DCSA API Design Principles 1.0

September 2020

Table of contents

Tables

Figures

Change history

Rev	Issue	Contributors	Description
1.0	September 2020	DCSA	First release

Terms, acronyms, and abbreviations

Term	Definition
API	Application programming interface
camelCase	In this document camelCase is defined as **lower camel case**. Definition of lower camel case is that initial letter is lower and every subsequent word is Capitalized.
CRUD	Create, Read, Update and Delete (equivalent to POST, GET, PUT/PATCH and DELETE rest operations)
FK	Foreign Key
HATEOAS	Hypermedia as the Engine of Application State
kebab-case	kebab-case combines words by replacing each space with a dash (-). All letters are lower case. For example, this-is-a-kebab-case-example.
PK	Primary Key
UUID	Universally unique identifier

1 Introduction

This chapter describes the purpose, structure and conventions used in this document.

1.1 Purpose and scope

The purpose of this document is to establish guidelines on how APIs should be designed, developed and implemented in the digital container shipping context. The guidelines describe the requirements that DCSA-compliant APIs should fulfil.

1.2 API characteristics

As described in the table below, a good API should have the following quality characteristics or have a balance of these quality characteristics.

Index	Characteristics	Descriptions
1	Suitable	APIs provide a well-defined service that optimizes value for consumers by having clearly defined responsibilities and an appropriate level of abstraction.
2	Maintainable	It is easy to understand how to interact with the API and development of maintainable code is facilitated.
3	Stable	The occurrence and impact of major changes are minimized.
4	Secure	Measures and practices are implemented to prevent abuse.
5	Performant	The performance consequences of API design are considered, monitored and optimized for common interactions.
6	Well-documented	Up-to-date, complete and easy-to-read documentation is available to facilitate adoption.

Table 1: API quality characteristics

The structure of this documents follows the order of the API quality characteristics mentioned in the table above.

1.3 Conventions

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 RFC 2119 (https://www.ietf.org/rfc/rfc2119.txt).

2 Suitable

Good APIs are suitable regarding their functionality and responsibilities. Suitable APIs provide a well-defined service that optimizes value for consumers by having clearly defined responsibilities and an appropriate level of abstraction.

To achieve this objective, two requirements are identified below:

- The API is constructed with the consumer in mind and **SHOULD** be based on User Stories.
- The API is constructed according to a minimalistic approach – less is more.

3 Maintainable

APIs need to be maintainable. This means it should be easy to understand how to interact with the APIs, and development of maintainable code should be facilitated. The requirements to achieve good maintainability are described in this chapter.

3.1 JSON

Requests and responses **MUST** be in The application/json Media Type for JavaScript Object Notation (JSON).

Requests and responses **MAY** be in other formats as well – e.g. XML Media Types.

3.2 URLs

MUST point to a resource (entity, process).

MUST consist of nouns and **MUST NOT** be actions. HTTP verbs (GET, PUT, PATCH, POST, DELETE) **MUST** be used for actions.

"/" **MUST** be used to indicate hierarchical relations.

Kebab-case **MUST** be used in URLs. Path parameters **MUST** be consistent with property names. Path parameters referring to property names **MUST** use camelCase.

Query parameters **MUST** use camelCase.

3.3 Collections

Collection items **MUST NOT** consist of composite keys. A unique key **MUST** identify a collection element. Collections **SHOULD** be pluralized.

3.4 Sorting

Sorting **SHOULD** be limited to specific fields. The parameter **MUST** be restricted to the values ASC (ascending order) and DESC (descending order) describing the sort direction. The direction **MAY** be omitted. If omitted ascending order **MUST** be used. A colon : is used to separate the field to sort by and the direction. Multiple fields can be used, in which case each field is separated by a comma,. The following table shows some examples:

Query Parameter	Explanation
sort=name	Sort by the name field in ascending (default) order.
sort=name:desc	Sort by the name field in descending order.
sort=name:asc	Sort by the name field in ascending order.
sort=name,age:desc	Sort first by the name field in ascending order. If two equal names exist, then sort those names by age in descending order.

Table 2: Examples of sorting

The sort parameter is optional and if not specified, the order is set by the implementer.

It is up to the implementer to find a suitable Collator to use when sorting on String values.

3.5 Pagination

GET requests on collection results SHOULD implement pagination. Links to current, previous, next, first and last page SHOULD be available in the response headers; more links MAY be present.

Link	Can Be Null	Explanation	Example
First-Page	Yes	A link to the first page. First-Page header link MAY be omitted if current page is the first page.	`<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="First-Page"`
Previous-Page	Yes	A link to the previous page. Previous-Page header link MAY be omitted if the current page is the first page.	`<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Previous-Page"`

Link	Can Be Null	Explanation	Example
Next-Page	Yes	A link to the next page. Next-Page header link MAY be omitted if the current page is the last page.	`<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Next-Page"`
Last-Page	Yes	A link to the last page. Last-Page header link MAY be omitted if the current page is the last page.	`<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Last-Page"`
Current-Page	No	A link to the current page.	`<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Current-Page"`

Table 3: Pagination

The default page size SHOULD be 100, if it is not specified on the endpoint. The default page size MAY be changed per endpoint. If the response payload is large, the page size SHOULD be changed to a smaller, more suitable number. The consumer SHOULD be able to override the default page size via the limit-parameter.

Keyset-based pagination MUST be used. The server generates all links to pages relative to the current page requested. It is not possible for the consumer to request a specific page – the consumer MUST select a page provided by the server. The server MUST provide links to the previous and next page (if possible) and SHOULD provide links to first and last page.

If the filter and/or sorting used when getting a collection result is changed, the pagination MUST be reset to first page.

It is important to note that the pagination represents a cut of the data at a specific moment in time. If somebody else triggers the same search a minute later, they might get a different data set for the same filtering/sorting criteria.

3.6 Property names

Property names on objects MUST be in camelCase.

Property names containing arrays SHOULD be plural.

Property names MUST NOT include FK (Foreign Key) or PK (Primary Key), as this exposes database design.

Boolean properties MUST be prefixed by either **is** or **has**.

3.7 Enum values

Enum values SHOULD be declared using UPPER_SNAKE_CASE.

3.8 Arrays

Empty arrays MUST NOT be represented with null – but MUST be empty lists [].

3.9 Date and Time properties

Date properties MUST be suffixed with *"Date"*. A Date property only contains a date in YYYY-MM-DD format.

Time properties MUST be suffixed with *"Time"*. A Time property only contains a specific time of a day in 24-hour-format using the following format: `hh:mm`. Time format MAY include seconds in which case the following is added `:ss`.

DateTime properties MUST be suffixed with *"DateTime"*. A DateTime property contains both a date and time.

Date, Time and DateTime MUST be presented using the ISO 8601 format.

Example	Explanation
2020-07-13	13th of July 2020
The following 5 timestamps represent the same time period in different Time Zones:	
2020-07-05T02:15-04:00	5th of July 2020, 02:15 in New York (EDT Eastern Daylight Time)
2020-07-05T08:15+02:00	5th of July 2020, 08.15 in Rotterdam, Netherlands (CEST Central European Summer Time)
2020-07-05T11:45+05:30	5th of July 2020, 11.45 in New Delhi, India (IST India Standard Time)
2020-07-05T14:15+08:00	5th of July 2020, 14.15 Shanghai, China (CST China Standard Time)
2020-07-04T23:15-07:00	4th of July 2020, 23.15 San Francisco (PDT Pacific Daylight Time)

Table 4: Examples of time format

3.10 UTF-8

Encoding MUST be UTF-8.

3.11 Query parameters

The following table shows naming conventions to be used:

Query Parameter	Description	Example
limit	Limits the number of objects returned in a collection-response. In responses that contain a lot of data per object, this might be lower. Any number > 0 can be used. *Default is 100 (unless otherwise specified).*	limit=20
cursor	A server-generated pointer to a page in a collection-response. This is never generated by the consumer. In collection-responses, predefined header links to next and previous page **MUST** be added and links to first and last **SHOULD** be added. These links are populated with a *cursor* query parameter. The *cursor* **MUST NOT** be modified by the consumer. If no *cursor* is set – then first page is returned.	cursor=ABC123
sort	A comma-separated list of field names to define the sort order. Field names **SHOULD** be suffixed by a colon (:) followed by either the keyword ASC (for ascending order) or DESC (for descending order) to specify direction. :ASC **MAY** be omitted, in which case ascending order will be used.	sort=name sort=name:ASC sort=name:DESC

Table 5: Query parameters naming conventions

3.12 Custom headers

Custom headers **MUST NOT** use the prefix *x-*. Please see [Deprecating the "X-" Prefix and Similar Constructs in Application Protocols.]

The following table contains the agreed DCSA header:

Header	Description
API-Version	*In requests:* used to specify the version of the contract (API version) requested by the consumer. *In responses:* used to indicate the version of the contract (API version) returned by the implementer.

Table 6: Custom headers Binary data

3.13 Binary data

When the payload is only binary data, it **MUST NOT** be encoded. The format returned will be made available during content negotiation.

When binary data is part of the payload (in a property), it **MUST** be Base64 encoded.

3.14 Error handling

Standard error codes **MUST** be used. The table below extends the standard http error codes with extra information. The extended error codes **SHOULD** be used together with standard error codes for easier bug finding during development and maintenance. This is not an exhaustive list; other codes may be used in implementations.

HTTP code	Error code	Error message
400 – Bad Request	invalidParameter	An invalid parameter or parameter value was supplied in the request.
400 – Bad Request	missingParameter	The API request is missing a required parameter.
400 – Bad Request	invalidQuery	The request query was invalid. Check the documentation to ensure that the supplied parameters are supported, and check if the request contains an invalid combination of parameters, or an invalid value.
400 – Bad Request	invalidUri	The requested URI is not represented on the server.

HTTP code	Error code	Error message
401 – Unauthorized	missingCredentials	The user is not authorized to make the request.
401 – Unauthorized	invalidCredentials	The supplied authorization credentials for the request are invalid.
401 – Unauthorized	expiredAccessToken	The supplied Access Token has expired.
403 – Forbidden	accessDenied	The requested operation is forbidden.
403 – Forbidden	insufficientPermissions	The authenticated user is not permitted to execute this request.
404 – Not found	notFound	The requested resource could not be found. 404 Not found MAY be used to mask sensitive information for both 400, 401 and 403.
405 – Method not allowed	httpMethodNotAllowed	The HTTP method for the request is not supported.
409 - Conflict	ResourceAlreadyExists	The resource already exists. Used when a "Unique constrain violation" occurs.
429 – Too many requests	rateLimitExceeded	Too many requests have been sent recently. A Retry-After header SHOULD be added to notify consumer when to try again.
500 – Internal server error	internalError	The request failed due to an internal error.

HTTP code	Error code	Error message
503 – Service Unavailable		Used for maintenance purposes A Retry-After header SHOULD be added to notify consumer when to try again.

Table5: Error handling

404 Not Found MAY be returned instead of *400 Bad Request*, *401 Unauthorized* or *403 Forbidden* for sensitive information. This can be used when the implementer of the API does not want to verify if restricted information is available.

Requests rejected before reaching the API (rejected at Gateway level) MAY omit the error object.

3.15 Subscription model

A subscription model is based on an event-driven architecture. Whenever an event occurs, the event is sent (pushed) to relevant recipients (subscribers). At any point in time, it MUST be possible for a recipient to query the server, to get (pull) the event again.

The subscription model is based on a callBack URL where the publisher POSTs new events. The publisher does not need to be authenticated (logged in) on the subscriber system prior to sending events, the authenticity and integrity of the message will be done through a Signature.

When the subscriber queries the publisher endpoints, an authentication process is required – see section 5.1 Endpoints.

For every *push* endpoint, an equivalent *pull* endpoint MUST exist.

Pull endpoints retrieve existing (historical) events. *Existing events* in this context means events that have already been processed or sent.

A Publishing service (referred to as Push endpoint in this document) sends new events when they are Created. *New events* in this context means events that have not yet been sent by the publisher.

The payload for the *pull* endpoint MUST be aligned with the payload of the *push* endpoint.

3.15.1 Subscription management

For events to be pushed by the publisher, the consumer/subscriber MUST create a subscription. Normal CRUD operations MUST be used in order to manage subscriptions.

A GET request on the subscription endpoint lists all available subscriptions.

A GET request with an ID lists a specific subscription.

A POST request creates a new subscription.

A PUT or PATCH request with an ID updates a subscription.

A DELETE request with an ID removes a subscription.

A subscription MUST contain the following 3 parameters: an ID, a callback and a secret. The ID is provided by the publisher and MUST be a UUID; the callback and the secret are provided by the subscriber. The callback is the endpoint the publisher MUST publish (POST) new events to. The secret MUST be a shared secret. Both the subscriber and publisher MUST store the secret together with the ID for later validation and signing of events.

A publisher MUST NOT send the secret in subsequent responses. When sending a subscription – the secret field in the subscription MUST be Null or omitted. When sending a subscription (via a GET request) - the publisher MUST sign the response by adding a Signature.

A subscription MAY contain 0 or more event filters. Events sent MUST fulfil the filters; if no filters are present, all events are sent.

Upon subscription creation, the publisher SHOULD validate the callback endpoint; this can be done by calling the endpoint with a HEAD request to verify the existence of the endpoint. If the endpoint fails, the creation of the subscription MUST fail.

Once a subscription is created, the publisher of the API MUST use the callback endpoint to publish events relevant to the subscriber. The publisher MUST use POST to publish the event on the callback endpoint. The callback endpoint SHOULD specify HTTPS.

The secret is a string used to encrypt a Signature for signing events. The Signature is used to verify the authenticity of the message. The secret SHOULD be sufficiently long to be cryptographically secure. RECOMMENDED size is at least a random 32 character-length hex string.

A secret SHOULD be updated every 3 months. It is the responsibility of the subscriber to update the secret.

3.15.2 Retry policy

When receiving an event, the consumer MUST acknowledge with an HTTP 2xx. If the request was not a success (2xx), the publisher MUST retry sending the event. If a Retry-After header is present in the response – this SHOULD be honored. Otherwise, it is RECOMMENDED that an exponential back-off retry policy is used. An example of this could be to retry after 1 min, 2 min, 4 min, 8 min, etc. It is up to the implementer to decide how long the retry period should last.

Every time a publisher needs to retry sending an event, the publisher MUST make sure the secret used is updated from the subscription. Cached subscriptions MUST not be used since the secret MAY have been updated since last retry.

3.15.3 Verification

When a subscriber receives an event on a callback endpoint, the event MUST be validated. The validation is done by creating a signature following the steps described in section 5.2 Signature used for the subscription model and comparing it to the signature in the event. If the signatures

do not match – the consumer SHOULD query the subscription endpoint and verify that the secret has not been updated by the publisher.

- If the publisher has changed the secret (can be verified by looking at the Signature-header in the subscription response), the subscriber MUST update the subscription with a new secret.
- If the secret has not been updated, the event MUST be discarded with a 400 Bad Request.

4 Stable

APIs need to be stable over time, which means the occurrence and impact of major changes are minimized. The requirements to achieve stability are described in this chapter.

4.1 Versioning

Semantic Versioning 2.0 MUST be used to specify version information restricted to the format *<MAJOR>.<MINOR>.<PATCH>.*

- *MAJOR* to be increased when backward compatibility is broken.
- *MINOR* to be increased when new functionality (e.g. new operations, new optional fields) is added in a backward-compatible manner
- *PATCH* to be increased when backward-compatible bug fixes are made. PATCH does not include new functionality.

Pre-releases (rule 9 from Sematic Versioning) and **build metadata** (rule 10) MUST NOT be used in API version information.

URI versioning MUST be used. Only <MAJOR> version is included – example: */v2/events. First version* SHOULD *include /v1/.* An *API-Version* custom header MAY be added to the request; if added it MUST only contain <MAJOR> version. *API-Version* header MUST be aligned with the URI version.

A custom header called *API-Version* MUST be added in the response-specifying version. The *API-Version* includes all three formats – example: *API-Version: 2.0.0.*

4.2 Backward compatibility

Backward compatibility MUST NOT be broken within a major release.

When adding new features, the following rules SHOULD be followed:

- Preferably add optional and not mandatory fields
- Never change the semantics of a field
- Enum range can be reduced (server needs to be able to handle old values)
- Enum range can be expanded but only when used for input parameters

Major versioning SHOULD be avoided:

- Create a new resource (a variant of the old) in addition to the old
- Create a new service endpoint (Duplicate and Deprecate: add Deprecation header to old endpoint; eventually add Sunset headers)
- Create a new API version in parallel with the old

Clients of the API SHOULD be robust:

- Be conservative with API requests and data passed as input
- Be tolerant with unknown fields in the payload, but do not eliminate them from payload if needed for subsequent PUT requests

Implementors of the API MUST NOT be more than 1 major version behind the latest version. No more than 3 parallel major versions SHOULD be available.

4.3 Deprecation

Deprecated endpoints MUST be documented in OpenAPI specification using the "Deprecated" property introduced in OpenAPI 3.0.

Deprecated endpoints SHOULD add Deprecation and Sunset headers to responses. See The Deprecation Header Field and The Sunset http Response Header Field.

A Link header providing additional information SHOULD be added in combination with the Deprecation header. If added, the link provided MUST point to the documentation for additional information.

Communication SHOULD be sent to consumers of deprecated endpoints where possible.

5 Secure

APIs need to be secure, which means measures and practices are implemented to prevent abuse. The requirements to achieve this objective are described in this chapter.

5.1 Endpoints

All endpoints MUST be secured. Security used SHOULD be OAuth2. Other security schemes MAY be used. HTTPS MUST be used.

5.2 Signature used for the subscription model

For a consumer to verify the authenticity of a message, the publisher MUST include a Signature header in the request. The Signature MUST match The 'Signature' HTTP Header. A Signature header uses the following pattern:

```
Signature: keyId=<subscription-id>, created=1402170695, headers="(request-target)
(created) date host", signature=<signature-string>
```

The Signature MUST contain:

- keyId=<subscriptionID>
- created=<Unix timestamp>
- headers="(request-target) (created) date host"
- signature=<Base64 encoded signature>

In addition, the Signature **MAY** contain extra header values such as:

- digest
- Content-length

keyId is the identifier for the key used to create the signature. The keyId **MUST** be the *subscriptionID*.

created is a Unix timestamp integer value.

headers is a list of headers to include in the signature. The headers **MUST** include the following values:

- (request-target) – refers to the request target endpoint (the callback in the subscription). The value is constructed by using the HTTP method in lowercase, a space " ", and then the path.
- (created) – refers to when the signature header was created and is a Unix-timestamp.
- date – refers to the date header of the request.
- host – refers to the host header of the request.

Other header values **MAY** be included.

signature is the signature of the request. It is constructed by creating a Signature String. The *Signature String* is encrypted using HMAC-SHA256 together with the secret from the subscription. The result of the encryption is then Base64 encoded.

6 Performant

The performance consequences of API design need to be considered, monitored and optimized for common interactions.

To achieve this objective, two requirements are identified below:

- Circuit Breaker Pattern **SHOULD** be used for fast fail. If the implementer of the API has internal errors or a database that is not responding - it would be a good idea to add a Circuit Breaker Pattern in order for the endpoint to respond quickly, instead of waiting for Timeout.
- Rate Limiting **SHOULD** be used to prevent too many requests.

7 Well-documented

Up-to-date, complete and easy-to-read documentation needs to be available to facilitate adoption. The requirements to achieve this objective are described in this chapter.

7.1 General documentation

Open API 3.0.x **MUST** be used for documenting endpoints. British English **MUST** be used.

SwaggerHub **MUST** contain the latest specification for the API:

- DCSA OpenAPI specification for Operational Vessel Schedules v1.0.0
- DCSA OpenAPI specification for Track & Trace v2.0.0

GitHub **MUST** contain the latest documents and links to the latest documents:
https://github.com/dcsaorg/DCSA-OpenAPI

7.2 HATEOAS

Where possible, header links **SHOULD** be added to discover relations to the resource requested.

Appendix A: Sequence diagrams of Subscription model

This appendix shows the flows for different scenarios:

- Subscriber creates a subscription at a publisher – Figure 1
- Publisher publishes an event to a subscriber – Figure 2
- Publisher updates (deletes) shared secret key and sends an event – Figure 3
- Subscriber updates shared secret key and receives an event – Figure 4
- Subscriber receives a fake event – Figure 5

Figure 1 shows the sequence when a subscriber creates a new subscription at the publisher. Both subscriber and publisher store SecretA (a shared secret) together with the subscriptionID 123 (which is a UUID). This is the starting scenario for the following sequences.

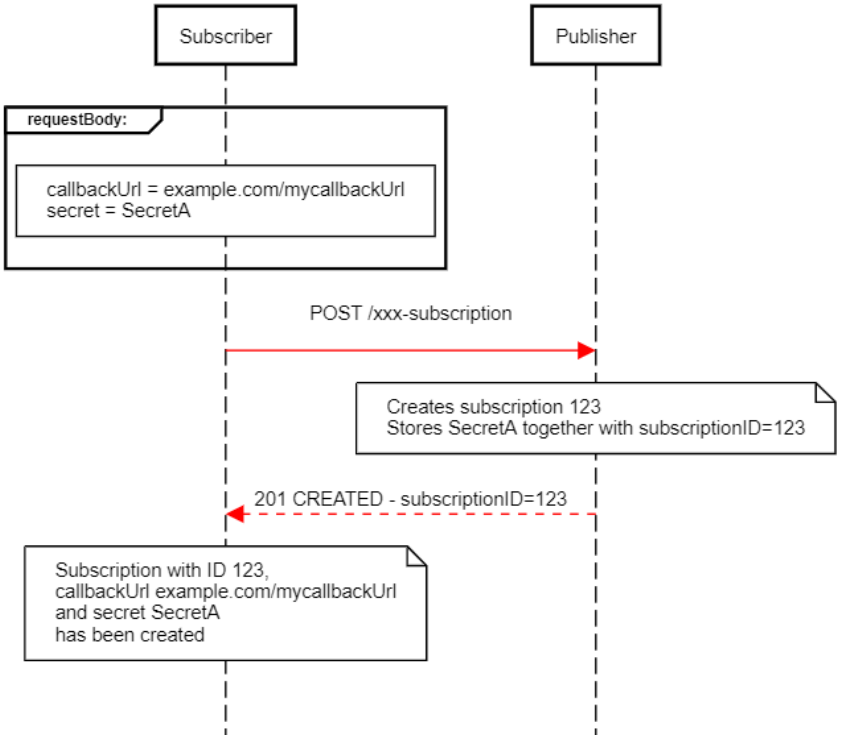

Figure 1: Creating a subscription

Figure 2 shows the sequence when an event is delivered successfully. A subscription exists with id=123; both subscriber and publisher have secret key = SecretA. In the sequence shown in Figure 2, the publisher publishes a new event. The subscriber validates the event against SecretA.

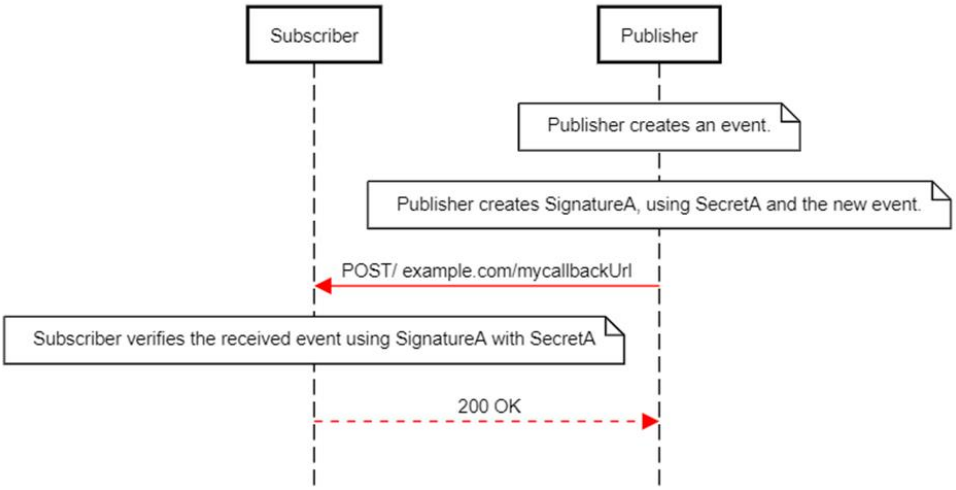

Figure 2: Publishing an event

Figure 3 shows the sequence when the publisher changes the secret key. A subscription exists with id=123; both subscriber and publisher have secret key = SecretA. In the sequence shown in Figure 3, the publisher updates the secret to SecretB and sends an event to the subscriber. The subscriber cannot validate the event and queries the subscription in order to verify the secret. The secret does not verify – so the subscriber knows that the secret has been changed. The subscriber now changes the secret to something new (SecretC) and fails the original message with a 400 Bad Request. The publisher will now retry sending the message with SecretC.

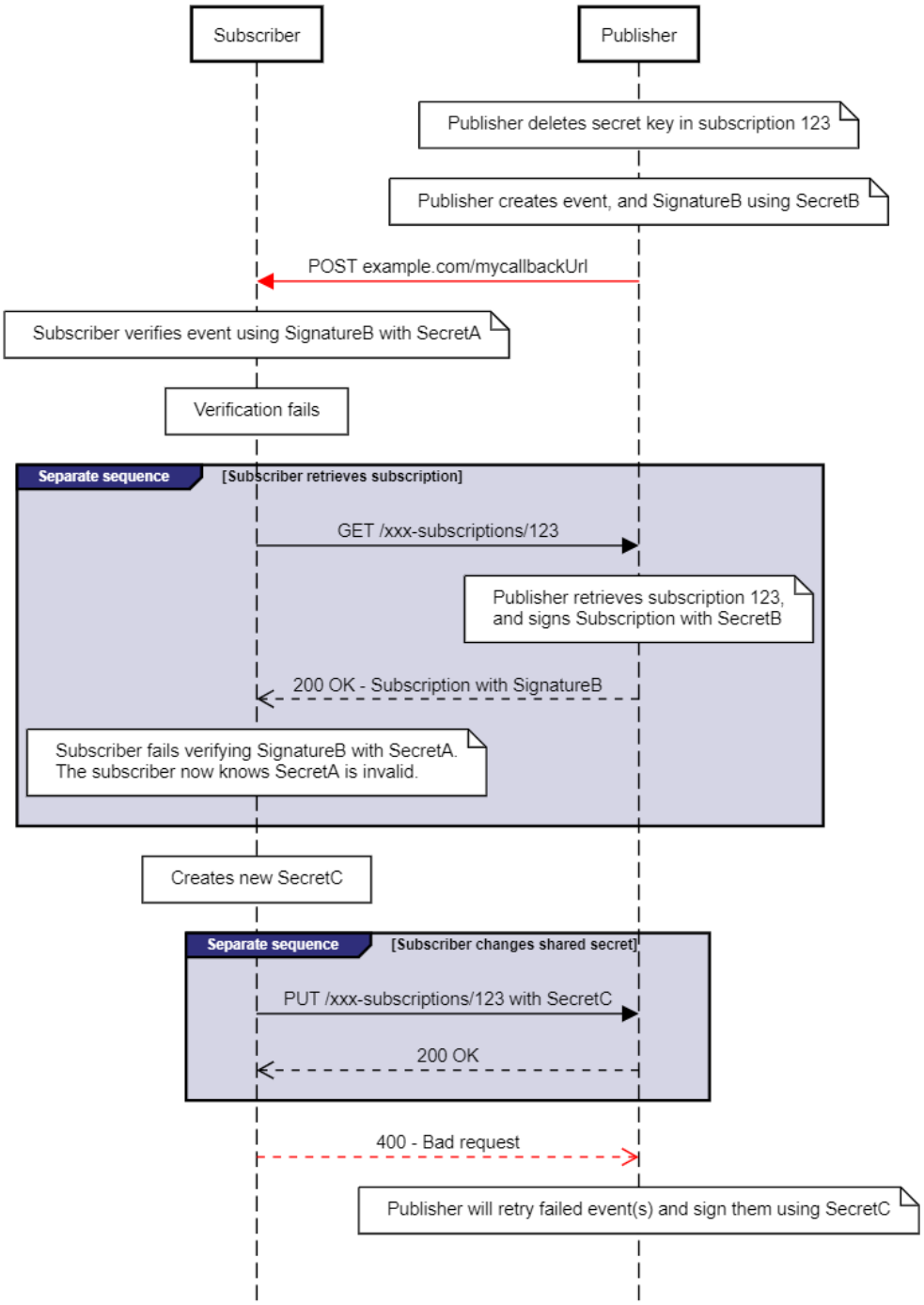

Figure 3: Publisher changes the secret key

Figure 4 shows the sequence when the subscriber changes the secret. A subscription exists with id=123; both subscriber and publisher have secret key = SecretA. In the sequence shown in Figure 4, the publisher creates an event and just before sending it to the subscriber – the subscriber changes the secret to SecretB. When the subscriber receives the event from the publisher, it will be signed with an "old" secret (SecretA). The subscriber fails the validation and needs to verify if the publisher has changed the secret. This is done by sending a GET /xxx-subscription/123 request to the publisher, the publisher sends the subscription and signs it with SecretB. The subscriber validates that SecretB is still valid and sends a 400 Bad request to the publisher in order for the publisher to retry sending the event. The publisher will now try to resend the event using the new SecretB.

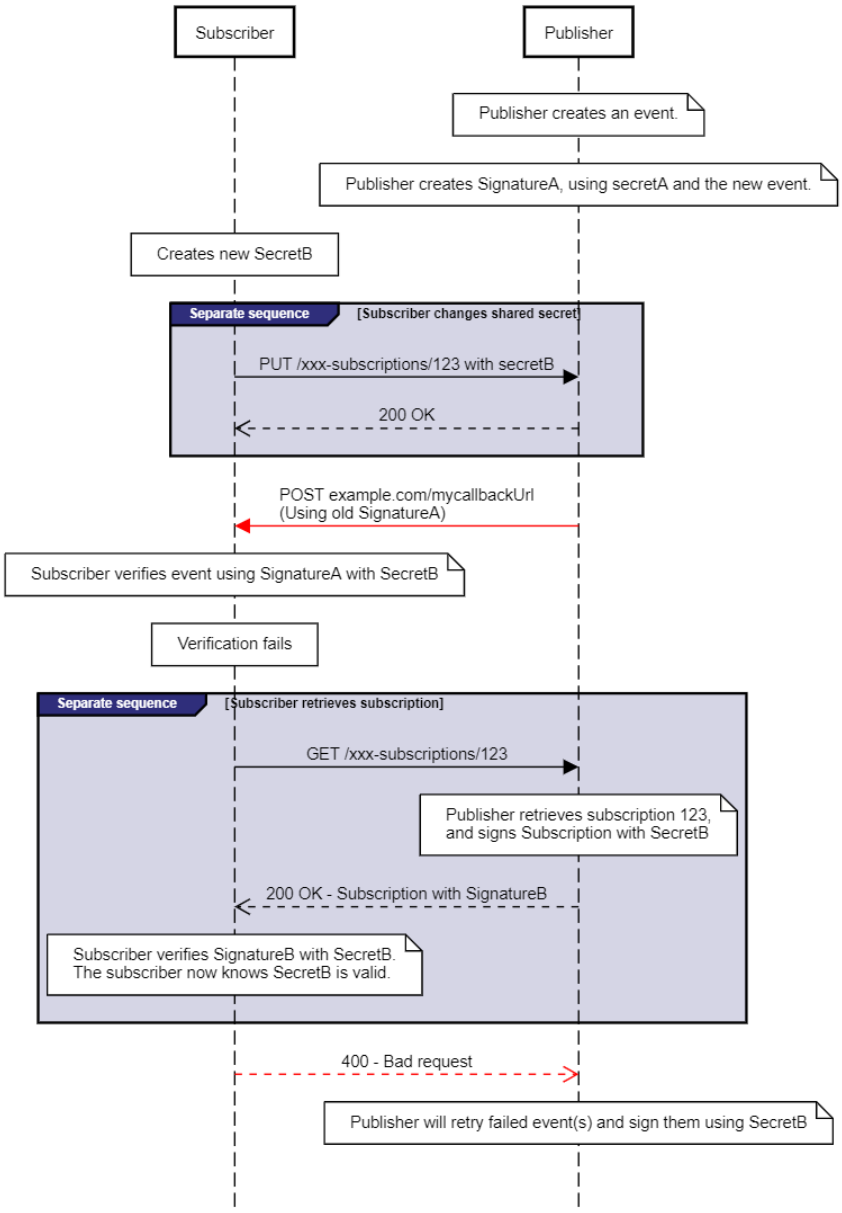

Figure 4: Subscriber changes the secret key

Figure 5 shows the sequence of a tampered event (an "illegal" publisher). A subscription exists with id=123; both subscriber and publisher have secret key = SecretA. In the sequence shown in Figure 5, the subscriber has SecretA connected with subscription 123. When the villain sends an event, pretending to be the publisher, the subscriber cannot verify the event. In order to know if the publisher might have changed the secret – the subscriber GETs subscription 123 from the publisher and verifies that SecretA is still valid. The subscriber now knows the event was sent by a villain.

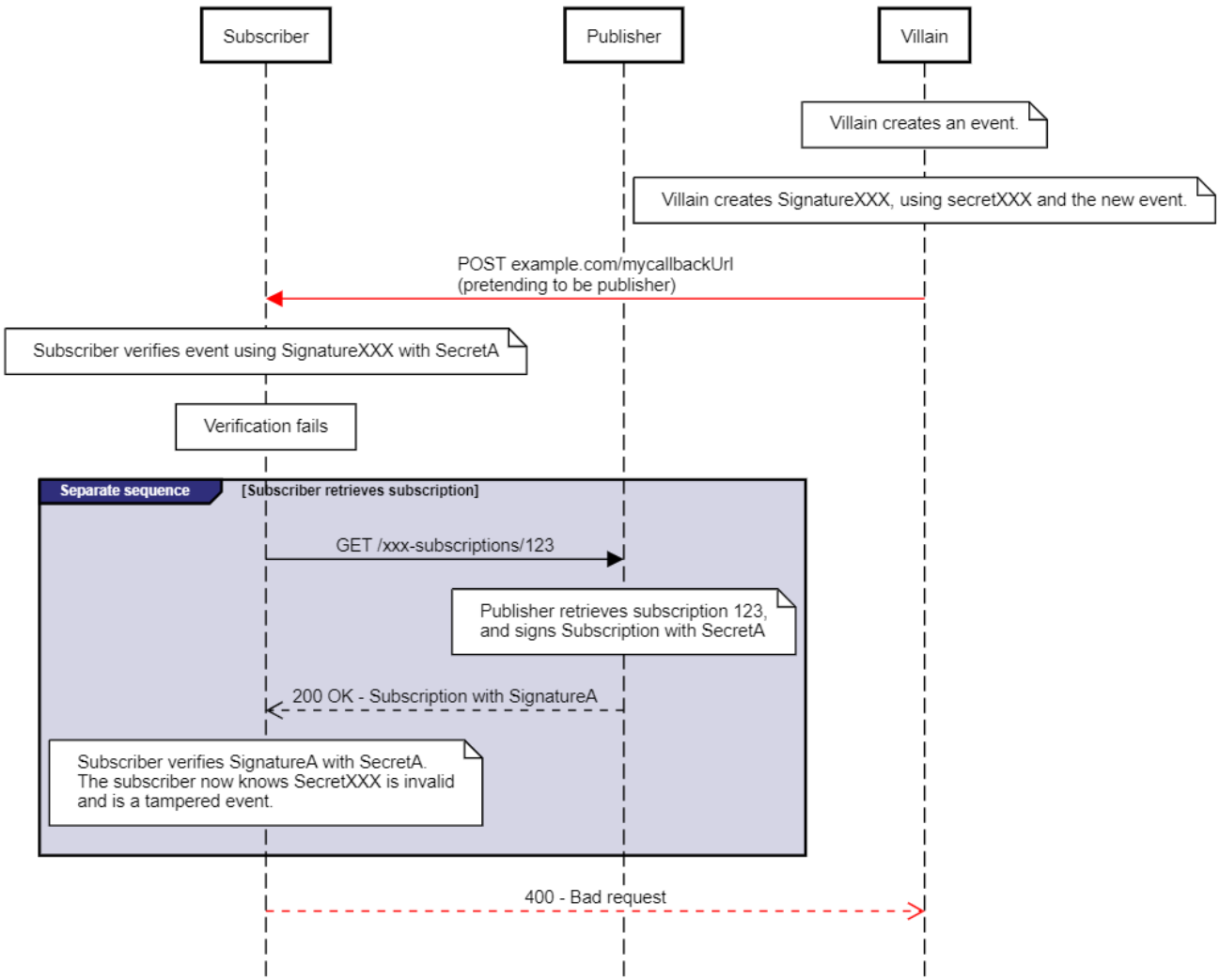

Figure 5: Tempered event