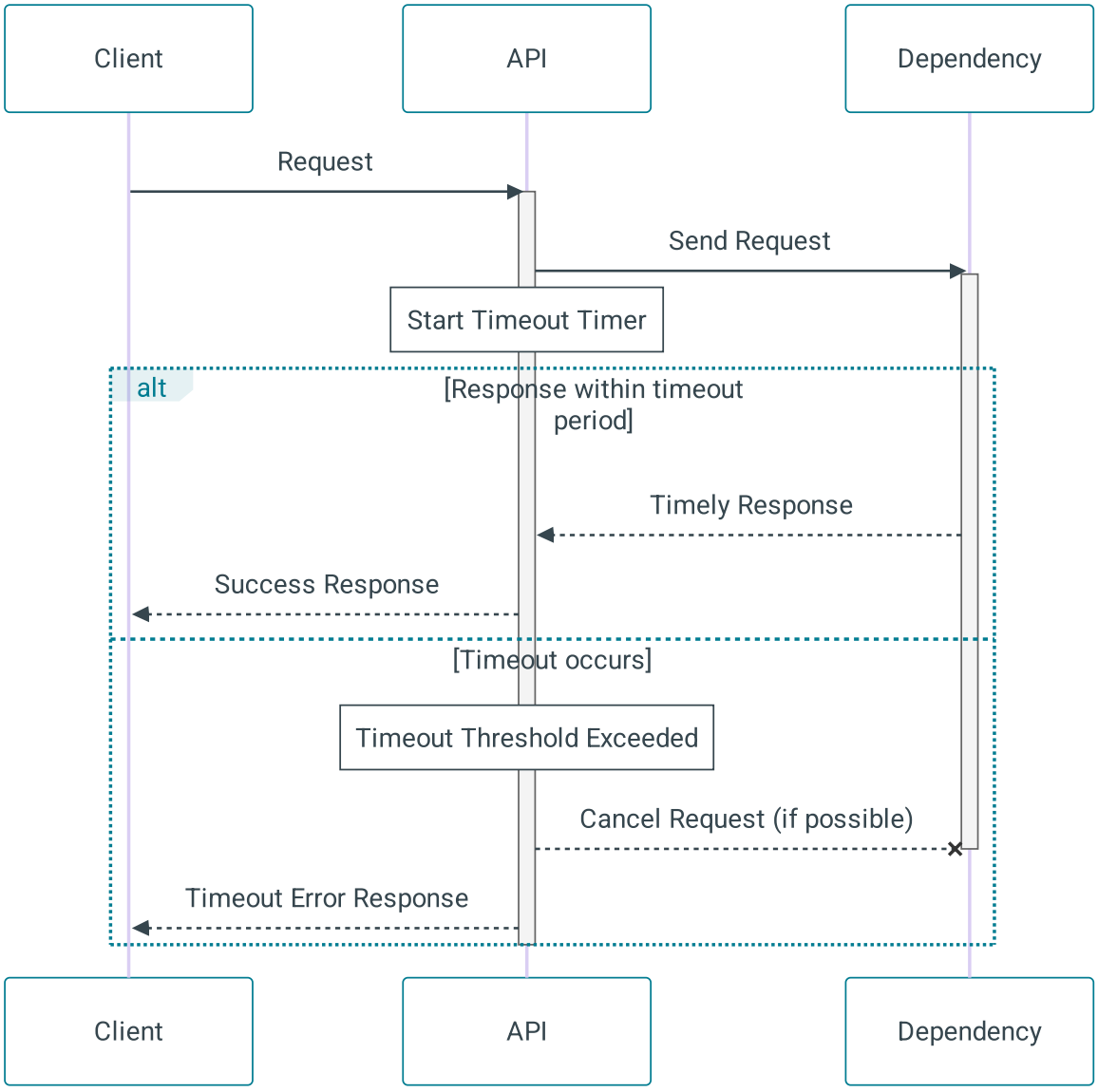
In the following python example, we show data retrieval from a distributed database service that occasionally experiences network issues and use the Tenacity library to handle retries with exponential backoff and jitter. The `@retry` decorator configures the retry behaviour to attempt up to 3 times for transient exceptions only, with exponential backoff starting at 100ms and built-in jitter.

```
import logging
from tenacity import (
    retry,
    stop_after_attempt,
    wait_random_exponential,
    retry_if_exception_type,
    before_sleep_log
)

logger = logging.getLogger(__name__)

class TransientException(Exception):
    """Represents a temporary failure that may succeed on retry"""
    pass

class ServiceException(Exception):
    """Represents a permanent failure or failure after retries"""
    pass

@retry(
    retry=retry_if_exception_type(TransientException),
    stop=stop_after_attempt(3),
    wait=wait_random_exponential(multiplier=0.1, max=2),
    before_sleep=before_sleep_log(logger, logging.WARNING),
    reraise=True
)
def get_customer_data(customer_id):
    """
    Retrieve customer data with automatic retry handling for transient
failures.

    This function will retry up to 2 times (3 attempts total) when
transient
    exceptions occur, using exponential backoff with jitter to prevent
    overwhelming the database.
    """
    try:
        return database_service.get_customer(customer_id)
    except TransientException:
        # Will be automatically retried by Tenacity
        raise
    except Exception as e:
        # Convert unexpected exceptions to ServiceException (not retried)
        raise ServiceException(f"Failed to retrieve customer data:
{str(e)}") from e
```

Timeouts

Timeouts prevent operations from hanging indefinitely.

- APIs **SHOULD** define timeouts for all external calls to dependent systems or services.
- Timeout values **SHOULD** be carefully chosen based on the performance characteristics of the dependent system and the API's SLA requirements.
- APIs **SHOULD NOT** rely on default or unspecified timeout settings, as these can vary widely across libraries and tools.
- A timeout **SHOULD** trigger a fallback mechanism or propagate an appropriate error to the client.



Example

In this python example, we set a 5-second timeout for the request to an external API using the Requests library. If the request takes longer than this, a `Timeout` exception is raised, allowing us to handle it gracefully with a fallback mechanism or propagate an appropriate error to the client.

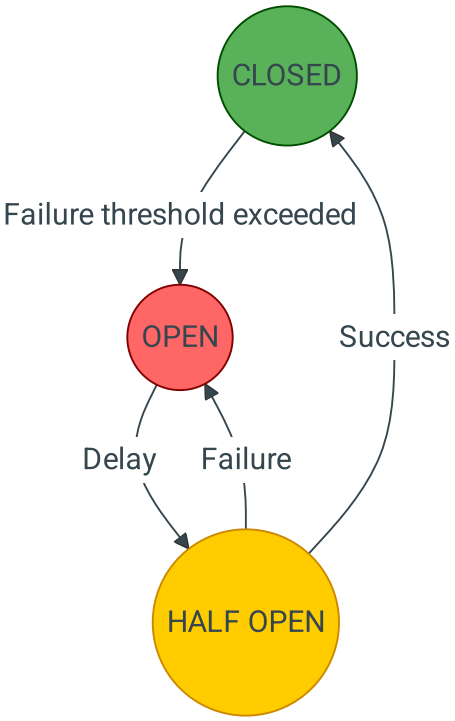
```
import requests
from requests.exceptions import Timeout, RequestException

try:
    # Set a 5-second timeout for the request
    response = requests.get('https://api.example.com/data', timeout=5)
    data = response.json()
except Timeout:
    print("The request timed out")
except RequestException as e:
    print(f"Request error: {e}")
```

Circuit Breakers

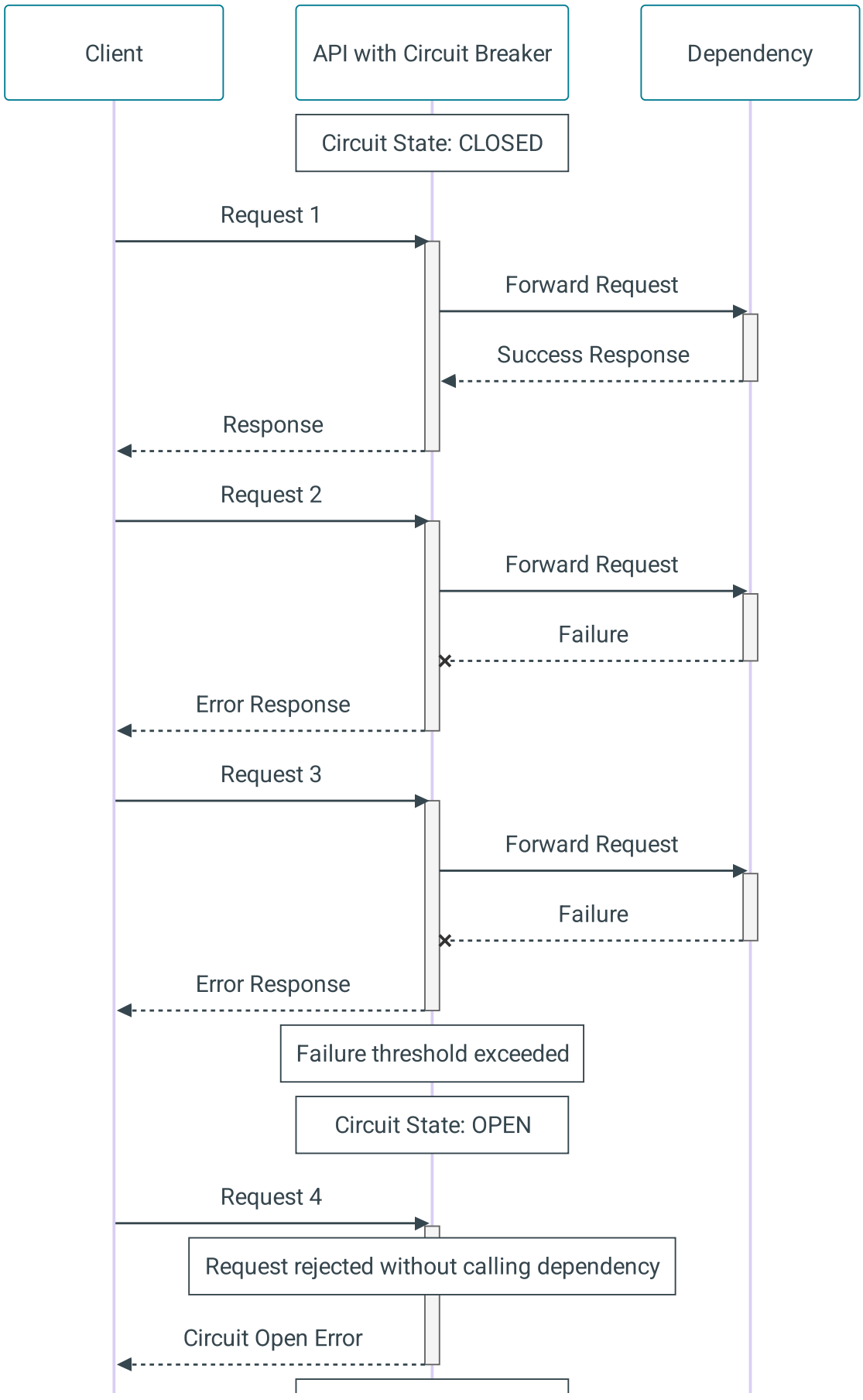
Circuit breakers protect systems from cascading failures by halting requests to unhealthy dependencies.

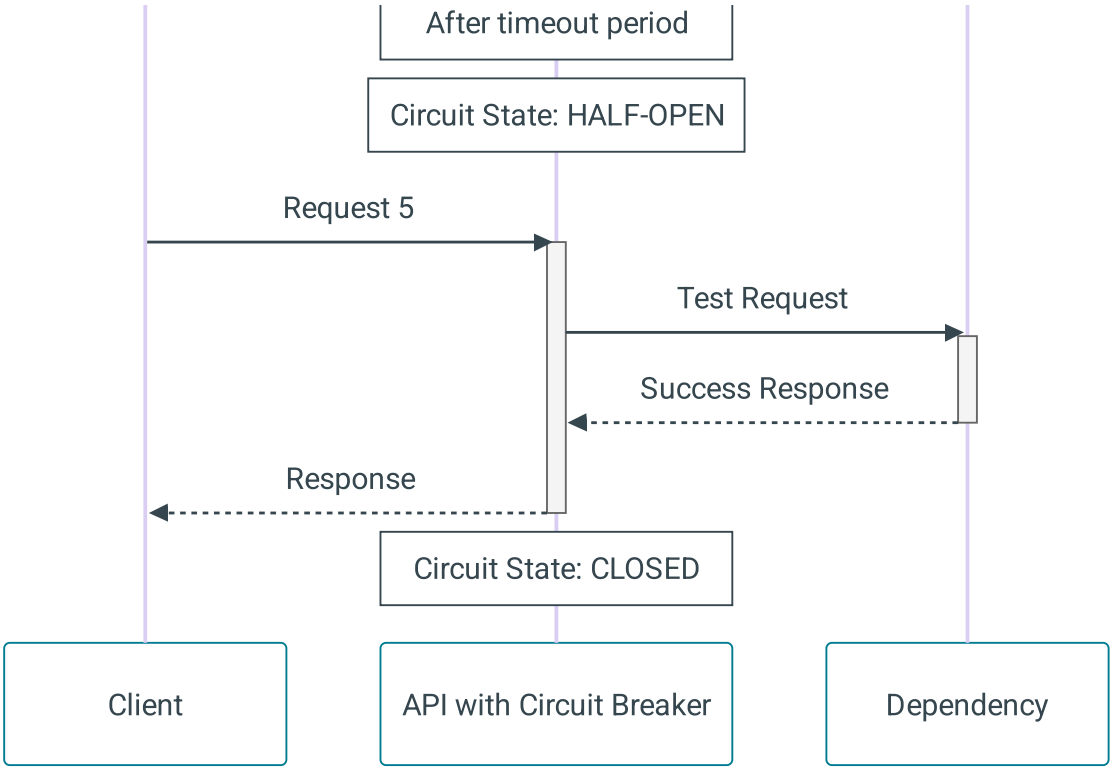
- Circuit breakers **SHOULD** be implemented for calls to external systems that are critical to the API's operation.
- APIs **SHOULD** configure circuit breakers with thresholds for failure rates and recovery intervals.
- When a circuit breaker is open, the API **MUST** provide a meaningful error response or fallback mechanism.
- Circuit breakers **MUST NOT** be used for internal components that are highly reliable and tightly coupled, as they introduce unnecessary complexity.

Flowchart Diagram



Sequence Diagram


Client
API with Circuit Breaker
Dependency
Circuit State: CLOSED
Request 1
Forward Request
Success Response
Response
Request 2
Forward Request
Failure
Error Response
Request 3
Forward Request
Failure
Error Response
Failure threshold exceeded
Circuit State: OPEN
Request 4
Request rejected without calling dependency
Circuit Open Error



Example

In this python example, we use the circuitbreaker library to protect calls to a recommendation service. The circuit breaker is configured to open after 3 failures out of 5 attempts (60% failure rate) and will stay open for 30 seconds before allowing a test request. When the circuit is open, we fallback to a cache of popular products instead of personalised recommendations.

```
import logging
from circuitbreaker import circuit, CircuitBreakerError

logger = logging.getLogger(__name__)

# Configure the circuit breaker:
# - fails when 3 out of 5 attempts fail (60% failure rate)
# - resets after 30 seconds in open state
@circuit(failure_threshold=3, recovery_timeout=30,
expected_exception=Exception)
def get_product_recommendations(user_id):
    """
    Retrieve product recommendations from the recommendation service.

    This function is protected by a circuit breaker that will open after
    3 failures out of 5 attempts, preventing further calls to the
potentially
    failing service for 30 seconds.
    """
    try:
        return recommendation_service.get_recommendations(user_id)
    except Exception as e:
        logger.error(f"Recommendation service error: {str(e)}")
        raise  # The circuit breaker will catch this

def get_recommendations_with_fallback(user_id):
    """
    Get product recommendations with circuit breaker protection and
fallback.
    """
    try:
        # This call is protected by the circuit breaker decorator
        return get_product_recommendations(user_id)
    except CircuitBreakerError:
        logger.warning(f"Circuit breaker open, using fallback for user
{user_id}")
        # Fallback to a simpler recommendation strategy
        return get_fallback_recommendations(user_id)
    except Exception as e:
        logger.error(f"Unexpected error in recommendations: {str(e)}")
        return []

def get_fallback_recommendations(user_id):
    """
    Provides a fallback when the recommendation service is unavailable.
    Returns popular products instead of personalised recommendations.
```

```
    """
    return popular_products_cache.get_popular_items(5)
```

Bulkheads

Bulkheads isolate failures to prevent them from impacting the entire system.

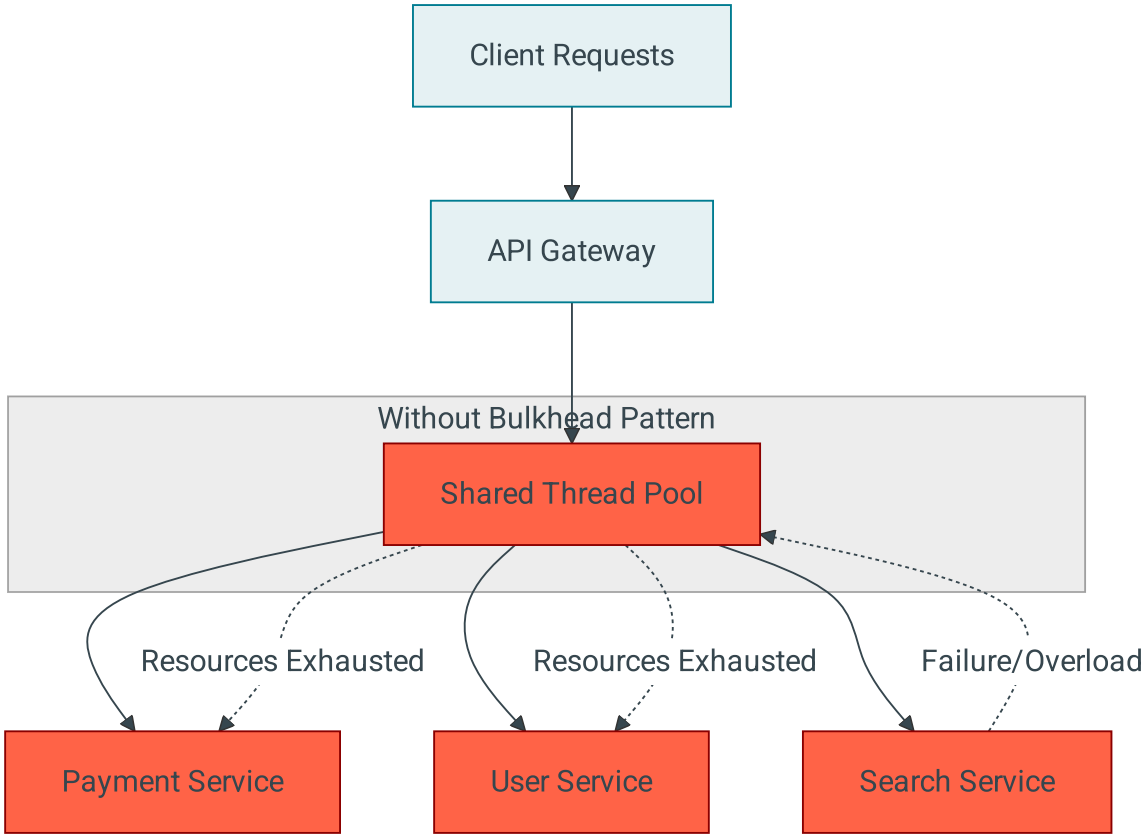
- APIs **SHOULD** use bulkheads to limit the impact of resource exhaustion (e.g., thread pools, connection pools) caused by a specific dependency or client.
- Bulkheads **MUST** be configured to allocate capacity proportionate to the criticality of the resource or operation.
- APIs **MUST NOT** allow a single poorly performing client or dependency to consume all available resources, degrading the experience for others.

Examples

In an e-commerce platform, the payment service, user service, and search service can be isolated using bulkheads. If the search service experiences high traffic or failure, the payment and user services remain unaffected, ensuring critical operations like checkout continue to function.

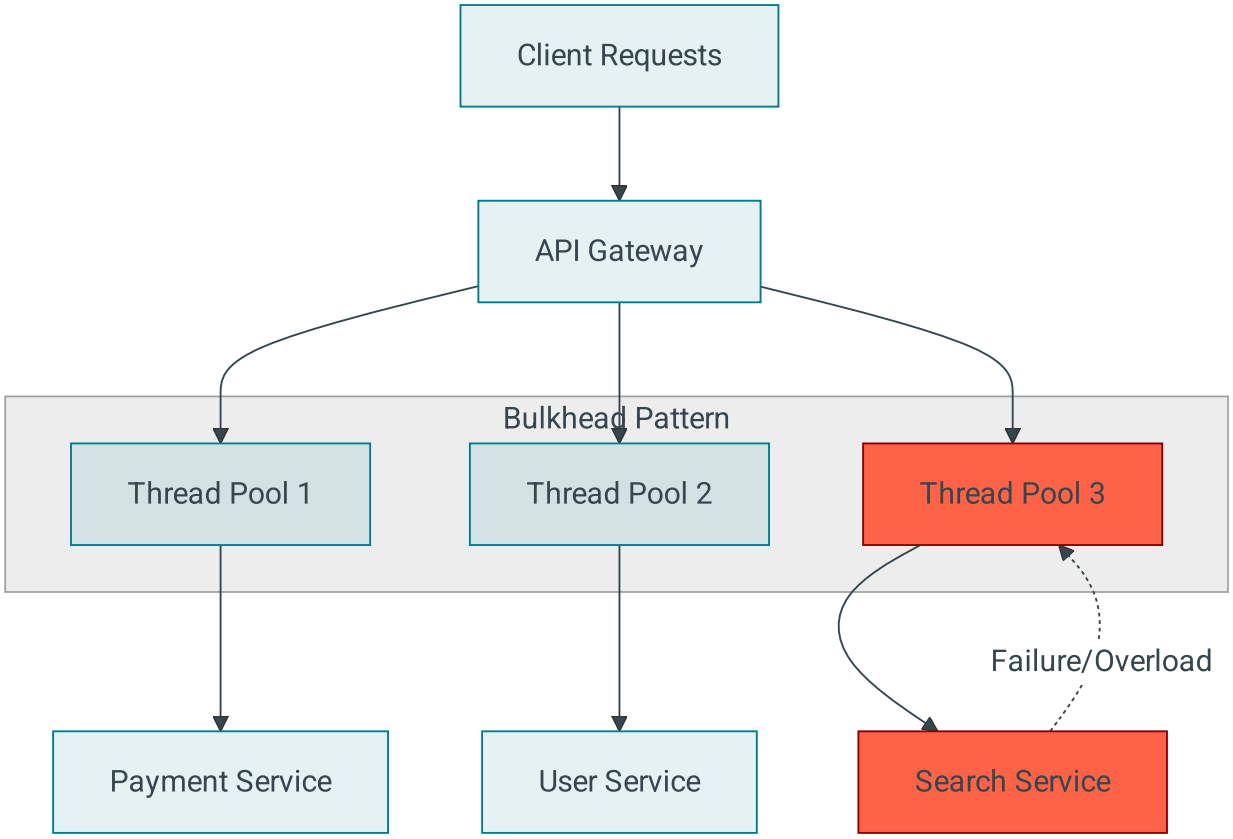
Without Bulkhead Pattern

When `Search` service fails, it consumes all available resources in the shared pool, causing `Payment` and `User` services to suffer as well.



With Bulkhead Pattern

Even though `Search` service has failed, `Payment` and `User` services continue to function, because resources are isolated with bulkheads.



Python Example

In this Python example, we implement the bulkhead pattern using `ThreadPoolExecutor` from the `concurrent.futures` module. The code creates separate thread pools for critical and non-critical operations, preventing failures in one service from consuming resources needed by others.

```
import asyncio
from concurrent.futures import ThreadPoolExecutor
from functools import partial

class ServiceExecutors:
    def __init__(self):
        # Dedicated pool for critical operations
        self.critical_pool = ThreadPoolExecutor(
            max_workers=4,
            thread_name_prefix="critical"
        )
        # Pool for non-critical operations
        self.normal_pool = ThreadPoolExecutor(
            max_workers=10,
            thread_name_prefix="normal"
        )

    async def execute_critical(self, func, *args):
        return await asyncio.get_event_loop().run_in_executor(
            self.critical_pool,
            partial(func, *args)
        )

    async def execute_normal(self, func, *args):
        return await asyncio.get_event_loop().run_in_executor(
            self.normal_pool,
            partial(func, *args)
        )
```

Usage example:

```
executors = ServiceExecutors()

# Payment processing - uses the critical pool (4 threads max)
async def process_payment(payment_id):
    return await executors.execute_critical(payment_service.process,
payment_id)

# Product search - uses the normal pool (10 threads max)
async def search_products(query):
    return await executors.execute_normal(search_service.find, query)

# Even if search_products overloads its thread pool,
# payment processing remains unaffected
```

This implementation demonstrates how:

- Critical operations like payments get dedicated resources (4 threads)
- Non-critical operations like search get separate resources (10 threads)
- If the search service becomes overloaded, payment processing continues normally
- Each service has its failure domain contained within its own thread pool

Fallbacks

Fallbacks provide alternative behaviour when a dependency fails.

- APIs **MUST** implement fallbacks for critical operations where failure would significantly impact the user experience.
- Fallbacks **SHOULD** provide meaningful degraded functionality (e.g., cached data, placeholder values) rather than returning generic errors.
- APIs **MUST NOT** use fallbacks that violate business logic, security, or data integrity requirements.
- Where fallbacks are implemented, the API **SHOULD** log the use of fallback mechanisms for monitoring and debugging purposes.

Example

If a weather API fails, the fallback could provide cached weather data from the last successful response. For a stock price API, a fallback might return the last known price or a default value.

```python
import requests

# Simulate a cache (in a real app, this would be persistent storage)
weather_cache = {
    "London": {"temperature": 15, "condition": "Cloudy"},
    "New York": {"temperature": 20, "condition": "Sunny"}
}

def get_weather(city):
    """Get weather data with fallback to cache if API fails"""
    try:
        # Try to get fresh data from the API
        response = requests.get(
            f"https://api.weather.example.com/current?city={city}",
            timeout=2
        )
        response.raise_for_status()
        return response.json()

    except Exception:
        # API call failed, use fallback
        print(f"Weather API failed. Using cached data for {city}")

        # Return cached data if available, or a default
        if city in weather_cache:
            return weather_cache[city]
        else:
            return {"temperature": None, "condition": "Unknown"}

# Example usage
weather = get_weather("London")
print(f"Weather: {weather['temperature']}°C, {weather['condition']}")
```

General Guidance

- Resilience patterns **MUST** be chosen based on the specific context and requirements of the API.
- Combinations of patterns **SHOULD** be used to address complex failure scenarios (e.g., retries with timeouts and circuit breakers).
- APIs **MUST** log and monitor resilience events (e.g., retries, circuit breaker state changes) to enable proactive troubleshooting and optimisation.

- Overuse or misuse of resilience patterns **MUST NOT** degrade overall performance or introduce unnecessary latency.

1.11.4 Monitoring & Observability

Effective monitoring and observability **MUST** be integrated into all APIs to ensure performance, reliability, continuous improvement, and meet any Service Level Agreements (SLAs) and compliance requirements. This section provides guidance on best practices, including the use of OpenTelemetry and key metrics to track.

Monitoring vs. Observability

While often used interchangeably, monitoring and observability serve distinct purposes in API management. **Monitoring** focuses on tracking predefined metrics and known system states, typically answering "*is the system working as expected?*" through dashboards and alerts. **Observability**, on the other hand, provides the capability to understand unknown system states and answer new questions without deploying additional instrumentation. It combines `metrics`, `logs`, and `traces` to give comprehensive insights into system behaviour.

Effective API management requires both, monitoring for known issues and observability for debugging complex, unforeseen problems that may emerge in distributed systems.

General Guidance

- **MUST** implement monitoring to gain visibility into API performance and health.
- Monitoring **SHOULD** be designed to provide actionable insights for troubleshooting and optimisation.
- APIs **SHOULD** adopt the observability standard OpenTelemetry for distributed tracing, metrics, and logs and ensure consistency across all APIs regardless of language / frameworks.
- APIs **MUST NOT** rely on ad-hoc monitoring solutions that lack consistency and integration.
- Monitoring **MUST NOT** introduce significant performance overhead that degrades the API's functionality.

- APIs in development or testing environments **SHOULD** use monitoring to identify issues early but **MUST NOT** rely on it as a substitute for proper testing.
- Monitoring systems **MUST** integrate with alerting tools such as Prometheus to notify teams of critical issues.
- **SHOULD** implement health checks to regularly verify the availability of API endpoints.
- **SHOULD** establish a centralised logging system to aggregate logs from all API services for easier analysis.
- **MUST NOT** neglect documentation of monitoring setups, as this **MUST** be accessible to all relevant teams.

Recommended Metrics

APIs **SHOULD** be tracking the following key metrics to ensure comprehensive monitoring:

Latency

- **MUST** measure the time taken to process requests, including response times for various endpoints.
- **SHOULD** categorise latency measurements by percentiles (e.g., p50, p95, p99) to identify performance issues.
- High latency **MUST** trigger alerts for investigation.
- **MUST NOT** ignore latency spikes, as they can indicate underlying problems with the service.

Traffic

- **MUST** monitor the volume of requests to understand usage patterns, traffic trends and detect anomalies.
- **SHOULD** include metrics such as requests per second (RPS) and client IP counts to gauge load.

- **MUST NOT** underestimate the importance of traffic analysis, as it helps in capacity planning.

Errors

- **MUST** track error rates (e.g., `4xx` and `5xx` responses) to identify issues with API endpoints.
- **SHOULD** categorise errors by type where appropriate (e.g., client errors, server errors) to facilitate troubleshooting.
- **MUST NOT** ignore increasing error rates; they **MUST** prompt immediate investigation.

Saturation

- **MUST** monitor resource utilisation (e.g., CPU, memory, database connections, thread pools) to detect saturation points.
- **SHOULD** set alerts for resource thresholds to proactively address potential bottlenecks.
- **MUST NOT** assume that higher resource utilisation is acceptable without appropriate scaling strategies.

DORA Metrics

To align with industry best practices, API teams **SHOULD** implement the four key metrics of DORA (DevOps Research and Assessment) to measure development and operational performance:

Throughput

- **Lead Time for Changes**: This metric measures the time it takes for a code commit or change to be successfully deployed to production. It reflects the efficiency of your software delivery process.
- **Deployment Frequency**: This metric measures how often application changes are deployed to production. Higher deployment frequency indicates a more efficient and responsive delivery process.

Stability

- **Change Failure Rate**: This metric measures the percentage of deployments that cause failures in production, requiring hotfixes or rollbacks. A lower change failure rate indicates a more reliable delivery process.
- **Mean Time to Recovery (MTTR)**: Measure the time taken to recover from failures.

Tools and Standards

Tool/Standard	Description
OpenTelemetry	A vendor-neutral framework for collecting and exporting telemetry data, allowing teams to choose their preferred backend for analysis and visualisation. It supports various programming languages and platforms, making it a versatile choice for API observability.
Grafana	A powerful open-source analytics and monitoring platform that integrates with various data sources, including Prometheus. It provides rich visualisations and dashboards for real-time monitoring and analysis of API performance metrics.
Grafana Loki	A log aggregation system designed to work seamlessly with Grafana. It allows for the collection, storage, and querying of logs, making it easier to correlate logs with metrics and traces for comprehensive observability.
Prometheus	An open-source monitoring and alerting toolkit designed for reliability and scalability, particularly suited for dynamic cloud environments. It collects metrics from configured targets at specified intervals, evaluates rule expressions, and can trigger alerts based on those evaluations.
ELK Stack (Elasticsearch, Logstash, Kibana)	A popular open-source stack for log management and analysis. Elasticsearch stores and indexes logs, Logstash processes and ingests logs from various sources, and Kibana provides a web interface for visualising and exploring log data.

1.12 Governance

TODO

- **Approval Process**: Guidelines for reviewing and changes to the guidelines or spectral rules.
- **Governance automation**: Using automated tools (linters, validators) to ensure compliance with the guidelines.
- Specteral Rules
- PR process
- **Change Management**: Version control and change request procedures.

1.13 Data Standards

Data standards provide a common language for representing information, enabling different systems to understand and process data without ambiguity.

Consideration **MUST** be given to the use of appropriate data standards to ensure consistency and ease of integration.

1.13.1 Core Principles

- APIs **MUST** use consistent data formats and standards across all endpoints.
- APIs **MUST** validate all incoming data against defined schemas.
- APIs **MUST** follow data protection and privacy requirements for sensitive data.
- APIs **MUST** document any deviations from standard formats.

1.13.2 Data Models vs. Data Representations

It is important to differentiate between *data models* and *data representations*:

- **Data Model** defines the structure, relationships, and constraints of data within a specific domain. It is a conceptual blueprint that outlines how data elements relate to each other and the rules governing their use.
- **Data Representation** is the concrete format in which data is serialised for exchange or storage. For RESTful APIs, this is commonly JSON.

1.13.3 Industry Standards

APIs **SHOULD** adopt a domain-specific UKHSA data model or adopt an existing industry standard where appropriate while still using JSON as its core/principal data representation.

When defining new APIs or uplifting APIs it is important to look for industry standards and open standards that have already been adopted within UKHSA or by other related organisations and industries, such as FHIR for health data which is used by NHS England and OMOP for data analysis.

FHIR Implementations

If implementing the FHIR standard:

- APIs **MUST** use FHIR UK Core profiles where they exist.
- APIs **MUST** document any extensions to standard FHIR resources.
- APIs **SHOULD** implement FHIR REST API patterns as described in the FHIR specification.
- APIs **MAY** create custom FHIR profiles when UK Core profiles don't meet your needs.

OMOP Implementations

If implementing the OMOP Common Data Model:

- APIs **MUST** use standardised clinical tables as defined in the OMOP CDM specification.
- APIs **MUST** map source terminologies/vocabularies to OMOP standard concepts.
- APIs **SHOULD** implement OMOP data quality assessment procedures.
- APIs **MAY** create ETL processes to synchronised between OMOP and other standards such as FHIR when both are needed.

1.13.4 Terminology Standards

Terminology (or controlled vocabularies) play a crucial role in ensuring that data has a consistent and unambiguous meaning.

Using common terminologies is essential for data quality, consistency, and interoperability.

Terminology is *not* the same as FHIR. FHIR provides the *structure* and *format* for exchanging data, while terminology defines the *meaning* of the data elements within that structure.

APIs **MUST** adopt standardised terminologies (e.g., SNOMED CT, ICD-10, dm+d) whenever applicable.

APIs **SHOULD** specify the required terminologies for each data element within their OpenAPI definition, taking into account regional differences.

Terminology Implementations

- APIs **SHOULD** use SNOMED CT for clinical terms.
- APIs **SHOULD** use ICD-10 for medical diagnosis.
- APIs **SHOULD** use dm+d for medicines and devices in England.
- APIs **SHOULD** document any regional terminology variations for Scotland, Wales, and Northern Ireland.
- APIs **SHOULD** provide terminology mappings when exchanging data across regions

1.13.5 Additional Considerations

Compliance

If there are regulatory or industry compliance requirements that mandate the use of specific data standards, these **MUST** be adhered to.

Interoperability

When APIs are designed to exchange data with external systems, especially within a specific industry or domain, a recognised data standard **SHOULD** be adopted. This ensures that both the API and the consuming systems can understand the data exchanged.

Over-Engineering

Data standards **MUST NOT** be applied blindly to every API. If an API's scope is extremely narrow, if it is not intended for data exchange, and if there are no compelling reasons for standardisation, then a custom model and representation may be more appropriate.

Performance Degradation

If adopting a data standard would introduce significant overhead in terms of processing or data size, and if interoperability is not a critical requirement, a standard **MUST NOT** be forced into the design.

Fit for purpose

If the data standard doesn't have the necessary types or fields to correctly describe the data, it **MUST NOT** be forced into the design.

Internal APIs (Limited Scope)

In cases where APIs are purely internal and their data is not intended for broader exchange, the use of data standards **MAY** be considered if it would improve the consistency between internal services.

1.13.6 Government Data Standards

As per the GDS Guidence you **SHOULD** design your APIs to follow appropriate government data standards in the Data Standards Catalog and External Standards Catalog.

Other relevent standards

- **JSON** (RFC8259) is a lightweight, text-based, language-independent data interchange format.
- **GeoJSON** (RFC7946) is a geospatial data interchange format based on JavaScript Object Notation (JSON).

See Common Data Types for additional standards.

1.14 Common Data Types

API types **MUST** use standard data formats.

Open API (based on JSON Schema Validation vocabulary) defines formats from ISO and IETF standards for date/time, integers/numbers and binary data.

APIs **MUST** use these formats, whenever applicable:

1.14.1 OpenAPI Formats Registry

The following list is provided for brevity and includes examples but please use OpenAPI Formats Registry as the master list.

OpenAPI format	OpenAPI type	Specification	Example
`bigint`	`integer`	arbitrarily large signed integer number	`7.72E+19`
`binary`	`string`	base64url encoded byte sequence following RFC7493 Section 4.4	`"VGVzdA=="`
`byte`	`string`	base64url encoded byte following RFC7493 Section 4.4	`"VA=="`
`date`	`string`	RFC3339 internet profile - subset of ISO 8601.	`"2019-07-30"`
`date-time`	`string`	RFC3339 internet profile - subset of ISO 8601.	`"2019-07-30T06:43:40.252Z"`
`decimal`	`number`	arbitrarily precise signed decimal number	`3.141593`
`double`	`number`	binary64 double precision decimal number - see IEEE 754-2008/ISO	`3.141593`

OpenAPI format	OpenAPI type	Specification	Example
		60559:2011	
`duration`	`string`	RFC3339 - subset of ISO 8601.	`"P1DT3H4S"`
`email`	`string`	RFC5322	`"example@example.com"`
`float`	`number`	binary32 single precision decimal number - see IEEE 754-2008/ISO 60559:2011	`3.141593`
`hostname`	`string`	RFC1034	`"www.example.com"`
`idn-email`	`string`	RFC6531	`"hello@bücher.example"`
`idn-hostname`	`string`	RFC5890	`"bücher.example"`
int32	integer	4 byte signed integer between -2^{31} and $2^{31}-1$	`7.72E+09`
int64	integer	8 byte signed integer between -2^{63} and $2^{63}-1$	`7.72E+14`
ipv4	string	RFC2673	`"104.75.173.179"`
ipv6	string	RFC4291	`"2600:1401:2::8a"`
`iri`	`string`	RFC3987	`"https://bücher.example/"`
`iri-reference`	`string`	RFC3987	`"/damenbekleidung-jacken-mäntel/"`
`json-pointer`	`string`	RFC6901	`"/items/0/id"`
`password`	`string`		`"secret"`
`period`	`string`	RFC3339 - subset of ISO 8601.	`"2022-06-30T14:52:44.276/PT48H"` `"PT24H/2023-07-`

OpenAPI format	OpenAPI type	Specification	Example
			`30T18:22:16.315Z" "2024-05-15T09:48:56.317Z/.."`
`regex`	`string`	regular expressions as defined in ECMA 262	`"^[a-z0-9]+$"`
`relative-json-pointer`	`string`	Relative JSON pointers	`"1/id"`
`time`	`string`	RFC3339 internet profile - subset of ISO 8601.	`"06:43:40.252Z"`
`uri`	`string`	RFC3986	`"https://www.example.com/"`
`uri-reference`	`string`	RFC3986	`"/clothing/"`
`uri-template`	`string`	RFC6570	`"/users/{id}"`
`uuid`	`string`	RFC4122	`"e2ab873e-b295-11e9-9c02-…"`

1.14.2 Additional Formats

APIs **SHOULD** also consider using the following formats.

format	OpenAPI type	Specification	Example
`bcp47`	`string`	multi letter language tag - see BCP 47. It is a compatible extension of ISO 639-1 optionally with additional information for language usage, like region, variant, script.	`"en-DE"`
`gtin-13`	`string`	Global Trade Item Number - see GTIN	`"5710798389878"`

format	OpenAPI type	Specification	Example
`iso-3166-alpha-2`	`string`	two letter country code - see ISO 3166-1 alpha-2.	`"GB"` Hint: It is `"GB"` not `"UK"`.
`iso-4217`	`string`	three letter currency code - see ISO 4217	`"EUR"`
`iso-639-1`	`string`	two letter language code - see ISO 639-1.	"en"

1.15 Integration Patterns

SHOULD use standard API integration patterns.

1.15.1 Anti-corruption layer

SHOULD wrap legacy APIs in an anti-corruption layer (ACL) so that consumers are able to use modern REST-based API semantics.

Examples

Translate `SOAP` calls to `REST` calls.

- The ACL makes SOAP requests to the legacy API and exposes a RESTful endpoint for the new system.
- The ACL takes the XML data from the SOAP response and transforms it into a JSON format that the modern system expects.
- The ACL can map legacy API operations to RESTful principles (resources and HTTP methods).

1.15.2 Asynchronous Request-Reply

SHOULD use the asynchronous request-reply API pattern for long running tasks, such as processing large datasets, image or video processing, or complex calculations.

- **Callback Pattern**: Use when the client provides an API to receive notifications.
- **Polling Pattern**: Use when the client does not provide an API to receive notifications.

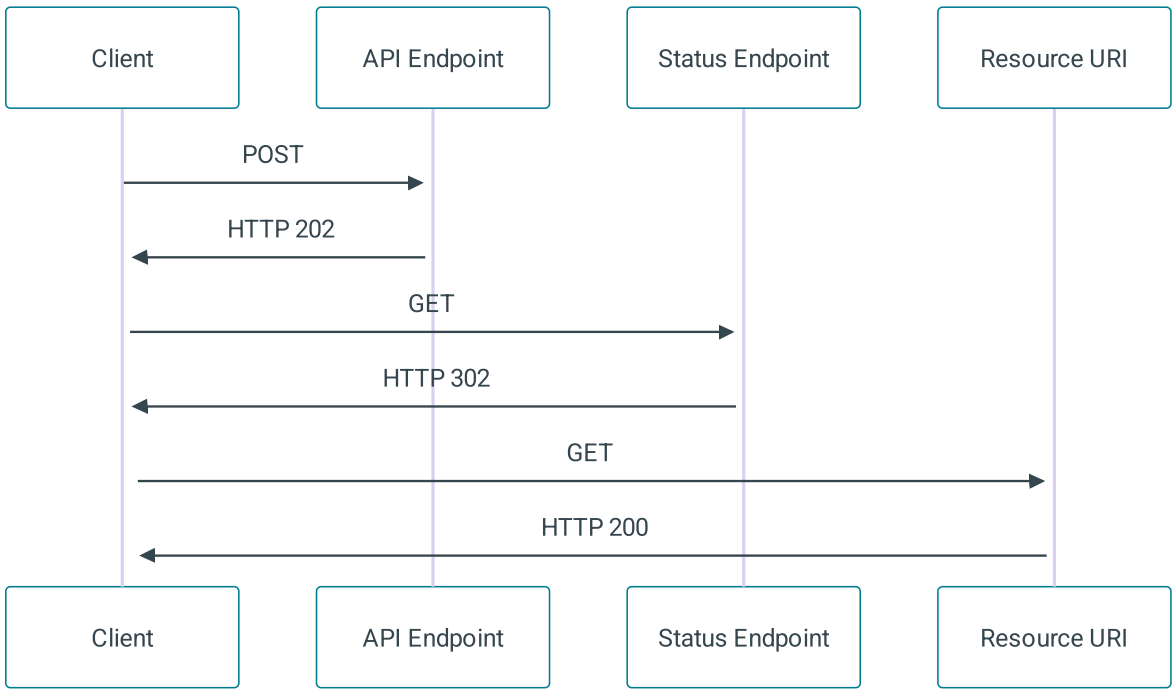
Examples

```
POST /namespace/product/v1/tasks/123/start
Response: 202 Accepted

{
  "task_id": "123",
  "status": "in_progress"
}
```

Poll for status:

```
GET /namespace/product/v1/tasks/123/status
Response: 200 OK

{
  "status": "completed",
  "result": "success"
}
```

Typical polling flow



1.15.3 Bulk

SHOULD use bulk integration patterns to manage the processing of large volumes of data or multiple transactions in a single operation. This approach enhances efficiency and reduces the overhead associated with processing individual requests.

Examples

- **Batch Processing APIs:** Endpoints that accept multiple records in a single request for insertion, update, or deletion.
- **Bulk Data Import/Export:** Tools and APIs that allow the import or export of datasets in formats like CSV, JSON, or XML.
- **Parallel Processing:** Distributing tasks across multiple processors or machines to handle large-scale data operations concurrently.

Best Practices

- **Validate Data Thoroughly:** Ensure all records meet validation rules before processing to avoid partial failures.
- **Provide Detailed Feedback:** After processing, return information about successes and failures for individual records.
- **Optimize Performance:** Utilize efficient data structures and algorithms to handle large datasets without significant delays.
- **Manage Resource Utilization:** Monitor and limit resource usage to prevent system overloads during bulk operations.

2 API Guidance Summary

2.1 API Guidance Summary

This is summary high-level guidance for API producers (application teams) on adopting the API principles, patterns and practices developed as part of the Big Rocks API strategy. Refer to the full API Strategy and other references linked at the end of this document for more information.

2.1.1 Definitions

- **APIM Platform**: API Management Platform, a UKHSA-wide platform for managing and accessing APIs.
- **Developer Portal**: Component of the APIM Platform used by developers to access APIs, onboard new APIs, and view API documentation.
- **API Catalogue**: Feature of the Developer Portal that will contain all information related to APIs onboarded and available on the APIM Platform.

2.1.2 Principles

These are the core high level principles to follow when designing, building, testing and deploying your APIs.

Prioritise Reusability

> **Tip**
>
> Apply the API Design Guidelines and use the features of Developer Portal.

- **Check the API does not already exist** by reviewing the UKHSA API Catalogue as well as the cross-government UK API Catalogue. Evaluate if an existing API could potentially be enhanced to also support the new use case.

- **Design your API to be reused**. APIs should be broken down into reusable composable interactions and data groupings. APIs should aim to be use case agnostic if possible and the design and naming must be consistent with the established API Design Guidelines.

Adopt API-first Practices

> **Tip**
>
> Apply the API Design Guidelines and use the features of Developer Portal.

- **Design the API first**. Follow GDS guidance and UKHSA API Design Guidelines.
- Produce an OpenAPI definition utilising the OpenAPI specification as the first output of your design process, and then develop it iteratively along with the service.
- Share this specification early in development using the Developer Portal to get early feedback on your design.

> **Note**
>
> API first is the practice of designing software starting with an API, before designing your web or mobile user interface. Developing the API before the rest of the service means a platform or service can be built around the API.

Use Established API Patterns & Standards

> **Tip**
>
> Adopt the recommended API patterns and data standards.

- **Adopt the recommended patterns & standards**, including industry and open standards where appropriate. Follow the Technology Code of Practice and other standards

recommended in UKHSA API guidelines, such as `HTTP REST`, `JSON` and related industry standards used by NHS such as `FHIR` and `OMOP`.

Prioritise API Security

> **Tip**
>
> Adopt the API security patterns.

- **Follow secure by design process** in the Secure by Design Guidelines and industry best practices, including the OWASP API Security Project. Ensure your API has extensive tests that validate inputs.
- Ensure APIs have robust authorisation and authentication based on industry standards, such as OAuth 2.0 and OpenID Connect. The APIM Platform will act as a “transparent proxy” in authorisation scenarios, which includes passing through of auth tokens to backend APIs.
- Ensure that APIs are protected against overuse using rate limits by leveraging the features of the APIM Platform.

Manage API Lifecycles

> **Tip**
>
> Adopt the API versioning & deprecation patterns.

- **Use the recommended versioning scheme** to set clear expectations for clients on how change will be managed. Keep the number of active versions of an API to a minimum and have a process to retire old API versions. Refer to the Big Rocks guidance on API patterns for more information.

Generate API Documentation

> **Tip**
>
> Create an OpenAPI definition and use the features of Developer Portal to publish it.

- **Ensure the API is well documented** using an OpenAPI definition. Documentation should be concise and easy for developers use. The specification is machine-readable and will support the generation of consistent accessible documentation. It can also be used to accelerate development and testing through code generation.
- Use the Developer Portal to make the API discoverable and ensure it is always accurate, consistent, usable, and discoverable. This documentation supplements the solution documentation on Confluence and LeanIX Basic Concepts and Modelling Guidance.

Support Testing with API Specifications

> **Tip**
>
> Adopt the recommended testing patterns and use the features of Developer Portal in the SIT environment to support testing.

- Use the OpenAPI definition to help **define testing requirements early in development** and to prepare the test scripts and data that will be needed. Use tools that automatically generate test stubs and client code from your OpenAPI definition to build functional tests.

Test API Performance

> **Tip**
>
> Adopt the recommended performance, reliability and monitoring guidelines and use the features of Developer Portal in the NFT environment.

- **Test performance meets non-functional requirements**. Response times and availability must conform to UKHSA standards and provide a high quality of service to clients.

Follow Regulations & UKHSA Governance

> **Tip**
>
> Check the onboarding requirements if you are API product or consumer and want to onboard your application or service.

- Follow the Technology Code of Practice and other UK Government standards. Follow relevant regulations, including the UK General Protection Regulation (GDPR). Ensure your API has a business case and technical solution that complies with organisational standards and is aligned with UKHSA technology strategy.
- Follow the Technology Governance Schedule and Audit Trail process before onboarding a new solution into the APIM Platform.
- Ensure API Onboarding requirements are met.

3 Spectral Rules

3.1 UKHSA Spectral Rules

3.1.1 Overview

A linting ruleset was created to support API Developers/Providers in achieving the standards described in the UKHSA API Guidelines, ensuring consistency, reliability, and security across all APIs developed within or on behalf of UKHSA.

As well as the rules described herein, the UKHSA ruleset includes the recommended built in spectral OpenAPI Rules and the Spectral Documentation Ruleset; These are common sense rules that ensure an OpenAPI definition adheres to the OpenAPI specification, as well as encourage high quality, rich documentation which is especially important for providing the best possible APIM Developer Portal experience.

Where rules been adopted from from existing open source API rulesets a link is supplied on the relevant rule page.

3.1.2 How to use the rules

Install Spectral

Spectral is a flexible JSON/YAML linter for creating automated style guides, with baked in support for OpenAPI (v3.1, v3.0, and v2.0), Arazzo v1.0, as well as AsyncAPI v2.x.

Install Spectral globally or as a dev dependency.

```
npm install -g @stoplight/spectral-cli
```

Read the official spectral documentation for more installation options.

Run Spectral against your OpenAPI definition

Run Spectral against your OpenAPI definition, referencing the spectral ruleset.

You can reference a ruleset hosted via HTTP server.

> You can only reference the raw Github URL if the github repository is public.

```
spectral lint openapi-definition.yml --ruleset
https://raw.githubusercontent.com/ukhsa-collaboration/api-
guidelines/refs/heads/main/.spectral.yaml
```

You can install the ruleset as via npm package and then reference that, bear in mind the UKHSA ruleset npm package is hosted in github so please read Github's documentation Working with the npm registry.

```
npm install @ukhsa-collaboration/spectral-rules
spectral lint openapi-definition.yml --ruleset ./node_modules/@ukhsa-
collaboration/spectral-rules/.spectral.yaml
```

or create a local `.spectral.yml` ruleset which extends the one in this repository.

```
echo "extends: ['@ukhsa-collaboration/spectral-rules']" > .spectral.yml
```

then you can just run the following.

```
spectral lint openapi-definition.yml
```

Review and fix any reported issues

Once the linter has highlighted any issues or errors, review and fix to ensure your OpenAPI definition remains compliant with the UKHSA guidelines.

CI/CD Github Actions

The following is a sample Github actions job which can be used as an example of setting up linting as part of you CI/CD pipeline.

```
...
jobs:
  lint-openapi:
    name: Lint OpenAPI
    runs-on: ubuntu-latest

    permissions:
      contents: read
      issues: read
      checks: write
      pull-requests: write

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '22.x'
          registry-url: 'https://npm.pkg.github.com'
          # Defaults to the user or organization that owns the workflow
file
          scope: '@ukhsa-collaboration'

      - run: npm install @ukhsa-collaboration/spectral-rules
        env:
          NODE_AUTH_TOKEN: ${{ secrets.GITHUB_TOKEN }}

      - name: Install spectral
        run: curl -L
https://raw.github.com/stoplightio/spectral/master/scripts/install.sh | sh

      - name: Lint example OpenAPI
        run: |
          spectral --version
          spectral lint "*.{json,yml,yaml}" -r ${{ GITHUB.WORKSPACE
}}/node_modules/@ukhsa-collaboration/spectral-rules/.spectral.yaml -f
github-actions
```

Additional Recommended Tooling

Tool	Description
VS Code Extension	Official spectral VS Code extension provides real time linting / intellisense on your OpenAPI definition.
Github Action	Official spectral Github action provides ability to lint your OpenAPI definition in CI/CD workflows.

Important

To run the spectral linter in your git hub CI/CD workflow you will need to ensure your repository is added to the list of repositories allowed to download the npm package.

Read the official spectral documentation for more development workflows.

3.2 MUST

3.2.1 MUST define a format for integer types

`integer` properties **MUST** have a format defined (`int32` , `int64` , or `bigint`).

Invalid Example

```
requestBody:
    content:
      application/json:
        schema:
          type: object
          properties:
            range:
              type: integer
```

Valid Example

```
requestBody:
  content:
    application/json:
      schema:
        type: object
        properties:
          range:
            type: integer
            format: int32
```

Zalando Guideline 171

3.2.2 MUST define a format for number types

`number` properties **MUST** have a format defined (`float` , `double` , or `decimal`).

Invalid Example

```
requestBody:
    content:
      application/json:
        schema:
          type: object
          properties:
            range:
              type: number
```

Valid Example

```
requestBody:
  content:
    application/json:
      schema:
        type: object
        properties:
          range:
            type: number
            format: float
```

Zalando Guideline 171

3.2.3 MUST define security schemes

All APIs **MUST** have a security scheme defined.

If an API doesn't have a security scheme defined, it means the entire API is open to the public. That's probably not what you want, even if all the data is read-only. Setting lower rate limits for the public and letting known consumers use more resources is a handy path to monetization, and helps know who your power users are when changes need feedback or migration, even if not just good practice.

Valid Example

```
components:
  securitySchemes:
    oAuth:
      type: oauth2
      description: This API uses OAuth 2 with the authorization code flow.
[More info](https://oauth.net/2/grant-types/authorization-code/)
      flows:
        authorizationCode:
          authorizationUrl: https://domain.test/api/oauth/dialog
          tokenUrl: https://domain.test/api/oauth/token
          refreshUrl: https://domain.test/api/oauth/token
          scopes:
            tests:read: read test results
            tests:write: submit test results
```

UKHSA Guidelines Security

3.2.4 MUST have info api audience

The `info` object **MUST** have an `x-audience` that matches at least one of these values:

audience	Use case
`company-internal`	for internal use only with UKHSA
`partner-external`	for UKHSA partners under a service agreement
`premium-external`	for publicly available but commercial/monetised APIs behind a paywall
`public-external`	for public and freely accessible APIs (e.g. Data Dashboard)

Valid Example

```yaml
info:
  title: Test Results Api
  x-audience: public-external
```

3.2.5 MUST have info contact email

The `info` object **MUST** have a `contact email` property that contains a valid email address for the responsible team.

Valid Example

```
info:
  ...
  ...
  contact:
    email: 'support.contact@acme.com'
```

Zalando Guideline 218

3.2.6 MUST have info contact name

The `info` object **MUST** have a `contact:name` property that contains a valid name for the team or person responsible for the API.

Valid Example

```
info:
  ...
  ...
  contact:
    name: 'Tequila Mockingbirds'
```

Zalando Guideline 218

3.2.7 MUST have info contact url

The `info` object **MUST** have a `contact:url` property that contains a valid URL to contact the team or person responsible for the API.

Valid Example

```
info:
  ...
  ...
  contact:
    ...
    url: https://acme.com
    ...
```

Zalando Guideline 218

3.2.8 MUST have info description

The `info` object **MUST** have a `description` property defined.

Valid Example

```yaml
info:
  description: This describes my API.
  ...
```

Zalando Guideline 218

3.2.9 MUST have info title

The `info` object must have a `title` property defined.

Valid Example

```
info:
  title: Payments API
  ...
```

Zalando Guideline 218

3.2.10 MUST have info value chain

The `info` object **MUST** have an `x-value-chain` that matches at least one of these values.

- `prevent`
- `detect`
- `analyse`
- `respond`
- `cross-cutting`
- `enabling`

Valid Example

```yaml
info:
  title: Test Results Api
  x-value-chain: detect
```

3.2.11 MUST have info version

The `info` object **MUST** have a `version` property that follows semantic rules to distinguish API versions.

Invalid Example

```
info:
  title: ...
  description: ...
  version: 1
  <...>
```

Valid Example

```
info:
  title: ...
  description: ...
  version: 1.1.0
  ...
```

Zalando Guideline 218 and Zalando Guideline 116

3.2.12 MUST NOT define request body for `GET` requests

A `GET` request **MUST NOT** accept a request body.

Defining a request body on a `HTTP` `GET` is frowned upon due to the confusion that comes from unspecified behaviour in the HTTP specification.

Invalid Example

```
paths:
  /results/{resultId}:
    get:
      summary: Get a specific test result
      description: Get a specific test result.
      operationId: getResult
      tags:
        - results
      requestBody:
        content:
          application/json:
            ...
```

3.2.13 MUST NOT use http basic authentication

APIs **MUST NOT** use `HTTP` Basic Authentication.

HTTP Basic is an inherently insecure way to pass credentials to the API. They're placed in the URL in base64 which can be decrypted easily. Even if you're using a token, there are far better ways to handle passing tokens to an API which are less likely to leak.

See OWASP advice.

Invalid Example

```
...
components:
  securitySchemes:
    basicAuth:
      type: http
      scheme: basic
...
security:
  - basicAuth: []
```

UKHSA Guidelines Security

3.2.14 MUST NOT use uri versioning

Path **MUST** not contain versions.

Invalid Example

```
/user/v2:
```

Valid Example

```
/user:
```

Zalando Guideline 115

3.2.15 MUST return 200 for api root

Root path **MUST** define a `200` response.

Valid Example

```yaml
paths:
  /:
    get:
      summary: Get API information.
      description: Get API information.
      operationId: getApiInfo
      tags:
        - API Meta Information
      responses:
        '200':
          description: This response returns a information about the API.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ApiInfo'
        default:
          $ref: '#/components/responses/UnexpectedError'
```

UKHSA Guidelines Versioning

3.2.16 MUST specify default response

Each `operation` **MUST** include a default error response that combines multiple errors.

Invalid Example

The example below contains only a `200` response.

```
responses:
   ...
get:
  summary: Get User Info by User ID
   tags: []
   responses:
     '200':
       description: OK
```

Valid Example

The example below contains a `200` response and a `default` response that references the `Problem` errors file.

```
responses:
   ...
get:
  summary: Get User Info by User ID
   tags: []
   responses:
     '200':
       description: OK
     default:
       description: User Not Found
       content:
         application/problem+json:
           schema:
             $ref: ../models/Problem.yaml
```

Zalando Guideline 151

3.2.17 MUST use camel case for property names

Property names **MUST** use camel-case strings that match this pattern: `^[a-z][a-z0-9]+(?:[A-Z][a-z0-9]+)*$`.

Name	Description
camel case	The first letter of the first word **MUST** begin with a lowercase letter, the first letter of each subsequent word **MUST** begin with a capital letter and **MUST NOT** contain any separators between words such as spaces or special characters such as hyphens or underscores.

Invalid Examples

```
CustomerNumber
Customer_Number
customer-number
```

Valid Examples

```
customerNumber
salesOrderNumber
billingAddress
```

UKHSA Guidelines Property Names

3.2.18 MUST use camel case for query parameters

Query parameters **MUST** use camel-case strings that match this pattern: `^[a-z][a-z0-9]+(?:[A-Z][a-z0-9]+)*$`.

Name	Description
camel case	The first letter of the first word **MUST** begin with a lowercase letter, the first letter of each subsequent word **MUST** begin with a capital letter and **MUST NOT** contain any separators between words such as spaces or special characters such as hyphens or underscores.

Invalid Examples

```
/product/v1/users?
max_results=10&StartIndex=20&OTHER_PARAM=thing&other_other_param=that
```

Valid Examples

```
/product/v1/users?maxResults=10&startIndex=20
```

UKHSA Guidelines Parameter Names

3.2.19 MUST use https protocol only

Servers **MUST** be `https` and no other protocol is allowed.

Invalid Example

```
servers:
  - url: http://azgw.api.ukhsa.gov.uk/detect/testing/v1
    ...
```

Valid Example

```
servers:
  - url: https://azgw.api.ukhsa.gov.uk/detect/testing/v1
    ...
```

UKHSA Guidelines Security

3.2.20 MUST use lowercase with hyphens for path segments

Path segments **MUST** use lowercase letters and hyphens to separate words.

Invalid Example

```
/BeachReport:
```

Valid Example

```
/beach-report:
```

UKHSA Guidelines Path Segments

Zalando Guideline 129

3.2.21 MUST use normalised paths

Path **MUST** start with a slash and **MUST NOT** end with a slash (except root path `/`).

Invalid Example

```
paths:
  /patient/:
    ...
  /patient/{patientId}/results/:
```

Valid Example

```
paths:
  /:
    ...
  /patient:
    ...
  /patient/{patientId}/results:
```

Zalando Guideline 136

3.2.22 MUST use normalized paths without empty path segments

Path segments **MUST** not contain duplicate slashes.

Invalid Example

```
paths:
  /user//report:
```

Valid Example

```
paths:
  /user-report:
```

Zalando Guideline 136

3.2.23 MUST use problem json as default response

The content type for the default response **MUST** be `application/problem+json`.

Invalid Example

The default response in this example incorrectly uses `application/json` as the content type.

```
responses:
   ...
get:
  summary: Get User Info by User ID
   tags: []
   responses:
     ...
     default:
       description: ...
       content:
         application/json:
           schema:
             $ref: ../models/Problem.yaml
```

Valid Example

The default response in this example correctly uses `application/problem+json` as the content type.

```
responses:
   ...
get:
  summary: Get User Info by User ID
   tags: []
   responses:
     ...
     default:
       description: ...
       content:
         application/problem+json:
           schema:
             $ref: ../models/Problem.yaml
```

Zalando Guideline 151

3.2.24 MUST use problem json for errors

The content type for `4xx` and `5xx` status codes **MUST** be `application/problem+json`.

Invalid Example

The content type for the `503` response in this example incorrectly uses the `application/json` content type.

```
responses:
   '503':
     description: ...
     content:
       application/json:
         schema:
           $ref: ../models/Problem.yaml
```

Valid Example

The content type for the `503` response in this example correctly uses the `application/problem+json` content type.

```
responses:
   '503':
     description: ...
     content:
       application/problem+json:
         schema:
           $ref: ../models/Problem.yaml
```

Zalando Guideline 176

3.2.25 MUST use valid problem json schema

`Problem` schema **MUST** include this set of minimal required properties and validations:

Valid Example

```yaml
ProblemDetails:
  type: object
  description: Schema for detailed problem information.
  properties:
    type:
      type: string
      description: A URI reference that identifies the problem type.
      format: uri-reference
      maxLength: 1024
    status:
      type: integer
      description: The HTTP status code generated by the origin server for
this occurrence of the problem.
      format: int32
      minimum: 100
      maximum: 599
    title:
      type: string
      description: A short, human-readable summary of the problem type. It
should not change from occurrence to occurrence of the problem, except for
purposes of localization.
      maxLength: 1024
    detail:
      type: string
      description: A human-readable explanation specific to this
occurrence of the problem.
      maxLength: 4096
    instance:
      type: string
      description: A URI reference that identifies the specific occurrence
of the problem. It may or may not yield further information if
dereferenced.
      maxLength: 1024
required:
  - type
  - status
  - title
  - detail
  - instance
```

Zalando Guideline 176

3.2.26 MUST use valid version info schema

`ApiInfo` schema **MUST** include this set of minimal required properties and validations:

Valid Example

```yaml
components:
  schema:
    ...
    ApiInfo:
      type: object
      description: Schema for detailing API information.
      properties:
        name:
          type: string
          description: The name of the API.
          example: Test Results API
        version:
          type: string
          pattern: '^(0|[1-9]\d*)\.(0|[1-9]\d*)\.(0|[1-9]\d*)(?:-((?:0|[1-
9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-
Z-]*))*))?(?:\+([0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?$'
          description: The version of the API.
          example: 1.0.0
        releaseDate:
          type: string
          format: date
          description: The release date of this API version.
          example: 2025-02-26
        documentation:
          type: string
          format: uri
          description: A URL to the API documentation.
          example:
https://developer.ukhsa.gov.uk/namespace/product/v1/docs
        releaseNotes:
          type: string
          format: uri
          description: A URL to the API release notes.
          example:
https://developer.ukhsa.gov.uk/namespace/product/v1/releaseNotes
      required:
        - name
        - version
        - releaseDate
        - documentation
        - releaseNotes
```

3.3 SHOULD

3.3.1 SHOULD always return json objects as top level data structures

The top-level data structure for a `request body` or `response body` **SHOULD** be an object.

Invalid Example

```yaml
requestBody:
  content:
    application/json:
      schema:
      type: array
      items:
        type: string
```

Valid Example

```yaml
requestBody:
  content:
    application/json:
      schema:
      type: object
      properties:
        firstName:
          type: string
        lastName:
          type: string
```

UKHSA Guidelines API Design

Zalando Guideline 210

3.3.2 SHOULD declare enum values using upper snake case format

`enum` and `x-extensible-enum` values **SHOULD** be in UPPER_SNAKE_CASE format.

Invalid Example

```
schema:
  measurement:
   type: string
     x-extensible-enum:
       - Standard
       - Metric
       - Imperial
       - Non-standard
```

Valid Example

```
schema:
  measurement:
   type: string
     x-extensible-enum:
       - STANDARD
       - METRIC
       - IMPERIAL
       - NON_STANDARD
```

Zalando Guideline 240

3.3.3 SHOULD define api root

APIs **SHOULD** have a root path (`/`) defined.

Good documentation is always welcome, but API consumers **SHOULD** be able to get a pretty long way through interaction with the API alone. They **SHOULD** at least know they're looking at the right place instead of getting a 404 or random 500 error as is common in some APIs.

Valid Example

```yaml
paths:
  /:
    get:
      summary: Get API information.
      description: Get API information.
      operationId: getApiInfo
      tags:
        - API Meta Information
      responses:
        '200':
          description: This response returns a information about the API.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ApiInfo'
        default:
          $ref: '#/components/responses/UnexpectedError'
```

UKHSA Guidelines Versioning

3.3.4 should have location header in 201 response

`201 Created` responses to `POST` methods **SHOULD** have a `Location` header identifying the location of the newly created resource.

See RFC9110 Section 10.2.2 for more information on the `Location` header.

Valid Example

```
paths:
  /results:
    get:
    ...
    post:
      summary: Submit a new test result
      description: Submit a new test result.
      operationId: submitResult
      tags:
        - Test Results
      requestBody:
      ...
      responses:
        '201':
          description: This response returns a JSON object containing the
test result data.
          headers:
            Location:
            description: The URL of the created test result.
            schema:
              type: string
              format: uri
              example:
https://azgw.api.ukhsa.gov.uk/detect/testing/v1/results/de750613-ef3c-
4f5d-8148-10308b91896c
      ...
```

UKHSA Guidlelines API Design

3.3.5 SHOULD limit number of resource types

Resource types (root URL paths) **SHOULD** be limited to no more than eight.

Zalando Guideline 146

3.3.6 SHOULD limit number of sub resource levels

Path **SHOULD** contain no more than 3 sub-resources (nested resources with non-root URL paths).

Invalid Example

```
/users/location/name/address/email:
```

Valid Example

```
/users/{userId}/{name}:
```

Zalando Guideline 147

3.3.7 SHOULD prefer standard media type names

Response content **SHOULD** use a standard media type `application/json` or `application/problem+json` (required for problem schemas).

Invalid Example

```yaml
'204':
  description: No Content
  content:
    application/xml:
      schema:
        type: object
        properties:
          name:
            type: string
          url:
            type: string
            format: uri-reference
```

Valid Example

```yaml
'204':
  description: No Conten
  content:
    application/json:
      schema:
        type: object
        properties:
          name:
            type: string
          url:
            type: string
            format: uri-reference
```

Zalando Guideline 172

3.3.8 SHOULD support `application/json` content request body

Every request **SHOULD** support at least `application/json` media type.

Valid Example

```yaml
paths:
  /results:
    ...
    post:
      summary: Submit a new test result
      description: Submit a new test result.
      operationId: submitResult
      tags:
        - results
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ResultRequest'
```

3.3.9 SHOULD use hyphenated pascal case for header parameters

Header parameters **SHOULD** use hyphenated Pascal case.

Name	Description
hyphenated pascal	Each word **MUST** begin with a capital letter, and be separated by a hyphen.

Invalid Example

```
parameters:
- schema:
    type: string
    in: header
    name: PascalCaseHeader
```

Valid Example

```
parameters:
- schema:
    type: string
    in: header
    name: Pascal-Case-Header
```

Zalando Guideline 132

3.3.10 SHOULD use standard http status codes

`response` **SHOULD** use standard HTTP status codes.

Invalid Example

`Error-500` is not a valid HTTP status code.

```
/weather:
  get:
    responses:
      'Error-500':
        description: Internal Server Error
```

Valid Example

`500` is a valid HTTP status code.

```
/weather:
  get:
    responses:
      '500':
        description: Internal Server Error
```

Zalando Guideline 150

3.3.11 SHOULD use `x-extensible-enum`

`enum` values **SHOULD** use the marker `x-extensible-enum` rather than `enum`.

Invalid Example

```yaml
deliveryMethods:
  type: string
  enum:
    - PARCEL
    - LETTER
    - EMAIL
```

Valid Example

```yaml
deliveryMethods:
  type: string
  x-extensible-enum:
    - PARCEL
    - LETTER
    - EMAIL
```

Zalando Guideline 112