EUROPEAN COMMISSION

DIRECTORATE-GENERAL FOR INFORMATICS

ISA² IPS REST API Profile

Version 1.0

Publication date: 8 September 2022

Document Author(s)

Author Name	Role
Jerry Dimitriou	Solution Architect & Technical Expert

Document Reviewer(s)

Reviewer Name	Role
Vlad Veduta	eDelivery Business Analyst
Bogdan Dumitriu	eDelivery Project Officer

Document Approver(s)

Reviewer Name	Role
Bogdan Dumitriu	eDelivery Project Officer

This document was created as a deliverable in the 2020 ISA^2 Innovative Public Services (IPS) action.

TABLE OF CONTENTS

1 INTRODUCTION

The eDelivery building block is constantly evaluated by organisations across the EU as an option for projects that require secure and reliable communication.

An observation from the feedback collected is that, while the eDelivery AS4 profile is a good fit for many of the above-mentioned projects, several others, with a growing diversity of requirements, would welcome the extension of the eDelivery building block with a profile to cater for the REST API architectural style. What is common to these potential projects is that at least one party to the data exchange would operate in a **light context**, loosely defined as a set of constraints and circumstances applying to organisations or individuals looking to consume a business service with a relatively small technical footprint. Such constraints could be linked to non-availability of any combination of human, financial, knowledge, technical or power resources, whether at design, installation or run-time.

A profile for a harmonised use of the REST API architectural style is thus proposed here to respond to this need, as part of the 2020 ISA^2 Innovative Public Services (IPS) action. If successful, the profile could, as a next step, be added to the eDelivery building block.

2 Scope and Goals

The aim of this specification is to create a profile of several HTTP and REST specifications catering for the “light context”. The profile is built on top of the HTTP/REST architectural pattern and will adopt technology to support this pattern based on existing standards, including standardised specifications, IETF Draft RFCs, RFCs etc. Its scope is to provide:

- Standardised data exchange
- Business-agnostic data exchange
- Secure data exchange

Focusing on the light context, the profile would enable the implementation of eDelivery-compliant data exchange benefiting from:

- Ease of deployment/installation
- Economy of resources on the client side
- Operation on mobile/personal environments
- Updated options for the data exchange patterns

The specification profiles also aspects of an HTTP / REST API ecosystem that are not part of the implementation itself, such as:

- Delegated Authentication Patterns
- Delegated Authorization Patterns
- Lifecycle management
- API Discoverability

3 NOTATION

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [BCP14] ([RFC2119] and [RFC8174]) when, and only when, they appear in all capitals, as shown here.

4 DOCUMENT STRUCTURE

The document consists of three main sections and two profile enhancements. The main sections of the profile are the API Core Profile, the API Documentation and Messaging API Specification. The two profile enhancements are the High-Security Enhancement and the Discoverability Enhancement.

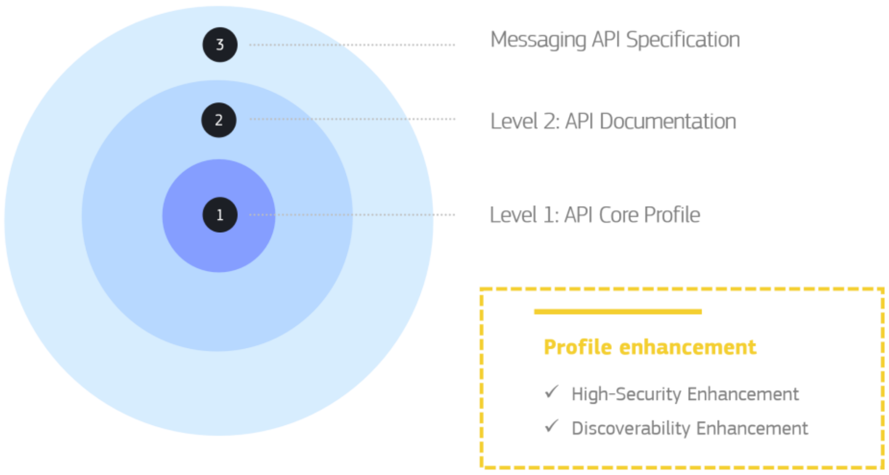
3
2
1
Messaging API Specification
Level 2: API Documentation
Level 1: API Core Profile
Profile enhancement
✓ High-Security Enhancement
✓ Discoverability Enhancement

Figure 1 Profile Structure

4.1 PROFILE SECTIONS

The ISA^2 IPS REST API profile consists of three hierarchical sections, each subsequent section depending on the preceding one. The API Core Profile (Level 1) is considered the baseline and mandatory part of the REST API Profile from a conformance perspective. The API Documentation goes one step further and defines how an OpenAPI Document should be structured for an API in order to be conformant (Level 2). An API conformant to Level 2 is an API that inherently conforms to Level 1 and also provides an OpenAPI Document according to the specification of Level 2. The final section provides a REST Messaging API Specification, itself conformant to Levels 1 and 2 of the profile, to be reused as a messaging specification in a light context, architecturally comparable to, but feature-different from the eDelivery AS4 profile [EDELIVERY-AS4-PROFILE].

4.1.1 Level 1: API Core Profile

The API Core Profile provides the baseline of the HTTP technologies that are profiled as part of the ISA^2 IPS REST API profile, covering the following major aspects of a REST API:

- Authentication
- Authorization
- Transport, message and payload level security
- Lifecycle management

- Common semantics on REST API design, resource representations, HTTP methods and error representations
- Documentation and discoverability

In the API Core Profile, several RFCs and Internet-Drafts from the IETF together with standards provided from organisations like W3C, ISA and ETSI are being selected and profiled to meet the requirements.

4.1.2 Level 2: API Documentation

Level 2 of the profile provides an extra layer of conformance, applicable for the documentation of the API. It profiles the OpenAPI Specification v3 and provides both predefined components that can be reused and extensions to the OpenAPI Specification for aspects not currently supported such as the lifecycle definition of an API.

4.1.3 Messaging API

The Messaging API Specification is an API specification that is based on both the API Core Profile and the API Documentation specifications and that makes use of the High-Security Enhancement of the profile. Its aim is to provide a common REST API Specification for messaging aligned with the scope and goals stated in the above section.

4.2 Profile Enhancements

Requirements expected to be common in many, but not all environments, are captured separately as Profile Enhancements. This avoids the need for all conformant implementations to be aligned to a highest common denominator, while still providing a standardised approach where necessary by allowing projects or organisations to mandate the use of the core profile sections in combination with a specific selection of enhancements.

The Profile Enhancements MUST be used in conjunction with the REST API Profile. The two currently defined Profile Enhancements are mutually compatible, so they MAY be used in combination.

4.2.1 High-Security Enhancement

The API Core Profile defines guidelines on how Message-Level Security and Payload Security MUST be implemented, if needed. The High-Security Enhancement makes their implementation mandatory, further restricting the algorithms that can be used.

4.2.2 Discoverability Enhancement

The API Core Profile defines a minimal set of guidelines on how APIs should support discoverability. The Discoverability Enhancement extends this set to mandate further discoverability guidelines. Thus, an API conforming to the Discoverability Enhancement MUST implement both the guidelines defined in the Discoverability section of the API Core Profile, by providing all the OpenAPI attributes prescribed therein, and the additional ones defined in the Discoverability Enhancement.

5 API Core Profile

This section defines the core parts of the profile.

5.1 Authentication and Authorization

Authentication and Authorization are key requirements that need to be properly profiled by the ISA^2 IPS REST API profile. The profile selects and endorses the proper standards that can be used in a light context while maintaining a high level of security.

In the REST API light context, the following actors are defined, following the definitions defined in standards, including [RFC6749], [OIDC-CORE]:

- **The Resource Owner (RO):** The RO is the entity capable of granting access to a protected resource. When the RO is a person it is referred to as an **end user**.
- **The Resource Server (RS):** The server hosting the (potentially) protected resources "owned" by the RO. The RS is the server that implements the REST API following this profile.
- **The Client:** This is the client application. The Client accesses the Resources provided by the RS owned by the RO.
- **The Authorization Server (AS):** The server that issues tokens that can be used for a Client's authorisation by an RO to access resources on an RS.
- **The Identity Provider (IdP):** The server that issues claims that can be used to authenticate an end user ([OIDC-CORE]) and/or a Client (extending [OIDC-CORE]).

The form of credentials that are issued by an AS is called an **Access Token (AT)**. The AT is usually a string representing an authorization issued to the Client by an AS and is consumed by the RS.

The Client needs to get authorization and/or authentication claims to access the RO resources, served in the form of REST APIs by one or more RS. When it operates in a light context, the Client does not have its own identity (in an OAuth 2.0 sense, it is a public Client) and so direct authentication and authorization mechanisms for the RO that would be trusted by the various RS are not usually applicable. Also, the provision of direct authentication services from the RS is not generally recommended since the Client can have access to the RO's password. Additionally, direct authentication (i.e., the Client using the RO's credentials) provides full access to the RO's resources without any granularity of the allowed access.

To overcome these deficiencies and provide both Authorization and Authentication, authentication and authorization delegation protocols are being endorsed as mandatory.

For Authorization, the IETF specifies **OAuth 2.0** [RFC6749], a protocol for delegated authorization. OAuth 2.0 specifies mechanisms where a Client can ask a RO for access to RS resources on behalf of the RO and receive an AT as proof that the RO agreed using an AS. However, [RFC6749] leaves many aspects unspecified, like the AT and the scope definition format. The current profile further specifies the token format and the supported authorization flows.

For Authentication, OpenAPI specification endorses OpenID Connect [OIDC-CORE], a profile of OAuth v2.0 used for delegated authentication. OpenID Connect further specifies the AT format, which is based on JWT specification.

This profile mandates using the OAuth v2.0 profile for authorization and recommends using OpenID Connect profile for authentication of an end user and Assertion Framework for OAuth 2.0 Client Authentication [RFC7521] for authentication of a Client.

5.1.1 Architecture Topologies

In a multi-service architecture, the service deployment topologies can vary, depending on the application domain, security and trust aspects, etc. Each such topology poses different challenges on interoperability and security considerations. This section defines the Authorization and IdP service topologies allowed by this profile and the different rules applicable to each of them.

5.1.1.1 Authorization Server Deployment Topology

5.1.1.1.1 Auth-Int Topology – Resource Server-provided Authorization Server

In this deployment topology, the AS is considered a part of the RS domain and it is trusted by the RS. The communication between the RS and the AS is considered as secure and internal, within a secure domain. However, the trust of the AS by the Client and the RO could vary.

5.1.1.1.2 Auth-Ext Topology – External Authorization Server

In this deployment topology, the AS is an external trusted third-party service. The communication between the RS and the AS is not considered secure and internal by default and extra security measures need to be taken into account. More specifically, when the AS is an externally deployed service and the AT format sent by the Client is opaque, e.g. is an authorization code, then:

- The Token Introspection [RFC7662] MUST be supported by the AS and used by the RS *and*
- It is RECOMMENDED that the JWT Response for OAuth Token Introspection [DRAFT-IETF-OAUTH-JWT-INTROSPECTION] be used as the format for Token Introspection endpoint response.

Conversely, when the AT format sent by the Client is not opaque, then:

- The Token Introspection [RFC7662] MAY be supported by the AS and used by the RS *and*
- It is RECOMMENDED that the AT follows the format defined in [RFC9068]

5.1.1.2 Identity Provider Deployment Topology

5.1.1.2.1 IdP-Int Topology – Authorization Server-provided Identity Provider

In this deployment topology, the IdP that provides the authentication service of the RO or the Client is provided by the AS. The communication between the IdP and the AS is considered as secure and internal, within a secure domain. However, the trust of the IdP by the Client and the RO could vary. As the AS and the IdP are considered part of a single service, is it RECOMMENDED to use OpenID Connect 1.0 [OIDC-CORE] as the authentication process, which can be combined with the authorization flows required by the RS (see the section on Authentication Framework Profile).

5.1.1.2.2 IdP-Ext Topology – External Identity Provider

In this deployment topology, the IdP that provides the authentication service of the RO or the Client is an external trusted third-party service. The authentication of the RO MUST be done using the predefined secure protocols as they are defined in the Assertion Framework for OAuth2.0 Client Authentication [RFC7521] and it is RECOMMENDED to be used for the Client Authentication. Furthermore, the AS MUST support either the Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants [RFC7522] or the JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants [RFC7523]. The use of IdP's that provide OpenID Connect 1.0 [OIDC-CORE] as described also in [RFC7523], is RECOMMENDED (see the section on Authentication Framework Profile).

5.1.1.3 Topological Combinations

In the following table, the possible combinations from the supported topologies are provided:

Topological Combinations	IdP-Int	IdP-Ext
Auth-Int	See Figure 2: Internal Authorization Server with Internal Identity Provider	See Figure 3: Internal Authorization Server with External Identity Provider
Auth-Ext	See Figure 4: External Authorization Server with Internal Identity Provider	See Figure 5: External Authorization Server with External Identity Provider

All diagrams below illustrate the scenario where the RO is an end user.

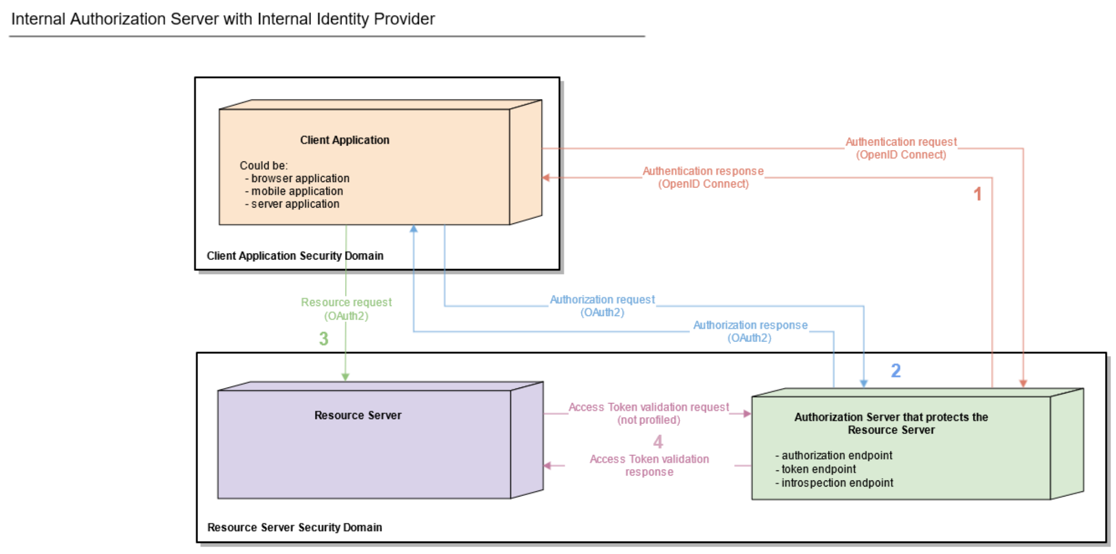
Internal Authorization Server with Internal Identity Provider
Client Application
Could be:
- browser application
- mobile application
- server application
Client Application Security Domain
Authentication request (OpenID Connect)
Authentication response (OpenID Connect)
1
Resource request (OAuth2)
3
Authorization request (OAuth2)
Authorization response (OAuth2)
2
Resource Server
Access Token validation request (not profiled)
4
Access Token validation response
Authorization Server that protects the Resource Server
- authorization endpoint
- token endpoint
- introspection endpoint
Resource Server Security Domain

Figure 2 Internal Authorization Server with Internal Identity Provider

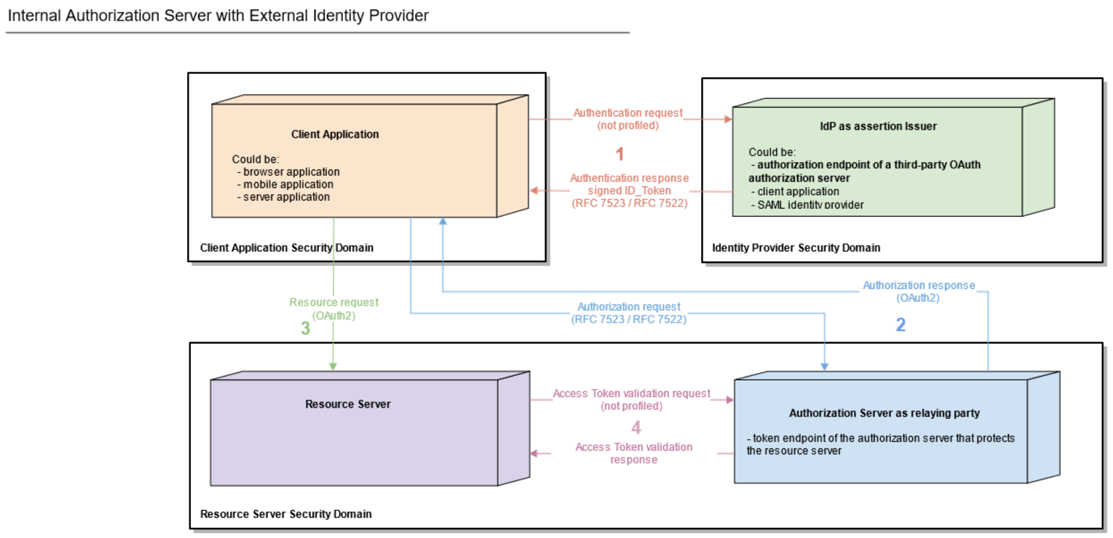
Internal Authorization Server with External Identity Provider
Client Application
Could be:
- browser application
- mobile application
- server application
Client Application Security Domain
Authentication request (not profiled)
1
Authentication response signed ID_Token (RFC 7523 / RFC 7522)
IdP as assertion Issuer
Could be:
- authorization endpoint of a third-party OAuth authorization server
- client application
- SAML identity provider
Identity Provider Security Domain
Authorization response (OAuth2)
2
Authorization request (RFC 7523 / RFC 7522)
Resource request (OAuth2)
3
Resource Server
Access Token validation request (not profiled)
4
Access Token validation response
Authorization Server as relaying party
- token endpoint of the authorization server that protects the resource server
Resource Server Security Domain

Figure 3 Internal Authorization Server with External Identity Provider

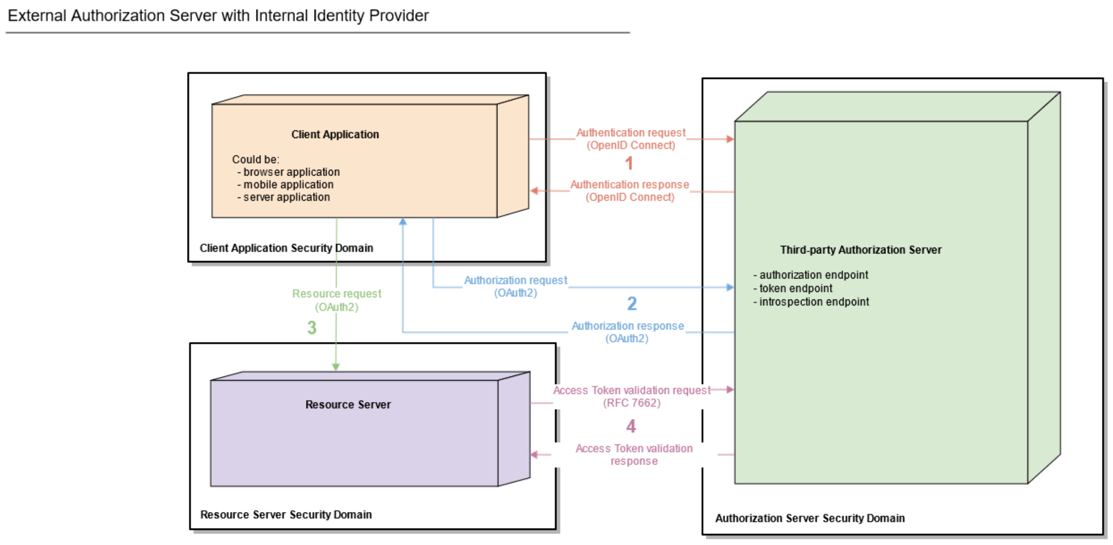
External Authorization Server with Internal Identity Provider
Client Application
Could be:
- browser application
- mobile application
- server application
Client Application Security Domain
Authentication request (OpenID Connect)
1
Authentication response (OpenID Connect)
Third-party Authorization Server
- authorization endpoint
- token endpoint
- introspection endpoint
Authorization request (OAuth2)
2
Authorization response (OAuth2)
Resource request (OAuth2)
3
Resource Server
Access Token validation request (RFC 7662)
4
Access Token validation response
Resource Server Security Domain
Authorization Server Security Domain

Figure 4 External Authorization Server with Internal Identity Provider

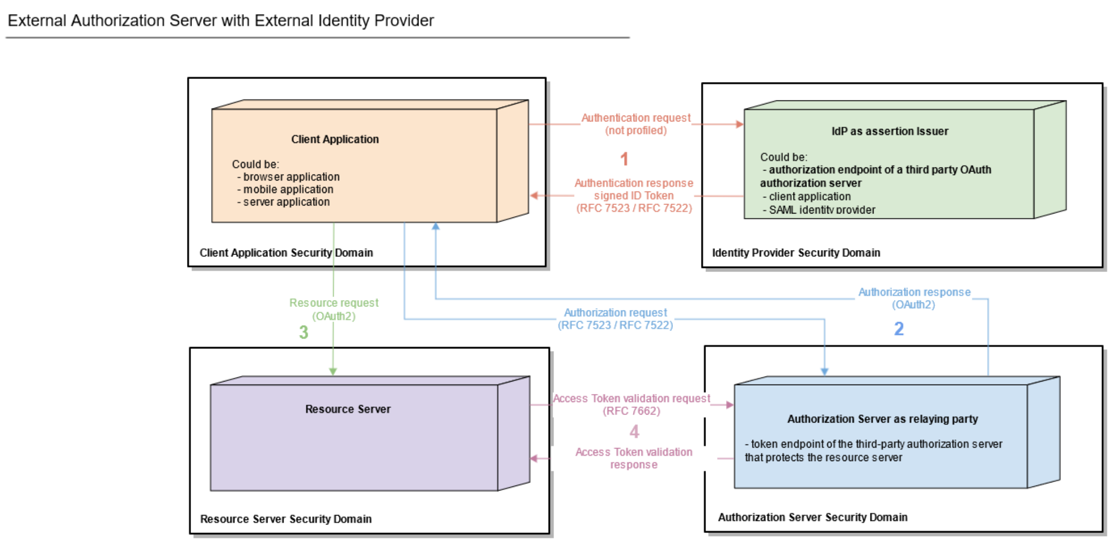
External Authorization Server with External Identity Provider
Client Application
Could be:
- browser application
- mobile application
- server application
Client Application Security Domain
Authentication request (not profiled)
1
Authentication response signed ID Token (RFC 7523 / RFC 7522)
IdP as assertion Issuer
Could be:
- authorization endpoint of a third party OAuth authorization server
- client application
- SAML identity provider
Identity Provider Security Domain
Resource request (OAuth2)
3
Authorization request (RFC 7523 / RFC 7522)
Authorization response (OAuth2)
2
Resource Server
Resource Server Security Domain
Access Token validation request (RFC 7662)
4
Access Token validation response
Authorization Server as relaying party
- token endpoint of the third-party authorization server that protects the resource server
Authorization Server Security Domain

Figure 5 External Authorization Server with External Identity Provider

5.1.2 OAuth 2.0 Authorization Framework Profile

OAuth 2.0 is the IETF Standard for Authorization [RFC6749]. It defines how the RO provides his consent to the Client to access his resources at the RS, by issuing an AT through an AS which is trusted by both the RS and the Client. This is achieved by the execution of predefined flows, also known as grants, between the client application, the RO (User) and the RS (the REST API implementation). OAuth 2.0, in its original specification, defines multiple flows that can be applied on multiple deployment scenarios, like a web browser application, a mobile native application, a service-to-service application, etc. However, many of these flows have been deprecated, found insecure or updated in more recent RFCs. The profile states which flows MUST be supported and which MUST NOT be supported by conformant REST API implementations.

Furthermore, the OAuth 2.0 specification does not include the definition of the token structure and integrity mechanisms. This profile further defines and restricts the type of tokens that MUST be supported by conformant REST API implementations in order to enhance confidentiality, integrity, overall security and interoperability.

5.1.2.1 Profiled OAuth Grants

The following OAuth Grants have been found secure and can be used. At least one of them MUST be used.

5.1.2.1.1 Authorization Code with PKCE Grant

Authorization Code with PKCE Grant is a flow defined in [RFC7636] that is meant to replace the simple Authorization Code Grant and also makes obsolete the Implicit Grant. The Authorization Code with PKCE Grant MUST be supported by the REST API implementations when the RO is an end user.

5.1.2.1.2 Client Credentials Grant

The Client Credentials Grant is a grant mostly meant to be used by service-to-service authorization and SHOULD be used only when the Client is considered a safe, confidential system. The flow MUST NOT be used when the Client is a browser or a native mobile app.

5.1.2.1.3 Assertion Bearer Grant

Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [RFC7521] is an extension to the OAuth 2.0 assertion framework and provides a way to use assertions for granting authorization. [RFC7521] defines the framework for using assertions and [RFC7522] and [RFC7523] formally implement it. [RFC7522] defines how SAML 2.0 Assertions can work as tokens for granting authorization and [RFC7523] defines how OpenID Connect tokens can be used as tokens for authorization grants. [RFC7521] and at least one of its implementations ([RFC7522] or [RFC7523]) MUST be supported by the AS when the IdP-Ext topology is used by the RS implementer.

5.1.2.2 Forbidden OAuth Grants

The following OAuth Grants have been found insecure and MUST NOT be used:

5.1.2.2.1 Authorization Code Grant

Authorization Code Grant has been found insecure and MUST NOT be used. Use Authorization Code with PKCE Grant instead.

5.1.2.2.2 Implicit Grant

The Implicit Grant has been found insecure and MUST NOT be used. Use Authorization Code with PKCE Grant instead.

5.1.2.2.3 Resource Owner Password Credentials Grant

The Resource Owner Password Credentials Grant is an insecure flow, defined as a legacy system support mechanism for granting authorization and it MUST NOT be used.

5.1.2.3 Access Token Format

OAuth 2.0 does not define in its core specification the structure of the AT. This means that the token can follow any kind of structure. AT's, issued by the token endpoint of the AS to be used by the Client, can be of two different types:

- opaque tokens, which are strings that do not carry explicit authorization information and for which the receiving RS must use a validation mechanism, usually provided by the AS;
- concrete tokens, which carry all the authorization information required and can be validated directly by the RS without the need to check with the AS.

When the AT is opaque, e.g. an UUID, the RS cannot validate it in-place and must use verification functionality from the AS. When using opaque tokens and when the AS and the RS are not co-located, are not run by the same entity, or are otherwise separated by some boundary (**Auth-Ext Topology**), the AS MUST implement the OAuth 2.0 Token Introspection [RFC7662] interface and the response token of the Token Introspection interface SHOULD conform to the JWT Response for OAuth Token Introspection [DRAFT-IETF-OAUTH-JWT-INTROSPECTION].

When the AT is not opaque, the tokens SHOULD conform to the JSON Web Token (JWT) Profile for OAuth 2.0 Acess Tokens [RFC9068]. Unsigned custom tokens MUST NOT be used.

5.1.3 Authentication Framework Profile

5.1.3.1 User Authentication Framework Profile

The OAuth 2.0 framework has been defined for implementing authorization grants between the RO, the Client and the RS, using an AS as a trusted third party. However, OAuth 2.0 does not explicitly specify how authentication of an end user can be done, leaving the door open to many different and incompatible implementations.

OpenID Connect v1.0 [OIDC-CORE] fills this gap, by creating an Identification Layer on top of the OAuth 2.0 framework, by introducing a new additional kind of token, called the ID Token that coexists with the typical AT's required by OAuth 2.0.

When end-user authentication is required, then the REST API implementation SHOULD use OpenID Connect for the RO authentication. Open ID Connect uses the same grants as OAuth 2.0 and thus the same restrictions apply to those grants as described in the section on Profiled OAuth Grants, with the exception documented in the section on Profiled OpenID Connect Flows. Furthermore, in order to promote interoperability and compatibility with the eIDAS eID regulatory framework, it is RECOMMENDED that the OpenID Connect for Identity Assurance 1.0 profile [OIDC-DRAFT-ID-ASSUR] of the ID token SHOULD be supported by the IdP and both the RS and the Client. OpenID Connect MUST be used when an external deployment topology is used for authentication (IdP-Ext topology).

In addition to end-user authentication, the authentication of the Client is also RECOMMENDED when possible.

5.1.3.2 Client Authentication Framework Profile

When Clients interact with AS's, they need to authenticate to the AS for the issuing of the AT. Several mechanisms are defined in the OAuth 2.0 specification for client authentication. This profile RECOMMENDS using the Assertion Framework for OAuth 2.0 Client Authentication [RFC7521]. It supports both the Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants [RFC7522] and the JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants [RFC7523] when external deployment topologies are used for both authentication and authorization (Auth-Ext and IdP-Ext topology).

5.1.3.3 Profiled OpenID Connect Flows

As OpenID Connect is based on OAuth 2.0, the same rules and profiles of grants apply also to OpenID Connect with one exception: for OpenID Connect to work properly on applications that are browser-based, for example, the Hybrid Flow MAY be supported. Since the Hybrid Flow requires the Implicit Grant to be provided, this profile allows the use of the Implicit Grant only as part of the OpenID Connect Hybrid Flow, and only when requesting short-lived tokens like the Identity Token.

5.2 SECURITY

5.2.1 Transport Level Security

This profile mandates the use of Transport Layer Security. TLS provides communication integrity, confidentiality and authentication. Server authentication, using a server certificate, allows the client to make sure the HTTPS connection is set up with the right server. REST API implementations conformant to this profile MUST therefore use TLS 1.2 [RFC5246] or TLS 1.3 [RFC8446]. TLS 1.3 is RECOMMENDED.

In line with guidance from ENISA, the following algorithms MUST be used when using TLS:

Key exchange	Certificate Verification	Bulk Encryption	Hashing
ECDHE	ECDSA RSA	AES_256_GCM CHACHA20_POLY1305 AES_128_GCM	(HMAC-)SHA-384 (HMAC-)SHA-256

All other algorithms MUST NOT be used for Key exchange, Certificate Verification, Bulk Encryption and Hashing. Furthermore, TLS Compression MUST NOT be used.

Other less secure protocols, such as SSL 2.0, SSL 3.0, TLS 1.0 and TLS 1.1 MUST NOT be used.

5.2.2 Message And Payload Level Security

This profile defines the security mechanisms that MUST apply for both payload and message-level security when such a security mechanism is considered by the REST API. For both payload-level and message-level signatures, the profile enforces the use of JWS detached signatures following the HttpHeaders Mechanism of the ETSI ESI JAdES specification [ETSI-JADES]. This structure is enforced for the following reasons:

- JWS, being a simple JSON Structure, can be supported by clients in a light context, while specifications like the ETSI ESI ASIC containers are more difficult to do.
- JWS in detached form does not change the payload structure, meaning that a client not supporting the validation of signature can continue to operate as if there was no signature applied.
- JWS Detached can be transported using an HTTP header, making its presence unintrusive and easily transportable.

Following ENISA's Good Practises in Cryptography – Primitives and Schemes [ENISA-CRYPTO-2020], the following algorithms found in [RFC7518] are selected for this profile, to be used in the following form:

- The ECDSA Algorithm with SHA-256 and P-256 Curve MUST be supported, with a key length of at least 256 bits. The value `"ES256"` for the `alg` parameter MUST be used in this case as defined in [RFC7518].
- The EdDSA Algorithm [RFC8032] using one of the curves defined in [RFC7748] SHOULD be supported and is RECOMMENDED for use, with a key length of at least 256 bits. The value `"EdDSA"` for the `alg` parameter MUST be used in this case and the curve shall be encoded in the `crv` parameter as defined in [RFC8037].

5.2.2.1 Payload Security

Payload security ensures the integrity and authenticity of the payload part of the message. The typical mechanisms to achieve this are digital signatures. When payload security is considered, the Detached JSON Web Signatures following the JAdES specification [ETSI-JADES] MUST be applied with the following restrictions:

- The JWS content (Data to be Signed) MUST be detached from the signatures as defined in [RFC7515] Appendix F.
- The signed `SigD` parameter object MUST be present in the JWS headers, denoting the use of the JAdES detached header profile.
- The value of the `mId` parameter MUST be set to `"http://uri.etsi.org/19182/HttpHeaders"`.

- The `pars` array of the `SigD` MUST contain only the element "digest", denoting that for the calculation of the signature only the digest of the HTTP payload must be taken into account, according to [RFC3230].
- The `alg` parameter MUST be set to the correct value depending on the algorithm used (see above).
- If the `alg` parameter is set to `"EdDSA"`, the `crv` parameter MUST be set to the correct value (see above).

The JWS structure shall be carried in HTTP header field named `edel-payload-sig`. The header field can be used in both requests and responses. The header field MUST not appear more than once in a message; if a message contains multiple `edel-payload-sig` header fields, the receiver MUST consider the signature invalid.

5.2.2.2 *Message-Level Security*

The Introduction section of [DRAFT-IETF-HTTPSBIS-MSG-SIGS] details why message integrity and authenticity are critical to the secure operation of many HTTP/REST applications.

When Message-Level Security is considered, the HttpHeaders Mechanism of the JAdES Specification [ETSI-JADES] MUST be used, with the following restrictions applied:

- The JWS content (Data to be Signed) MUST be detached from the signatures as defined in [RFC7515] Appendix F.
- The signed `SigD` parameter object MUST be present in the JWS headers, denoting the use of the JAdES detached header profile.
- The value of the `mId` parameter MUST be set to `"http://uri.etsi.org/19182/HttpHeaders"`.
- The `pars` array of the `SigD` MUST contain **at least** the following elements:
    - the element "(request-target)", for containing the HTTP Request URI
    - the element "host", for containing the host the message was submitted to, if present
    - the element "origin", for containing the scheme, hostname, and port from which the request was initiated, if present
    - the element "content-encoding", if present
    - the element "content-type", if present
    - the element "content-length", if present
    - the element "digest", for taking into account the Digest header that contains the hash value of the HTTP payload.
- The `alg` parameter MUST be set to the correct value depending on the algorithm used (see above).
- If the `alg` parameter is set to `"EdDSA"`, the `crv` parameter MUST be set to the correct value (see above).

Implementations that make use of the HTTP Header fields for data representation SHOULD also include these header fields in the `pars` array.

The JWS structure MUST be carried in HTTP header field named `edel-message-sig`. The header field can be used in both requests and responses. The header field MUST not appear more than once in a message; if a message contains multiple `edel-message-sig` header fields, the receiver MUST consider the signature invalid.

5.2.2.3 Signature Representations

Both Payload and Message-Level Security are realised using the JAdES profile [ETSI-JADES] of the JSON Web Signature. The following snippet shows the OpenAPI declaration and usage of the JWS detached signature used in an HTTP Response:

```
openapi: 3.1.0
info:
  title: JAdES Signatures
  summary: An example showcasing JAdES signatures
  description: An example showcasing JAdES signatures as JWS detached
               signatures for securing a sample REST endpoint
               (/certificate)
  termsOfService: https://domain.server.io/terms-of-service
  license:
    name: EUPL-1.2 or later
    url: https://eupl.eu/1.2/en/
  version: 1.0.0
  x-edelivery:
    lifecycle:
      maturity: supported
    publisher:
      name: ACME Publisher
      URL: https://www.acme-publisher.org/
externalDocs:
  description: The ISA² IPS REST API Core Profile
  url: https://joinup.ec.europa.eu/collection/api4dt/document/isa2-ips-rest-
api-profile
servers:
- url: https://domain.server.io/v2
tags:
- name: DetachedPayloadSignature
  description: Operations using payload security
- name: DetachedMessageSignature
  description: Operations using message-level security
paths:
  /openapi:
    get:
      summary: Returns the OpenAPI Document for the API
       ...
      responses:
        200:
          description: ...
          content: {
            $ref: 'https://spec.openapis.org/oas/3.1/schema/2021-05-20'
             ...
          }

  /certificate:
    get:
      tags:
      - DetachedMessageSignature
      summary: Get a Certificate
      securitySchemes:
        OAuth2:
          type: oauth2
          flows:
            authorizationCode:
              authorizationUrl: https://example.com/api/oauth/dialog
              scopes:
                send:message: send a message
       ...
      responses:
        200:
          headers:
            edel-message-sig:
              $ref: '#/components/headers/edel-message-sig'
          description: List of Certificates
          content: { ... }

components:
  headers:
    edel-payload-sig:
      schema:
        $ref: '#/components/schemas/JwsCompactDetached'

    edel-message-sig:
      schema:
        $ref: '#/components/schemas/JwsCompactDetached'

  schemas:
    JwsCompactDetached:
      title: The format for the message-level and payload signature
      description: Defines the string pattern as a regular expression that
                   MUST be followed to represent detached JWS compact tokens
      "$id": https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/api-core-profile/components/schemas/jws-compact-detached.json
      "$schema": https://json-schema.org/draft/2020-12/schema
```

```
        type: string
        format: jws-compact-detached
        pattern: ^[A-Za-z0-9_-]+(?:(\\.\\.)[A-Za-z0-9_-]+){1}
```

5.3 LIFECYCLE MANAGEMENT

REST APIs are typically published to the web and then consumed by clients. This builds a dependency between the clients that are considered the consumers of the API and the producer which is the API implementation service. Properly governed APIs should have a streamlined process of evolution and change introduction. This profile mandates certain aspects of the lifecycle of the APIs, like versioning and deprecation.

5.3.1 API Versioning Semantics

Conformant APIs MUST use semantic versioning [SEMVER] for their lifecycle management. In semantic versioning, the version number is split into three different sections: the major version, the minor version and the patch version. The following sections describe how semantic versioning MUST apply.

Any changes of the documentation of a REST API MUST be considered as changes of the API itself and treated accordingly from a versioning standpoint.

It should be noted that the prescriptions of this section apply only when the API specification and the deployed version are synchronised. In situations where an API specification evolves independently from the deployed version, e.g. it is provided as a deployable/implementable standardised specification with a separate lifecycle, the version deployed at the server MUST be considered and not the version of the API specification.

For example, a client implementing a more recent version of a standardised API, which the server does not support yet, is **not compatible** with the server, as it might use functionality found in the newer version of the API specification that is not yet implemented by the server.

5.3.1.1 Backward-compatible changes

The API designer should strive for API evolution and backward-compatible changes rather than backward-incompatible changes with existing client code.

Backward-compatible changes that do not affect any of the clients include the addition of new operations and schemas. Also, depending on the client's expectations, the addition of optional fields into pre-existing schemas MAY be considered a backward-compatible change.

5.3.1.2 Backward-incompatible changes

The API designer might require to introduce a backward-incompatible change with existing client code. The following changes are considered backward-incompatible:

- Removing, renaming, or moving API entities such as:
    - operation endpoints

  - HTTP methods associated with endpoints
  - operation query, body, or header parameters
  - schema properties
  - security schemes
- Changing the way how existing features need to be used, e.g. by introducing new preconditions to be fulfilled
- Changing an already present workflow
- Making optional parameters or schema properties mandatory
- Changing documented functional or non-functional behaviour in significant ways

Furthermore, the REST API designer may consider any other changes that are not strictly breaking as backward-incompatible, and thus warranting a major version upgrade, if there is a potential impact of the current client base that requires a proper migration period.

Also, depending on the client's expectations, the addition of optional fields into pre-existing schemas SHOULD be considered a backward-incompatible change.

5.3.2 API Lifecycle

APIs conformant to this profile MUST publish information about their maturity level. To provide lifecycle metadata of the API such as its maturity, deprecation and sunset, the OpenAPI Document MUST contain the `info.x-edelivery.lifecycle` object following the below structure:

Field Name	Type	Description	Optionality
maturity	string	The maturity level of the API. It MUST contain one of the following values: • development • supported • deprecated	Mandatory
deprecatedAt	string	The date when the API has been deprecated. The date format MUST follow [RFC3339]	Optional
sunsetAt	string	The date when the API will be sunset. The date format MUST follow [RFC3339]	Optional

5.3.2.1 Deprecation

Deprecation of resources and operations is considered a mechanism for a smooth transition between major versions. Resources and operations affected by backward-incompatible changes MUST be marked as deprecated before being removed or otherwise changed in a new major version of the API. Deprecation of the API could be applied at the operation level (resource + HTTP verb) or for the API as a whole.

When deprecating individual operations, the use of the `deprecated` attribute for the concerned operation(s) MUST be set to `true` in the OpenAPI Document of the API. Furthermore, the **Deprecation HTTP Response Header** for the specific operation MUST be set to **true**, according to the Deprecation Response Header Internet-Draft [DRAFT-IETF-HTTPAPI-DPRC-HDR].

When deprecating the API as a whole, the OpenAPI Document of the API must contain in the `info.x-edelivery.lifecycle` property the following declared attributes:

- The `info.x-edelivery.lifecycle.maturity` attribute MUST be set to `deprecated`.
- The `info.x-edelivery.lifecycle.deprecatedAt` attribute MUST be set to the date the API was deprecated.

Example:

```
OpenAPI:
  ...
  info:
    ...
    x-edelivery:
      lifecycle:
        maturity: deprecated
        deprecatedAt: 2020-12-31
    ...
```

Furthermore, when the API is deprecated as a whole, the Deprecation HTTP Response Header for every operation MUST also be set to true, according to the Deprecation Response Header Internet-Draft [DRAFT-IETF-HTTPAPI-DPRC-HDR].

5.3.2.2 Sunset

Sunsetting of operations is done with the use of semantic versioning. New major versions of the API MUST have all the deprecated operations of the previous major versions removed. It is recommended practice to announce deprecation sufficiently in advance to allow clients to upgrade.

For sunsetting the API, the OpenAPI Document of the API must contain in the `info.x-edelivery.lifecycle` property the following declared attributes:

- The `info.x-edelivery.lifecycle.maturity` attribute MUST be set to `deprecated`.
- The `info.x-edelivery.lifecycle.deprecatedAt` attribute MUST be set to the date the API was deprecated.
- The `info.x-edelivery.lifecycle.sunsetAt` attribute MUST be set to the date the API will be withdrawn and not accessible any more.

Example:

```
OpenAPI:
  ...
  info:
```

```
...
x-edelivery:
  lifecycle:
    maturity: deprecated
    deprecatedAt: 2020-12-31
    sunsetAt: 2021-02-28
...
```

Furthermore, when the API is marked as sunset, the Deprecation HTTP Response Header for every operation MUST also be set to true, according to the Deprecation Response Header Internet-Draft [DRAFT-IETF-HTTPAPI-DPRC-HDR] and the Sunset HTTP Response Header [RFC8594] for every operation must be set to the date the API is to be sunset.

5.3.3 API Deployment Considerations

Following the current best practices, the API MUST declare its current major version number in the URL, using URL versioning. Only the major number of the API version MUST be used in the URL. The following pattern MUST be applied to the API URL:

```
https://{domain}/{baseURL}/v{version.major}/
```

The following table explains the API syntax pattern:

URL Part	Description
domain	the domain name the URL is deployed at (server)
baseURL	the base URL of the deployed API
version.major	The major version number of the API, from its semantic API versioning

5.4 Common Semantics

5.4.1 REST API Design

According to section 5.2 in [FIELDING-2000],

> *[t]he key abstraction of information in the REST architecture is the notion of resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. [...] A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.*

Resources are addressed using URIs. The URI path conveys a REST API's resource model, with each forward slash-separated path segment corresponding to a unique resource within the model's hierarchy. Implementers SHOULD follow this profile, which defines the typical structure of a resource URI and the naming conventions for the specific resource archetypes: the **document**, the **collection**, the **store** and the **controller**.

5.4.1.1 URI Structure

According to [RFC3986], the generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment as shown in the example below.

Scheme		Authority	Resource Path	Query	Fragment
`https`	`://`	`example.com:8042`	`/messaging/messages`	`?from=44556`	`#subject`

This profile enforces the following **resource path** naming conventions and rules:

- Forward slash separator (/) MUST be used to indicate a hierarchical relationship. The forward slash (/) character is used in the resource path to indicate a hierarchical relationship between resources.
- A trailing forward slash (/) MUST NOT be included in URIs. A REST API MUST generate and communicate clean URIs and SHOULD be intolerant of any client's attempts to identify a resource imprecisely. REST APIs conformant to this profile SHOULD NOT expect a trailing slash and SHOULD NOT include them in the links that they provide to clients.
- Resource path segments MUST be separated using a hyphen (-). In particular, in the case of resource identifiers, the hyphen (-) MUST be used instead of a space, but other special characters are allowed.
- Query parameter names MUST be separated using underscore (_).
- Resource paths and query strings MUST URL encode data into UTF-8 octets.
- Sub-resource collections MAY exist directly beneath an individual resource. This should convey a relationship to another collection of resources.
    - `/messages/{message-id}/responses`
- Resource identifiers SHOULD follow the recommendations described in the subsequent section.
- Resources MUST conform to one of the defined resource archetypes (see section on Resource Archetypes), following their name convention rules.
- File extensions SHOULD NOT be used in Resource Paths in order to select resource representations. The best practice is to use headers in the HTTP request to define the file types that are used/expected.

The following table provides a formal grammar on the resource path and query structure that MUST be followed:

Term	Description	Formal Definition	Example
resource-path	The hierarchical URI sub structure defining an addressable resource	'/' base-url '/' version ('/' namespace)* ('/' resource)+	/archive-system/v2/messaging-service/messages/123-121
base-url	The base URL of the deployed API	ALPHA (ALPHA | DIGIT | '-')*	archive-system
version	The version of the deployed API	'v' (DIGIT)+	v2

Term	Description	Formal Definition	Example
namespace	Namespace identifiers are used to provide a context and scope for resources. They are determined by logical boundaries implemented by the API platform.	ALPHA (ALPHA | DIGIT | '-')*	messaging-service
resource	The hierarchical structure of a (sub)resource	resource-name ['/' resource-id]	messages/123-121
resource-name	The name of the resource	ALPHA (ALPHA | DIGIT | '-')*	messages
resource-id	The identifier of a resource	value	123-121
query	The filter string	name '=' value ('&' name = value)*	subject=report
name	The filter name	ALPHA (ALPHA | DIGIT | '_')*	subject
value	A value that is potentially URL encoded (used as a resource identifier or as a filter value)	URL-encoded value	report%202020

Note: ALPHA and DIGIT are used as defined in section 6.1 of [RFC2234].

5.4.1.2 Resource Identifiers

Resource identifiers identify a resource or a sub-resource. They MUST conform to the following guidelines:

- The lifecycle of a resource identifier MUST be defined as part of resource's domain model, where it can be guaranteed to uniquely identify a single resource.
- A UUID or a hash value (e.g. HMAC-based) SHOULD be used as a resource identifier.
- For security and data integrity reasons, all sub-resource IDs MUST be scoped within the parent resource only. Example:
    - `/users/21725f10-b819-4dea-ad76-0d169652274b/linked-accounts/5100a97c-dfd8-480d-ba5a-a24d0c278e92`

Even if the account "5100a97c-dfd8-480d-ba5a-a24d0c278e92" exists, it MUST NOT be returned unless it is linked to the user "21725f10-b819-4dea-ad76-0d169652274b".

- Resource identifiers SHOULD try to use ASCII characters. There SHOULD NOT be any ID using UTF-8 characters.
- Resource identifiers values MUST perform URL encoding for any character other than URI unreserved characters (as defined in section 2.3 of [RFC3986]).

5.4.1.3 Resource Archetypes

5.4.1.3.1 Document

A **document** resource is a singular concept that is akin to an object instance. Document resources are usually inside collection resources. A document's state representation typically includes both fields with values and links to other related resources. Singular nouns or resource identifiers MUST be used to denote document resource archetypes.

```
http://example.com/api/v1/messaging/messages/{message-id}
http://example.com/api/v1/user-management/users/admin
```

5.4.1.3.2 Collection

A **collection** resource is a server-managed list of resources. Clients may propose new resources to be added to a collection. However, it is up to the collection to choose to create a new resource or not. A collection resource chooses what it wants to contain and also decides the URIs of each contained resource. Plural nouns MUST be used to denote collection resource archetypes.

```
http://example.com/api/v1/messaging/messages
http://example.com/api/v1/user-management/users/{user-id}/accounts
```

5.4.1.3.3 Store

A **store** is a client-managed resource repository. A store resource lets an API client put resources in, get them back out, and decide when to delete them. A store never generates new URIs. Instead, each stored resource has a URI. The URI was chosen by a client when it was initially put into the store. Like collections, plural nouns MUST be used to denote store resource archetypes.

```
http://example.com/api/v1/song-management/users/{user-id}/playlists
```

5.4.1.3.4 Controller

A **controller** resource models a procedural concept. Controller resources are like executable functions, with parameters and return values, inputs and outputs. Controllers are the exception to the rule, using **verbs** to denote controller archetypes and they MUST appear as the last segment in a resource URI.

```
http://example.com/api/v1/cart-management/users/{user-id}/cart/checkout
http://example.com/api/v1/song-management/users/{user-id}/playlist/play
```

5.4.1.4 Query Parameters

Following the HTTP URI structure and the REST API best practices, the query parameters must be placed in the query component after the '?' character that denotes the start of the query component. The following rules apply to the query parameters:

- Query parameter names MUST be separated using underscore (_).
- Query parameter names MUST start with a letter and SHOULD be in all lowercase. Only alphanumeric characters and the underscore (_) character SHALL be used.
- Query parameter values MUST be URL encoded.
- Query parameters SHOULD be optional.
- Some query parameter names are reserved, as indicated in the section below.

5.4.1.4.1 Pagination and Query Parameters

A common use of query parameters is to provide information on the pagination of results when collection resources are returned. The profile defines the following query parameters that SHOULD be used for result pagination:

- **limit:** The number of resources of a collection to be returned from a request. The limit MUST be a positive integer
- **offset:** The offset the response should start providing resources of the collection from. It MUST be a non-negative integer
- **cursor**: An alternative to using the offset is to provide an identifier of the position as of which the next resources of the collection should be returned. The cursor MUST be an (opaque) string
- **q:** A generic query parameter used to express complex queries on the resource. The structure of the query string is resource specific and MUST be defined per resource
- **sort:** Used to express the sorting order of resources in a collection. It MUST follow the following regular expression: `(-|+)<field-name> (',' (-|+)<field-name>)*` where:
  - "+" denotes ascending order and "-" descending order,
  - <field-name> is a string representation of a field of the resource,
  - the sort order of resources should follow the order of the fields.

As an example, to obtain the third page of one hundred results, sorted by the title of the playlist in ascending order and by the position in the playlist in descending order, the following query string would be provided:

```
https://api.example.org/endpoint?limit=100&offset=200&sort=+playlist-title,-
playlist-position
```

When offset-based pagination is used, the response SHOULD provide pagination hints. It is RECOMMENDED to provide such hints by including a the Link header [RFC5988] in the response with links with relation types `first`, `previous`, `next` and `last`, e.g.:

```
Link: <https://api.example.org/endpoint?offset=0>; rel=first,
      <https://api.example.org/endpoint?offset=100>; rel=previous,
      <https://api.example.org/endpoint?offset=300>; rel=next,
      <https://api.example.org/endpoint?offset=700>; rel=last
```

When cursor-based pagination is used, the response MUST provide the value of the next cursor. It is RECOMMENDED to provide this value by including a Link header [RFC5988] in the response with a link with relation type `next`, e.g.:

```
Link: <https://api.example.org/endpoint?cursor=cGxheWxpc3Q6WDc2QjBRVE45>;
rel="next"
```

5.4.1.4.2 Pre-defined Projection Parameters

The profile recommends the support of two different resource projections, a minimal one and a complete one. The complete one will return the full structure of the resource representation of all the resources of a collection. In contrast, the minimal one returns only the minimal required structure to distinguish a resource inside the collection. This can be a single Identifier or a collection of mandatory fields of the resource.

The profile RECOMMENDS the use of the Prefer HTTP field as defined in [RFC7240] for requesting the minimal or complete representation of resources. Specifically, when the client requires the complete representation to be provided it SHOULD add the Prefer header with the value "return=representation" as shown below.

```
Prefer: return=representation
```

When the client requires the minimal representation to be provided it SHOULD add the Prefer header with the value "return=minimal" as follows:

```
Prefer: return=minimal
```

As the "Prefer" header is optional, it MUST be declared in the operations of the OpenAPI Document when supported by the API.

5.4.2 Common Payload Representations

The profile RECOMMENDS the use of predefined vocabularies as common representations of resources. An API implementer conformant to this profile SHOULD use these representations when possible instead of creating a new schema representation from scratch. The following registries and repositories provide re-usable vocabularies in JSON-LD format.

5.4.2.1 ISA^2 Core Vocabularies

The ISA^2 Core Vocabularies [ISA-CORE-VOC] are simplified, reusable, and extensible data models that capture the fundamental characteristics of an entity, such as a person or a public organisation, in a context-neutral manner. ISA^2 has developed the Core Vocabularies for public administrations in an open process with the active involvement of the SEMIC action stakeholders.

The Core Vocabularies are:

- **Core Person**: captures the fundamental characteristics of a person, e.g. name, gender, date of birth, location.
- **Core Business**: captures the fundamental characteristics of a legal entity (e.g. its identifier, activities) which is created through a formal registration process, typically in a national or regional register.
- **Core Location**: captures the fundamental characteristics of a location, represented as an address, a geographic name or geometry.
- **Core Criterion and Core Evidence**: describes the principles and the means that a private entity must fulfil to become eligible or qualified to perform public services. A Criterion is a rule or a principle that is used to judge, evaluate or test something. An evidence is a means to prove a Criterion.
- **Core Public Organisation**: describes public organisations in the European Union.

The vocabularies are provided in JSON-LD format, making them re-usable in a REST API implementation and can be extended according to the ISA^2 Core Vocabulary rules.

5.4.2.2 Schema.org

Schema.org is a collaborative, community activity with a mission to create, maintain, and promote schemas for structured data on the Internet, on web pages, in email messages, and beyond. Schema.org vocabulary can be used with many different encodings, including RDFa, Microdata and JSON-LD. These vocabularies cover entities, relationships between entities and actions, and can easily be extended through a well-documented extension model. REST API Implementers SHOULD try to reuse representations found in schema.org in JSON-LD format.

5.4.3 Single and Multipart Resource Representations

The profile supports both the use of single-part and of multipart resource representations. Single-part representation is used when the request or response resource representation can be structured in a single body of a single specific content type. In contrast, multipart representation is used when a compound or multiple representations must be returned or provided.

5.4.3.1 Single-part Resource Representations

Single-part resource representation is the most common one for structuring request or response representations. It is used when it is known at design time that the representation is of a single and one specific media type, e.g. application/json. The HTTP representation metadata fields are used for any metadata required to further describe and qualify the representation.

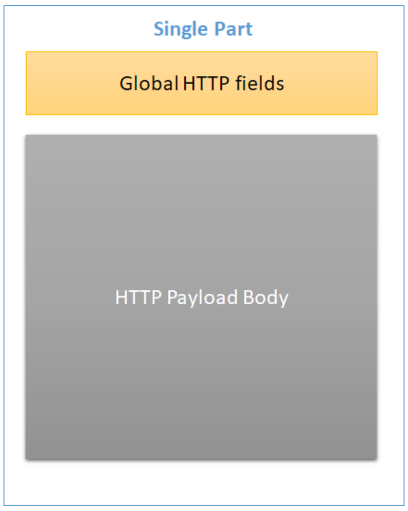

Figure 6 Single-part Resource Representation

5.4.3.2 Multipart Resource Representations

The multipart structure is used when the resource provided cannot be properly represented using a single body of a single specific media type. When the client needs to send multiple representations in one message or the server needs to respond with multiple resource representations to a client's request or provide a compound representation, consisting of multiple interrelated representations of different content types, the multipart structures MUST be used according to the following sections.

5.4.3.2.1 Use of multipart for bundling independent representations

When bundling multiple independent payloads in an HTTP message, either Multipart Mixed or Multipart Parallel MUST be used. When body subparts contained in the message need to be bundled in a particular order then Multipart Mixed MUST be used. Alternatively, when the body subparts do not have to follow a particular order, the Multipart Parallel media type should be used. Any "Multipart" subtypes that an implementation does not recognize must be treated as being of subtype "Mixed".

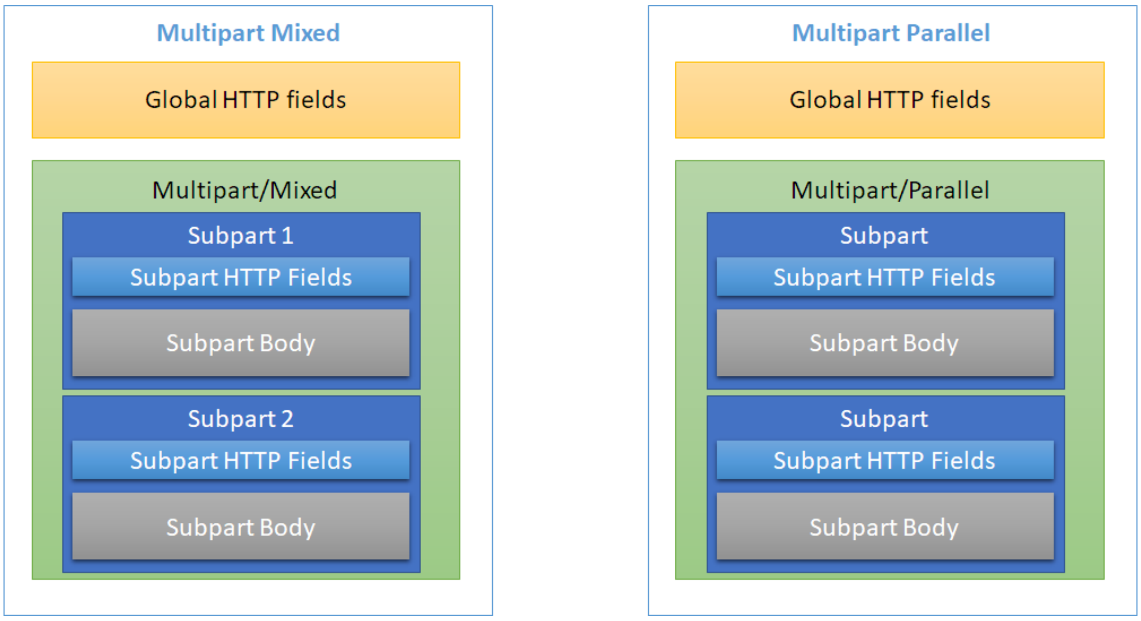

Figure 7 Multipart Mixed or Multipart Parallel Resource Representation

5.4.3.2.2 Use of multipart for bundling inter-related representations

When bundling multiple representations in an HTTP message of compound objects consisting of several inter-related body parts and when proper handling cannot be achieved by individually processing the constituent body parts, but rather only by processing the HTTP Message as an aggregate, Multipart Related MUST be used. The profile enforces the use of Multipart Related as defined in [RFC2387]. In Multipart Related, there must always be a "root" subtype, that only references but is not referenced, and the rest of the subtypes must be referenced either by the root or by the other subparts.

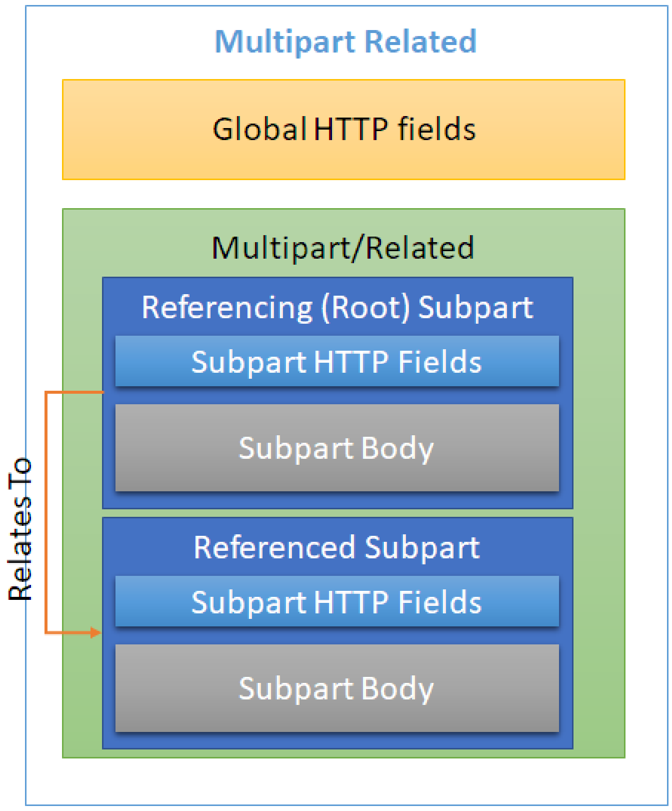

Figure 8 Multipart Related Resource Representation

5.4.3.2.3 Summary

The following table summarises the different multipart variants that MUST be used, depending on the representation requirements:

Multipart subtype	RFC	Use Case
Mixed	[RFC2046] - Section 5.1.3	Provision of multiple resources **with** an explicit order
Parallel	[RFC2046] - Section 5.1.6	Provision of multiple resources **without** an explicit order
Related	[RFC2387]	Provision of a compound set of resource representations to be processed as a single resource

5.4.4 Common Semantics on Methods

This section provides a usage profile of the HTTP Methods as defined in the [RFC7231]. APIs conformant to this profile MUST only use the following subset of HTTP methods:

Note
Custom methods MUST NOT be used when creating an API based on a specification.

5.4.4.1 GET

GET MUST be used to retrieve a representation of a resource**.** It is a strict read-only method, which should never modify the state of the resource. For transfer of a current selected representation or a specific range of the target resource. (See section 4.3.1 of [RFC7231])

5.4.4.2 HEAD

HEAD MUST be used to only retrieve the HTTP response headers. HEAD returns the same response as GET, except that the API returns an empty body. It is strictly read-only. For transfer of HTTP header information of a current selected representation or a specific range of the target resource. (See section 4.3.2 of [RFC7231])

5.4.4.3 POST

POST MUST be used to create a new resource in a collection or to execute an action resource. The POST request's body contains the suggested state representation of the new resource to be added to the server-owned collection. The response should contain a Location HTTP header containing the newly created URI. Successful POST requests will usually generate a 201 (if resources have been created), 202 (if the request was accepted but has not been finished yet), and exceptionally 204 with Location header (if the actual resource is not returned) response. POST operations may have side effects (i.e. modify state) and are not required to be idempotent. (See section 4.3.3 of [RFC7231])

5.4.4.4 PUT

PUT MUST be used to update by replacing a stored resource under a consumer-supplied URI. It MAY be used to insert a new resource in case the client application decides on the resource identified but this use is NOT RECOMMENDED.

If the URI refers to an already existing resource, the enclosed entity SHOULD be considered as a new version to replace the one residing on the server. If the target resource is successfully modified in accordance with the state of the enclosed representation, then a 200 (OK) response SHOULD be sent to indicate successful completion of the request.

If the URI does not point to an existing resource, and that URI is capable of being defined as a new resource, the server can create the resource with that URI. The server MUST inform the client by sending a 201 (Created) response to indicate successful creation. PUT operations may have side effects (i.e. modify state) but MUST be idempotent. (See section 4.3.4 of [RFC7231])

5.4.4.5 PATCH

PATCH MUST be used to update a resource either partially or fully. The enclosed payload contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version. The PATCH method affects the resource identified by the Request-URI, and it also MAY have side effects on other resources; i.e. new resources may be created, usually sub-resources of the target resource, or existing ones modified, by the application of a PATCH. PATCH operations are not required to be idempotent, however, they will often be in practice. (See section 2 of [RFC5789])

5.4.4.6 DELETE

DELETE MUST be used to remove a resource from its parent. Once a DELETE request has been processed for a given resource, the resource can no longer be found by clients. Therefore, any future attempt to retrieve the resource's state representation, using either GET or HEAD, must result in a 404 ("Not Found") status returned by the API. (See section 4.3.5 of [RFC7231])

5.4.4.7 OPTIONS

OPTIONS MUST be used for requesting information about the communication options available for the target resource, at either the origin server or an intervening intermediary. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action. (See section 4.3.7 of [RFC7231])

5.4.5 Common Semantics on Status Codes

HTTP Semantics [RFC7231] defines the "status codes". The full list of available status codes is tracked in the IANA Registry defined for this purpose [HTTP-STATUS-CODES-REG]. They are distinguished as Successful (2xx), Redirection (3xx), Client Error (4xx) and Server Error (5xx). This section provides a usage profile of the status codes allowed by this profile and the methods for which they can be used. No other status codes than those profiled below MUST be returned by the REST API server. To cater for intermediate components that may not be under the control of the server (e.g., API gateways, firewalls), Clients MUST be prepared to handle other status codes as well.

For Client Errors and Server Errors, whenever an explanation of the error situation must be provided, it MUST be represented using Problem+JSON [RFC7807] (see section on Error Messages).

5.4.5.1 Successful (2xx)

Code	Status	Methods
200	OK	GET, HEAD, PUT, PATCH, POST, DELETE, OPTIONS
201	Created	POST, PUT (when PUT creates a new resource)
202	Accepted	POST, PUT, PATCH, DELETE
204	No Content	POST, HEAD, PUT, PATCH, DELETE, OPTIONS

5.4.5.2 Redirection (3xx)

Code	Status	Methods
301	Moved Permanently	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
303	See Other	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
304	Not Modified	GET, HEAD
307	Temporary Redirect	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS

5.4.5.3 Client Error (4xx)

Code	Status	Methods
400	Bad Request	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
401	Unauthorized	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
403	Forbidden	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
404	Not Found	GET, HEAD, PUT, PATCH, DELETE, OPTIONS, POST (if parent resource is not found)

405	Method Not Allowed	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
406	Not Acceptable	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
409	Conflict	POST, PUT, PATCH
412	Precondition Failed	POST, PUT, PATCH, DELETE
413	Payload Too Large	POST, PUT, PATCH
415	Unsupported Media Type	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS

5.4.5.4 Server Error (5xx)

Code	Status	Methods
500	Internal Server Error	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
502	Bad Gateway	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
503	Service Unavailable	GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS

5.4.6 Error Messages

APIs conformant to this profile MUST use the predefined HTTP Error status codes (4xx and 5xx, see section 6 of [RFC7231]). When this is considered not enough by the API designer, the Problem+JSON [RFC7807] MUST be used for representing problem details. The `status`, `title` and `type` attributes MUST always be present. Problem detail syntax can be further profiled according to [RFC7807].

An example of a Problem Details JSON object is presented below:

```
{
  "type": "https://example.com/.../problems/resourceNotFound",
  "title": "Citizen not found",
  "status": 404,
  "detail": "No citizen with ID number 0206731645",
  "instance": "problems/d9e35127-e9b1-4201-a211-2b52e52508df"
}
```

The JSON schema for the Problem Details object is defined as follows:

```
Problem:
    "$schema": https://json-schema.org/draft/2020-12/schema
    "$id": https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/api-core-profile/components/schemas/problem.json
    title: A Problem Details object (RFC 7807) defined by the ISA² IPS
           REST API Core Profile
    description: A Problem Details object (RFC 7807) with ISA² IPS REST
                 API extensions, used for signals (responses) to
                 messages
    type: object
    properties:
      type:
        type: string
        format: uri
        description: An URI reference that identifies the problem type.
                     When dereferenced, it SHOULD provide human-readable
                     documentation for the problem type (e.g. using
                     HTML).
        default: about:blank
      title:
        type: string
        description: A short summary of the problem type, written in
                     English and readable for engineers (usually not
                     suited for non technical stakeholders and not
                     localized).
        example: Service Unavailable
      status:
        type: integer
        format: int32
        description: The HTTP status code generated by the origin server
                     for this occurrence of the problem.
        minimum: 200
        exclusiveMaximum: 600
        example: 503
      detail:
        type: string
        description: A human-readable explanation specific to this
                     occurrence of the problem.
      instance:
        type: string
        format: uri-reference
        description: A URI reference that identifies the specific
                     occurrence of the problem. It may or may not
                     yield further information if dereferenced.
    required:
      - status
      - title
      - type
    additionalProperties: true
```

5.5 Documentation

APIs conformant to this profile MUST be documented using the OpenAPI v3 standard, in the form of an OpenAPI Document. The OpenAPI Document MUST be available under `https://{domain}/{baseURL}/v{version.major}/openapi` following the versioning rules defined in the section on API Deployment Considerations.

Furthermore, the **latest deployed version** of the OpenAPI Document must always be accessible under `https://{domain}/{baseURL}/openapi`

5.6 DISCOVERABILITY

The REST API should provide the proper mechanisms for it to become discoverable both in terms of its structure and operations. For facilitating the discoverability, the API MUST have a complete OpenAPI v3 document accessible under its base URL (see section on Documentation). Furthermore, the OpenAPI Document MUST contain:

- The API name using the `info.title` property;
- The lifecycle status of the API, using the `info.x-edelivery.lifecycle` properties as described in the section on API Lifecycle.

Additionally, to support a common requirement of API repositories, the OpenAPI Document MUST contain the `info.x-edelivery.publisher` object as described below:

Field Name	Type	Description
name	string	The name of the publisher
URL	string	The URL pointing to a web page providing information about the publisher

The following example provides a summary of the fields required for API discoverability:

```
info:
  title: The API
  x-edelivery:
    publisher:
      name: The API Publishing Organisation
      url: http://www.organisation.org/
    lifecycle:
      maturity: deprecated
      deprecatedAt: 2020-12-31
      sunsetAt: 2021-12-31
```

6 API DOCUMENTATION

6.1 INTRODUCTION

This profile section provides a ruleset for documenting a REST API conformant to Level 2 of this profile. For Level 2 profile conformance, the technical documentation of the API MUST be implemented by creating an OpenAPI Document compliant with the OpenAPI v3 standard [OAS-V3] and adhering to the rules described in this profile section.

6.2 DOCUMENTATION RULES

6.2.1 General

The OpenAPI Document openapi field MUST be at least version 3.1.0. It MUST contain an externalDocs entry pointing to the ISA² IPS REST API Core Profile as in the following example.

```
openapi: '3.1.0'
...
externalDocs:
  description: The ISA² IPS REST API Core Profile
  url: https://joinup.ec.europa.eu/collection/api4dt/document/isa2-ips-rest-
api-profile
...
```

6.2.2 Info

The Info object provides metadata about the API that can be used also for discoverability. This profile defines the following specification extensions and rules for the **info** object.

6.2.2.1 Info Object Specification Extensions

6.2.2.1.1 x-edelivery.lifecycle

The x-edelivery.lifecycle object is a specification extension defined in the API Core Profile. Its main purpose is to provide lifecycle metadata of the API such as its maturity, deprecation and sunset. The structure is defined in the API Lifecycle section of the API Core Profile.

6.2.2.1.2 x-edelivery.publisher

The x-edelivery.publisher object is a specification extension defined in the API Core Profile. Its main purpose is to provide metadata about the publisher of the API, which is commonly used in API catalogues for better discoverability. The structure in the Discoverability section of the API Core Profile.

6.2.2.2 Info Object Rules

The Info object provides metadata about the API. The metadata MAY be used by the clients if needed and MAY be presented in editing or documentation generation tools for convenience. The following rules apply in the OpenAPI Info object properties:

The `info.summary` MUST be present and provide a short summary of the API.

The `info.description` SHOULD be present providing an extensive description of the API.

The `info.termsOfService` URL MUST be present and point to the terms of service of the API.

The `info.license` object MUST be present and describe the license of the API.

The `info.version` MUST be present and MUST follow the semantic version formatting of **Major.Minor.Patch**, conforming to the API Versioning Semantics section of the API Core Profile.

The `info.x-edelivery.lifecycle` object MUST be present denoting the current maturity of the API, following the rules defined in the API Lifecycle section of the API Core Profile.

The `info.x-edelivery.publisher` object MUST be present, providing information on the API publisher, according to the Discoverability section of the API Core Profile.

6.2.3 Servers

The **servers** section of the OpenAPI Document provides the necessary connectivity information to a target server. The following rules have been defined:

The `servers` object SHOULD be present providing information on the deployed instances of the API by the publisher (see `info.x-edelivery.publisher`).

For each declared Server object, the `url` MUST start with **HTTPS**, denoting the use of TLS.

6.2.4 Paths

The Paths object describes the endpoints of the API and, as such, it MUST contain at least one Path Item object. It MUST contain either one Path Item object "/openapi.json" and/or one Path Item object "/openapi.yaml", with a parameterless GET operation that responds with the API's current OpenAPI Document, in accordance with the Documentation section of the API Core Profile. Each Path Item included in the Paths object MUST have the `summary` field present, describing the resource representation of the specific path. Path Items MUST also follow the guidelines defined in the URI Structure section of the API Core Profile.

6.2.4.1 Path Item Operations

The Path Item Operations describe the API's input and output expectations. These include the expected parameters, request and response body schema definitions and response codes.

Each Operation object MUST have their `summary` field defined, describing the operation's functionality.

6.2.4.1.1 Operation Response Rules

All operations defined MUST have at least one response with a successful response code (e.g. in the 2xx range) or a redirection response code (e.g. in the 3xx range). When the response code is 200 then a response body MUST be present. When the response code is 204 then a response body MUST NOT be present.

When error responses are being defined in the 400–599 range of HTTP Status codes, with the response body defined, the response body MUST follow the Problem+JSON schema as defined in the Error Messages section of the API Core Profile by creating a reference to the Problem+JSON schema in the components of the OpenAPI Document of the ISA^2 IPS REST API Core Profile.

6.2.4.1.2 Operation Deprecation Rules

When an operation is deprecated, it MUST mark the `deprecated` field as `true`. Furthermore, the **Deprecation** HTTP Response header, as defined in the OpenAPI Document of the ISA^2 IPS REST API Core Profile, MUST be present in the **headers** section of the Operation object, by referencing the corresponding pre-defined header from the OpenAPI Document of the ISA^2 IPS REST API Core Profile.

6.2.4.1.3 Operation Digital Signature Rules

Operations that provide either a detached message signature or a detached payload signature MUST declare it in a Header object in the **headers** section, by referencing the corresponding pre-defined header from the OpenAPI Document of the ISA^2 IPS REST API Core Profile.

Operations that expect either a detached message signature or a detached payload signature MUST declare it as a parameter object, with the "`in`" property value denoted as "`header`", by referencing the corresponding pre-defined header from the OpenAPI Document of the ISA^2 IPS REST API Core Profile.

For operations that include payload signature, the "DetachedPayloadSignature" tag MUST be present in the `tags` array of the operation and the detached payload signature MUST reference the **Edel-Payload-Sig** header in the OpenAPI Document of the ISA^2 IPS REST API Core Profile.

For operations that include message signature, the "DetachedMessageSignature" tag MUST be present in the `tags` array of the operation and the detached message signature MUST reference the **Edel-Message-Sig** header in the OpenAPI Document of the ISA^2 IPS REST API Core Profile.

6.2.5 Security

The **security** section of the OpenAPI Document provides a complete declaration of which security mechanisms can be used across the API. The following rules, conforming to the API Core Profile, MUST be applied:

6.2.5.1 Security Schemes

Security schemes are descriptions of the supported security mechanisms that are supported by the API. The scheme attribute is allowed to have only the "bearer" value when the HTTP Request contains a JWT or SAML Token to be used for authentication and authorization.

6.2.5.2 OAuth Flows

The `flows` field found in the Security Scheme objects typically lists the OAuth flows that are supported by the API. From the supported flows, the API MUST use only the `clientCredentials` and `authorizationCode`. The `implicit` and `password` flows MUST NOT be used.

6.3 THE OPENAPI DOCUMENT OF THE ISA^2 IPS REST API CORE PROFILE

The OpenAPI Document of the ISA^2 IPS REST API Core Profile is a library of pre-defined components. Many of those components are part of the documentation ruleset, referenced as stated in the Documentation Rules section.

6.3.1 Pre-defined components

This section defines the pre-defined components introduced by this profile. These components are referenced in the API Core Profile and the Documentation Rules section.

6.3.1.1 Headers

6.3.1.1.1 Edel-Message-Sig

The **Edel-Message-Sig** header is used to carry a JAdES-compliant JWT detached message signature that an operation expects as a header parameter or returns as a response header. It follows the compact detached representation of JWT as per section 7.1 of [RFC7515].

```
components:
  headers:
    Edel-Message-Sig:
      description: The custom header used for carrying the detached
                   signature for signing the HTTP Message
      schema:
        $ref: '#/components/schemas/JwsCompactDetached'
...
  schemas:
    JwsCompactDetached:
      title: The format for the message-level and payload signature
      description: Defines the string pattern as a regular expression
                   that MUST be followed to represent detached JWS compact
                   tokens
      "$id": https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/api-core-profile/components/schemas/jws-compact-detached.json
      "$schema": https://json-schema.org/draft/2020-12/schema
      type: string
      format: jws-compact-detached
      pattern: ^[A-Za-z0-9_-]+(?:(\\.\\.)[A-Za-z0-9_-]+){1}
```

6.3.1.1.2 Edel-Payload-Sig

The **Edel-Payload-Sig** header is used to carry a JAdES-compliant JWT detached payload signature that an operation expects as a header parameter or returns as a response header. It follows the compact detached representation of JWT as per section 7.1 of [RFC7515].

```
components:
  headers:
    Edel-Payload-Sig:
      description: The custom header used for carrying the detached
                   signature for signing the payload
      schema:
        $ref: '#/components/schemas/JwsCompactDetached'
...
  schemas:
    JwsCompactDetached:
      title: The format for the message-level and payload signature
      description: Defines the string pattern as a regular expression
                   that MUST be followed to represent detached JWS compact
                   tokens
      "$id": https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/api-core-profile/components/schemas/jws-compact-detached.json
      "$schema": https://json-schema.org/draft/2020-12/schema
      type: string
      format: jws-compact-detached
      pattern: ^[A-Za-z0-9_-]+(?:(\\.\\.)[A-Za-z0-9_-]+){1}
```

6.3.1.2 Schemas

6.3.1.2.1 Problem Details object (RFC 7807) Schema

The problem schema is the JSON schema of Problem+JSON [RFC7807], used for expressing in more detail the errors that occurred in an HTTP Response. According to the ruleset, the schema MUST be used when the response code is either a 4xx or a 5xx and the version referenced MUST be the one defined in the OpenAPI Document of the ISA² IPS REST API Core Profile.

The OpenAPI documentation schema for Problem Details object is already defined in the Error Messages section of the API Core Profile.

6.3.1.2.2 JWS Compact Representation Schema

The JWS compact representation schema defines the string pattern as a regular expression, denoting the structure a JWS compact token MUST follow to be a valid compact JWS Representation as per section 7.1 of [RFC7515]. It is used for defining the value of the message-level signature and of the payload signature.

```
JwsCompactDetached:
  title: The format for the message-level and payload signature
  description: Defines the string pattern as a regular expression that
               MUST be followed to represent detached JWS compact tokens
  "$id": https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/api-core-profile/components/schemas/jws-compact-detached.json
  "$schema": https://json-schema.org/draft/2020-12/schema
  type: string
  format: jws-compact-detached
  pattern: ^[A-Za-z0-9_-]+(?:(\\.\\.)[A-Za-z0-9_-]+){1}
```

6.3.2 OpenAPI Document Instance

The OpenAPI document instance provided in the section below can also be downloaded at the following link: openapi.yml.

```
openapi: 3.1.0
```

```
info:
  title: ISA² IPS REST API Core Profile - OpenAPI Document Specification
  summary: OpenAPI Document Specification supporting the ISA² IPS REST API
           Core Profile
  description: This specification provides definitions introduced by the
               ISA² IPS REST API Profile
  contact:
    name: eDelivery support office
    url: https://europa.eu/!BPvjcw
    email: EC-EDELIVERY-SUPPORT@ec.europa.eu
  license:
    name: EUPL-1.2 or later
    url: https://eupl.eu/1.2/en/
  version: 1.0.0
  x-edelivery:
    lifecycle:
      maturity: supported
    publisher:
      name: European Commission
      URL: https://ec.europa.eu/

externalDocs:
  description: The ISA² IPS REST API Core Profile
  url: https://joinup.ec.europa.eu/collection/api4dt/document/isa2-ips-rest-
api-profile

components:
  headers:
    Edel-Message-Sig:
      description: The custom header used for carrying the detached
                   signature for signing the HTTP Message
      schema:
        $ref: '#/components/schemas/JwsCompactDetached'

    Edel-Payload-Sig:
      description: The custom header used for carrying the detached
                   signature for signing the payload
      schema:
        $ref: '#/components/schemas/JwsCompactDetached'

    Deprecation:
      description: Deprecation HTTP Response Header following the
                   Internet-Draft "The Deprecation HTTP Header Field"
                   (https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-
deprecation-header-01)
      schema:
        $ref: '#/components/schemas/DateValue'
    Sunset:
      description: Sunset HTTP Response Header following [RFC8594]. For
                   every operation it MUST be set to the date the API is
                   to be sunset
      schema:
        $ref: '#/components/schemas/DateValue'

  schemas:
    JwsCompactDetached:
      title: The format for the message-level and payload signature
      description: Defines the string pattern as a regular expression that
                   MUST be followed to represent detached JWS compact
                   tokens
      "$id": https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/api-core-profile/components/schemas/jws-compact-detached.json
      "$schema": https://json-schema.org/draft/2020-12/schema
      type: string
      format: jws-compact-detached
      pattern: ^[A-Za-z0-9_-]+(?:(\\.\\.)[A-Za-z0-9_-]+){1}

    DateValue:
      type: string
      format: date

    Problem:
      title: A Problem Details object (RFC 7807) defined by the ISA² IPS
             REST API Core Profile
      description: A Problem Details object (RFC 7807) with ISA² IPS REST
                   API extensions, used for signals (responses) to
                   messages
      "$id": https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/api-core-profile/components/schemas/problem.json
      "$schema": https://json-schema.org/draft/2020-12/schema
      type: object
      properties:
        type:
          type: string
          format: uri
          description: An URI reference that identifies the problem type.
                       When dereferenced, it SHOULD provide human-readable
                       documentation for the problem type (e.g. using
                       HTML).
          default: about:blank
        title:
          type: string
          description: A short summary of the problem type, written in
```

```
                     English and readable for engineers (usually not
                     suited for non-technical stakeholders and not
                     localized).
        example: Service Unavailable
      status:
        type: integer
        format: int32
        description: The HTTP status code generated by the origin server
                     for this occurrence of the problem.
        minimum: 200
        exclusiveMaximum: 600
        example: 503
      detail:
        type: string
        description: A human-readable explanation specific to this
                     occurrence of the problem.
      instance:
        type: string
        format: uri-reference
        description: A URI reference that identifies the specific
                     occurrence of the problem. It may or may not
                     yield further information if dereferenced.
    required:
      - status
      - title
      - type
    additionalProperties: true
```

7 Messaging API Specification

7.1 Introduction

This API Specification defines a **secure, reliable and payload-agnostic protocol** for message exchange using the tools and specifications that are profiled in the API Core Profile, taking into account the light context constraints. The Messaging API Specification defines common messaging patterns to enable a messaging protocol following REST API principles.

This section of the ISA^2 IPS REST API Profile starts by presenting the Message Exchange Patterns and Recipient Addressing Schemes that are covered by the Messaging API Specifications before proceeding to the user and signal Message Specification and then the specification's API Endpoints.

7.2 Message Exchange Patterns

The Message Exchange Patterns define the different ways of executing a business message exchange between two parties. The patterns define the flows and interactions between the parties together with the expected outcomes on each interaction step. Following the paradigms found in current state-of-the-art message exchange protocols, the following patterns are defined:

7.2.1 Send Message with No Response – Push

The "Send Message with No Response – Push" pattern describes the simplest interaction between two parties. This pattern uses the push variation, in which the client directly pushes the message to be transferred to the server, with the following steps:

1. The Client initiates the communication by creating a message containing the **message payload** provided by the Submitter (1)
2. The Client submits the **message** to the receiving Server (2)
3. The Server receives the message from the client and validates it. Upon validation it sends: (3)
    a. An **acknowledgement**, if the message is valid
    b. An **error signal**, if the message is invalid
4. The Server forwards the **message payload** to the Receiver (4)

The following response signals are applicable to this message exchange pattern:

- step 3: Message Accepted
- step 3: Message Rejected: Invalid/Duplicate Message ID
- step 3: Message Rejected: Invalid Message Signature
- step 3: Message Rejected: Invalid Addressing

- step 3: Message Rejected: Invalid Message Format

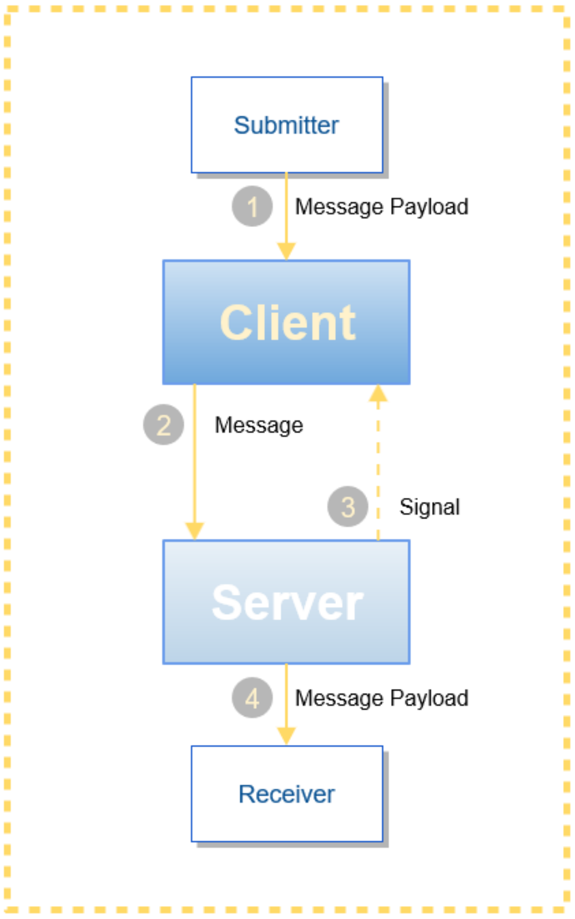

Figure 9 Send Message with No Response – Push

7.2.2 Send Message with No Response – Pull

This pattern provides an alternative to the "Send Message with No Response – Push" pattern. In this pattern, the Receiver sends a **pull signal** using its Client to a Server holding a message for delivery. It comprises the following steps:

1. The Submitter creates and stores a **message payload** on the Server for asynchronous pulling by the Submitter (1)

2. The Client of the Receiver initiates the communication, on behalf of the Receiver, and makes a message request by sending a **pull signal** to the Server of the Submitter (2)

3. The Server receives the pull signal from the Client and validates it. Upon validation it sends (3):

    a. The **message**, if the pull signal is valid

    b. An **error signal**, if the pull signal is invalid

4. The Client forwards the **message payload** to the Receiver (4)

The following response signals are applicable to this message exchange pattern:

- step 3: Pull Error: No final recipient configured for the pulling user
- step 3: Pull Error: No Message Found
- step 3: Pull Error: Unauthorized

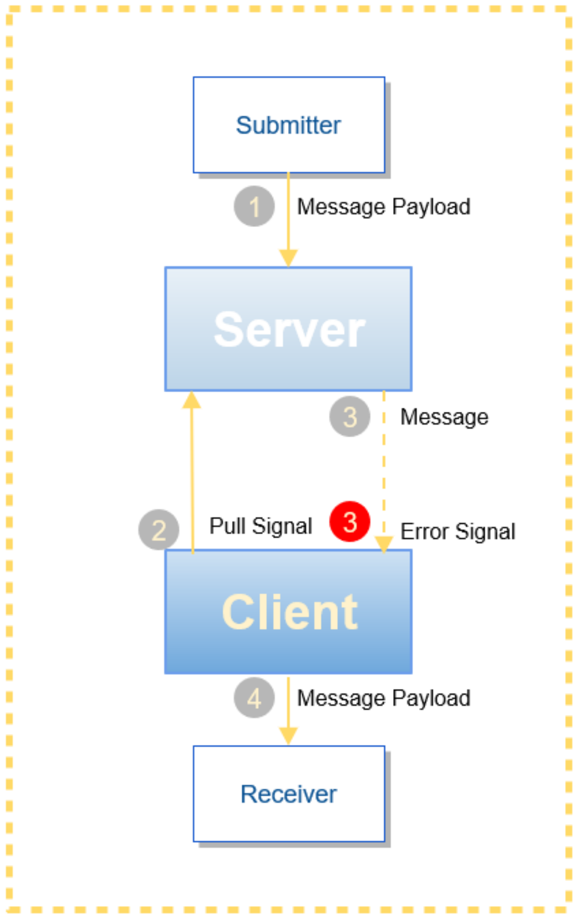

Figure 10 Send Message with No Response – Pull

7.2.3 Send Message with Synchronous Response

The "Send Message with Synchronous Response" is a typical HTTP Request / Response message. This pattern expects a complete request-response flow in a single HTTP connection. The following steps define the pattern interactions:

1. The Client initiates the communication by creating a message containing the **message payload** provided by the Requester (1)

2. The Client submits the **message** to the receiving Server (2)

3. The Server receives the message from the Client and validates it. Upon validation:
   a. It forwards the **message payload** to the Responder, if the message is valid (3)
   b. It responds directly with an **error signal** and skips to step 6, if the message is not valid (5)

4. The Responder receives the message payload and sends the **response payload** to the Server (4)

5. The Server creates and sends a **response message** as response, synchronously (5)

6. The Client receives the response message, validates it, and forwards the **message payload** to the Requester (6)

The following response signals are applicable to this message exchange pattern:

- step 3: Message Rejected: Invalid/Duplicate Message ID
- step 3: Message Rejected: Invalid Message Signature
- step 3: Message Rejected: Invalid Addressing
- step 3: Message Rejected: Invalid Message Format

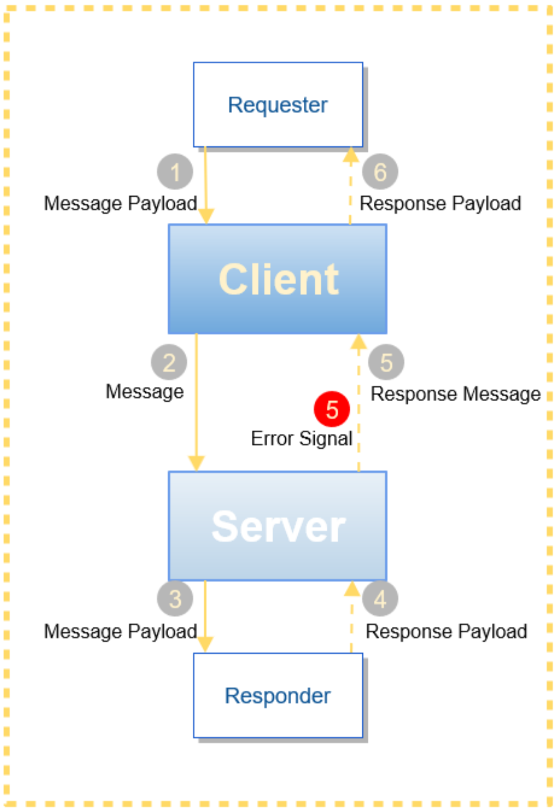

Figure 11 Send Message with Synchronous Response

7.2.4 Send Message with Asynchronous Response – Push and Pull

This message pattern combines the "Send Message With No Response – Push" and "Send Message With No Response – Pull". The Push pattern is used for submitting the message and the Pull pattern is then used to provide a response message back. The following steps define the message pattern:

1. The Client initiates the communication by creating a message containing the **message payload** provided by the Requester (1)

2. The Client submits the **message** to the receiving Server (2)

3. The Server receives the message from the client and validates it. Upon validation it sends: (3)
   a. An **acknowledgement**, if the message is valid
   b. An **error signal**, if the message is invalid

4. The Server forwards the **message payload** to the Responder (4)

5. The Responder creates and stores a **response payload** on the Server for asynchronous pulling by the Requester (5)

6. The Client initiates the communication, on behalf of the Requester, and makes a message request by sending a **pull signal** to the Server (6)

7. The Server receives the pull signal from the client and validates it. Upon validation it sends (7)
   a. The **response message**, if the pull signal is valid
   b. An **error signal**, if the pull signal is invalid

8. The Client forwards the **response payload** to the Requester (8)

The following response signals are applicable to this message exchange pattern:

- step 3: Message Accepted
- step 3: Message Rejected: Invalid/Duplicate Message ID
- step 3: Message Rejected: Invalid Message Signature
- step 3: Message Rejected: Invalid Addressing
- step 3: Message Rejected: Invalid Message Format

- step 7: Pull Error: No final recipient configured for the pulling user
- step 7: Pull Error: No Message Found
- step 7: Pull Error: Unauthorized

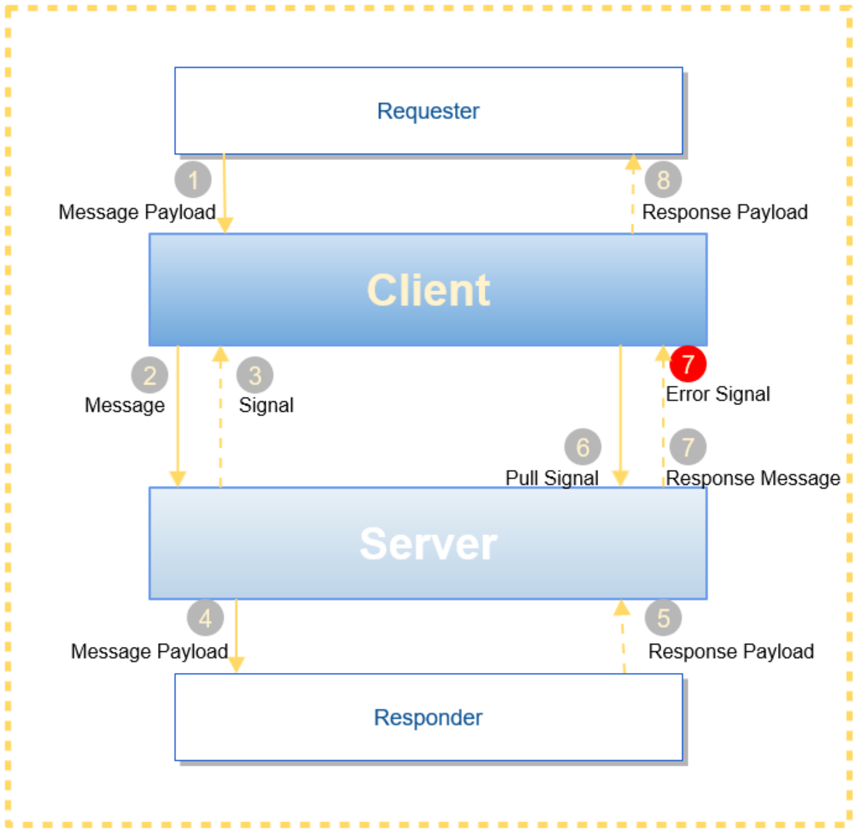

Figure 12 Send Message with Asynchronous Response – Push and Pull

7.2.5 Send Message with Asynchronous Response – Push and Webhook Pull

This message pattern extends the "Send Message with Asynchronous Response – Push and Pull" with a webhook provided by the client. The webhook is used to signal the Requester when the response is ready to be retrieved and then the Requester uses the "Send Message with No Response – Pull" to do so. The following steps define the message pattern:

1. The Client initiates the communication by creating a message containing the **message payload** provided by the Requester (1)

2. The Client submits the **message** to the Server, together with a webhook (2)

3. The Server receives the message from the client and validates it. Upon validation it sends: (3)
   a. An **acknowledgement**, if the message is valid

   b. An **error signal**, if the message is invalid

4. The Server forwards the **message payload** to the Responder (4)

5. The Responder creates and stores a **response payload** on the Server for asynchronous pulling by the Requester (5)

6. The Server sends a **signal** to the designated webhook URL (6)

7. The Client initiates the communication, on behalf of the Requester, and makes a message request by sending a **pull signal** to the Server (7)

8. The Server receives the pull signal from the client and validates it. Upon validation it sends (8)
   a. The **response message**, if the pull signal is valid
   b. An **error signal**, if the pull signal is invalid

9. The Client forwards the **response payload** to the Requester (9)

The following response signals are applicable to this message exchange pattern:

- step 3: Message Accepted
- step 3: Message Rejected: Invalid/Duplicate Message ID
- step 3: Message Rejected: Invalid Message Signature
- step 3: Message Rejected: Invalid Addressing
- step 3: Message Rejected: Invalid Message Format
- step 6: Message Response is ready
- step 8: Pull Error: No final recipient configured for the pulling user
- step 8: Pull Error: No Message Found
- step 8: Pull Error: Unauthorized

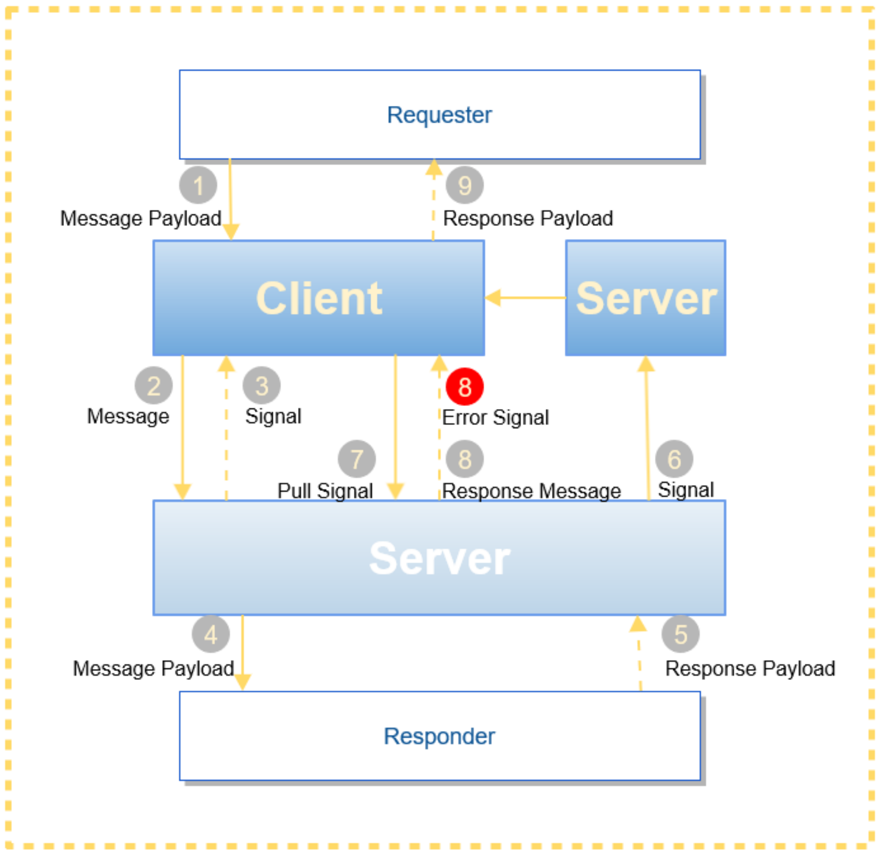

Figure 13 Send Message with Asynchronous Response – Push and Webhook Pull

7.2.6 Send Message with Asynchronous Response – Push and Webhook Push

This message pattern allows a push and push pattern using a provided webhook by the client. The webhook is used to send a response message to the Requester using the "Send Message with No Response – Push" pattern. The following steps define the message pattern:

1. The Client initiates the communication by creating a message containing the **message payload** provided by the Requester (1)

2. The Client submits the **message** to the receiving Server together with a webhook (2)

3. The Server receives the message from the client and validates it. Upon validation it sends: (3)

    a. An **acknowledgement**, if the message is valid

    b. An **error signal**, if the message is invalid

4. The Server forwards the **message payload** to the Responder (4)

5. The Server initiates the communication by creating a message containing the **response payload** provided by the Responder (5)

6. The Server submits the **response message** to the provided webhook (6)

7. The Webhook Server receives the response message from the Server and validates it. Upon validation it sends: (7)
    a. An **acknowledgement**, if the response message is valid
    b. An **error signal**, if the response message is invalid

8. The Webhook Server forwards the **response payload** to the Requester through the Client (8)

The following response signals are applicable to this message exchange pattern:

- steps 3, 7: Message Accepted
- steps 3, 7: Message Rejected: Invalid/Duplicate Message ID
- steps 3, 7: Message Rejected: Invalid Message Signature
- steps 3, 7: Message Rejected: Invalid Addressing
- steps 3, 7: Message Rejected: Invalid Message Format

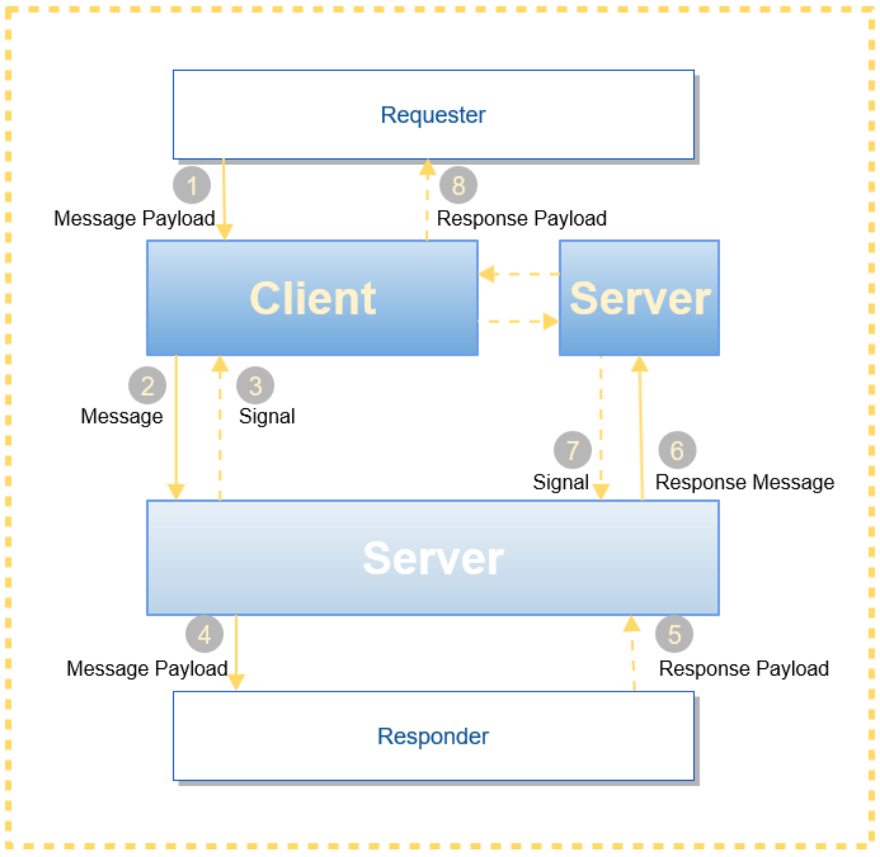

Figure 14 Send Message with Asynchronous Response – Push and Webhook Push

7.2.7 Send Message with Asynchronous Response – Pull and Push

This pattern provides an alternative to the "Send Message with Asynchronous Response – Push and Pull" pattern. In this pattern, the Responder sends a **pull signal** using its Client to a Server holding a message for delivery and sends a response message using push. It comprises the following steps:

1. The Requester creates and stores a **message payload** on the Server for asynchronous pulling by the Responder (1)

2. The Client initiates the communication, on behalf of the Responder, and makes a message request by sending a **pull signal** to the Server (2)

3. The Server receives the pull signal from the Client and validates it. Upon validation it sends: (3)

    a. The **message**, if the pull signal is valid

    b. An **error signal**, if the pull signal is invalid

4. The Client forwards the **message payload** to the Responder (4)

5. The Client initiates the communication by creating a response message containing the **response payload** provided by the Responder (5)

6. The Client submits the **response message** to the Server (6)

7. The Server receives the response message and validates it. Upon validation it sends: (7)
    a. An **acknowledgement**, if the response message is valid
    b. An **error signal**, if the response message is invalid

8. The Server forwards the **response payload** to the Requester(8)

The following response signals are applicable to this message exchange pattern:

- step 3: Pull Error: No final recipient configured for the pulling user
- step 3: Pull Error: No Message Found
- step 3: Pull Error: Unauthorized
- step 7: Message Accepted
- step 7: Message Rejected: Invalid/Duplicate Message ID
- step 7: Message Rejected: Invalid Message Signature
- step 7: Message Rejected: Invalid Addressing
- step 7: Message Rejected: Invalid Message Format

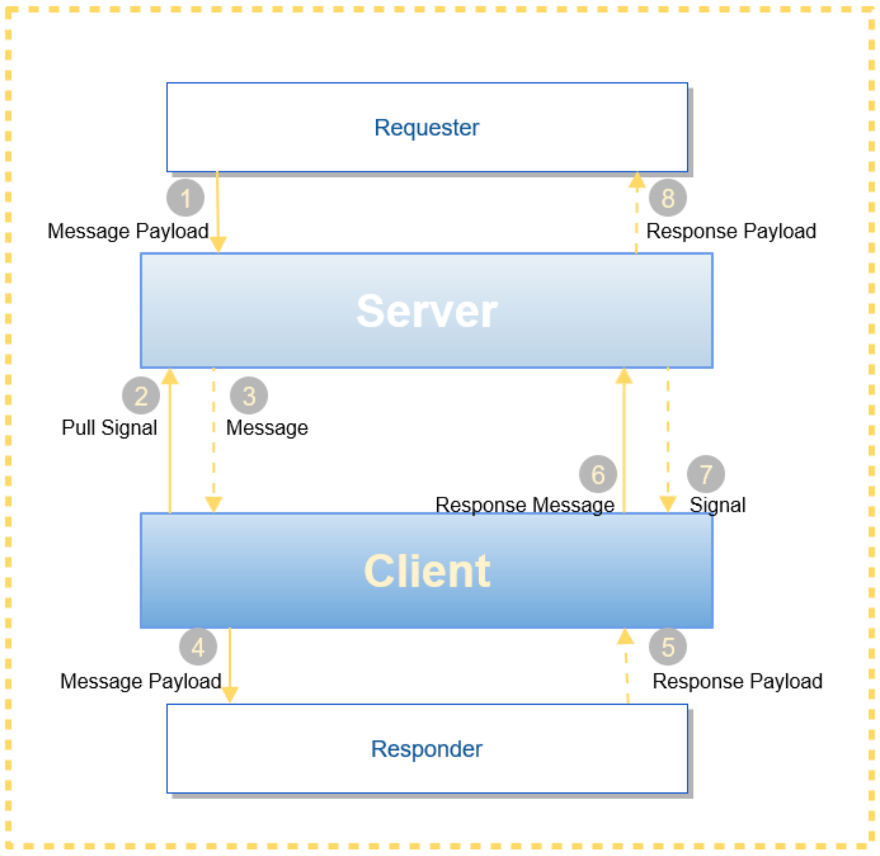

Figure 15 Send Message with Asynchronous Response – Pull and Push

7.3 Recipient Addressing Schemes

The message exchange patterns describe the exchange between a single HTTP client and a single secure resource server that implements the messaging endpoints. However, there could be situations where the message has more than one recipient or even an unknown number of recipients, acting as a broadcast message. The Messaging API Specification caters for the following recipient addressing variations:

7.3.1 Single Known Recipient

The Single Known Recipient scheme is the baseline addressing scheme, in which the sender is aware of the recipient and is explicitly providing it, using the Final-Recipient HTTP Header Field (see Common Message Fields section).

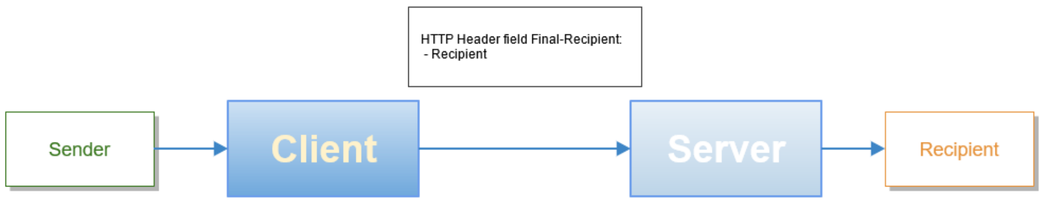

Figure 16 Single Known Recipient

7.3.2 Multiple Known Recipients

The Multiple Known Recipients scheme is an extension of the baseline scheme. This scheme allows the sender to address **multiple** known recipients by explicitly listing their identifiers in the Final-Recipient HTTP Header Field (see Common Message Fields section).

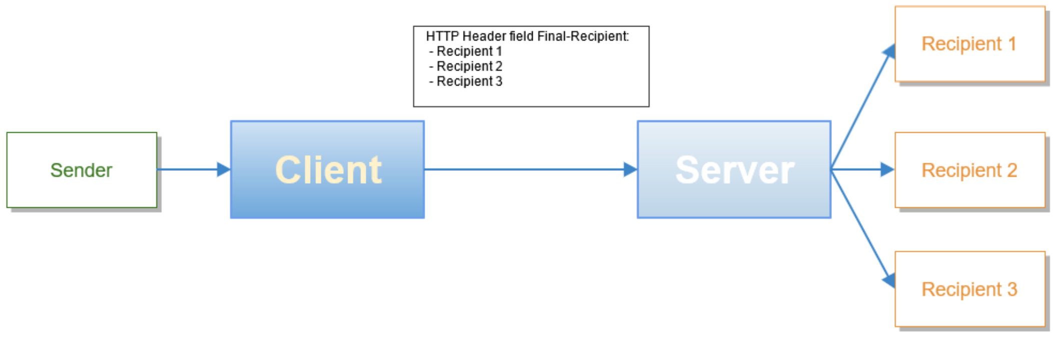

Figure 17 Multiple Known Recipients

7.3.3 Unknown Recipients

The Unknown Recipients scheme is a variation of the baseline scheme. When this scheme is used, the sender is not aware of the recipient(s) and thus cannot provide the proper identifiers for them, but can use an abstract identifier (e.g., a role, a group, etc.) that can be resolved by the resource server to one or more concrete recipients.

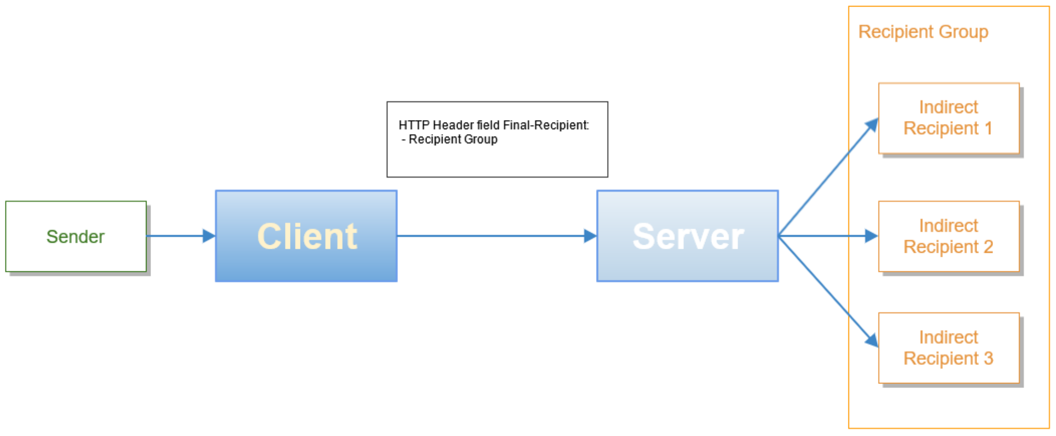

Figure 18 Unknown Recipients

7.4 Message Specification

7.4.1 Common Message Fields

The messaging API defines a set of metadata to be reused in the endpoint definitions of the APIs. This metadata carries the necessary information for secure and reliable messaging. Each endpoint defined in the API Endpoints section uses a subset of the metadata defined in this section. Table 1 provides a complete list of the metadata used in the Messaging API

Specification. Whether a field is mandatory or optional is defined alongside each specific endpoint in the API Endpoints section.

Metadata	Description	Format	Location
Original-Sender	A string identifying the Original Sender	String	HTTP Field
Original-Sender-Token	The ID Token proving the identity of the Original Sender	JWT or OIDC Token	HTTP Field
Final-Recipient	A string identifying the Final Recipient	String	HTTP Field
Timestamp	The timestamp of the message generation	Date	HTTP Field
Edel-Message-Sig	The detached JAdES signature signing the message to be sent	JWT Compact	HTTP Field
Edel-Payload-Sig	The detached JAdES signature signing the payload to be sent	JWT Compact	HTTP Field
Response-Webhook	The URL to which the server will send the response	URL	HTTP Field
Signal-Webhook	The URL to which the server will send the signal	URL	HTTP Field
Message-Id	The unique identifier of the response message received or the unique identifier of the signal message received	String	HTTP Field
Service	The service of the response message received	String	HTTP Field
Action	The action related to the service of the response message received	String	HTTP Field
messageId	The unique identifier of the message being submitted	String	Resource URL
service	The service the message is submitted to	String	Resource URL
action	The action related to the service the message is submitted to	String	Resource URL
rMessageId	The unique identifier of the response message being submitted	String	Resource URL
rService	The service under which the response message should be submitted	String	Resource URL
rAction	The action of the service under which the response message should be submitted	String	Resource URL

Table 1 Messaging API Metadata

7.4.1.1 *Original-Sender*

The Original Sender is the entity that initiates the submission of the message. The value provided in this field MUST identify a single entity. The representation of the original sender is out of scope of the current specification.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP message; if an HTTP message contains multiple Original-Sender header fields, the receiver MUST consider the HTTP message invalid. An invalid HTTP *request* message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

The information provided in the field is meant to identify the sender of the message, therefore data protection considerations apply.

7.4.1.2 *Original-Sender-Token*

The Original Sender represents the authenticated entity acting as the user who sends the message using the client. Following the API Core Profile, the original sender MUST be identified either via an OpenID Connect Token or via a signed JSON Web Token, any of which is represented as a Compact JWT token. This token MUST be carried under the Original-Sender-Token HTTP Field.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP message; if an HTTP message contains multiple Original-Sender-Token header fields, the receiver MUST consider the HTTP message invalid. An invalid HTTP *request* message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

The information provided in the field is meant to identify the sender of the message, therefore data protection considerations apply.

7.4.1.3 *Final-Recipient*

The Final Recipient is(are) the entity(ies) to which the message is addressed. A single identifier or multiple identifiers can be provided as the value for this field, as detailed in the Recipient Addressing Schemes section. If multiple identifiers are present, they MUST be separated by commas. The representation of the final recipient is out of scope of the current specification.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP message; if an HTTP message contains multiple Final-Recipient header fields, the receiver MUST consider the HTTP message invalid. An invalid HTTP *request* message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

The information provided in the field is meant to identify the receiver(s) of the message, therefore data protection considerations apply.

7.4.1.4 MessageId

The messageId is the unique identifier of the message submitted. It MUST be defined by the client. It is used for reliable messaging for guaranteeing the at-most-once message submission (no duplicate message-ids are allowed by the server implementing the API).

7.4.1.5 Timestamp

The Timestamp is the date (and optionally the time) at which the message was generated, encoded as specified by [RFC3339]. It is provided by the client and verified by the server.

Notes:

- refer also to the erratum id 5624 of [RFC3339] that clarifies that using date-time is not mandatory; other choices such as full-date or partial-time are equally allowed.
- clock skew requirements are out of scope of the current specification, but implementers are encouraged to define them as applicable to their specific domain.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP message. If an HTTP message contains multiple Timestamp header fields, the receiver MUST consider the HTTP message invalid. An invalid HTTP *request* message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.1.6 Edel-Message-Sig

The Edel-Message-Sig carries the signature of the **HTTP Message** following the API Core Profile on Message-Level Security. Following the light context constraints, the signature is optional for the client messages, but is RECOMMENDED for server messages.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP message. If an HTTP message contains multiple Edel-Message-Sig header fields, the receiver MUST consider the HTTP message invalid. An invalid HTTP *request* message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.1.7 Edel-Payload-Sig

The Edel-Payload-Sig carries the signature of a **subpart of the Multipart message** (see User Message section) following the API Core Profile on Payload Security.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear among the global header fields of the HTTP message and MUST not appear more than once among the header fields of each HTTP message subpart. Otherwise the receiver MUST consider the HTTP message invalid. An invalid HTTP *request* message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message subpart that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.1.8 Service

The service metadata defines the service under which the message should be submitted. It is combined with the **action** metadata to provide a complete domain-level message target. The service metadata MUST be a URL Safe string.

7.4.1.9 Action

The action metadata defines the action of the service under which the message should be submitted. It is combined with the **service** metadata to provide a complete domain-level target of the message. The action metadata MUST be a URL Safe string.

7.4.1.10 RMessageId

The rMessageId is the identifier of the response message being submitted. It MUST be generated by the client submitting the response message.

7.4.1.11 Message-Id

When the rMessageId URL parameter cannot be used for providing the identifier of the response message, e.g. in synchronous responses, the Message-Id header field can be used instead. It identifies either a response or a signal message received synchronously.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP response message. If an HTTP response message contains multiple Message-Id header fields, the receiver MUST consider the HTTP response message invalid.

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.1.12 RService

The response service metadata defines the service under which the response message should be submitted. It is combined with the **response action** metadata to provide a complete domain-level message target. The response service metadata MUST be a URL Safe string.

7.4.1.13 Service

When, in synchronous responses, the rService URL parameter cannot be used for providing the service of the response message, the Service header field can be used instead for the purpose. It is combined with the action from the Action header field to provide a complete domain-level message target.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP response message. If an HTTP response message contains multiple Service header fields, the receiver MUST consider the HTTP response message invalid.

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.1.14 RAction

The response action metadata defines the action of the service under which the response message should be submitted. It is combined with the **response service** metadata to provide a complete domain-level target of the response message. The response action metadata MUST be a URL Safe string.

7.4.1.15 Action

When, in synchronous responses, the rAction URL parameter cannot be used for providing the action of the service of the response message, the Action header field can be used instead for the purpose. It is combined with the service from the Service header field to provide a complete domain-level message target.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP response message. If an HTTP response message contains multiple Action header fields, the receiver MUST consider the HTTP response message invalid.

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.1.16 Response-Webhook

The Response-Webhook provides the callback URL that the server MUST use for sending a response. It is required when implementing the "Send Message with Asynchronous Response – Push and Webhook Push" exchange pattern.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP request message. If an HTTP request message contains multiple Response-Webhook header fields, the receiver MUST consider the HTTP request message invalid. An invalid HTTP request message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.1.17 Signal-Webhook

The Signal-Webhook provides the callback URL that the server MUST use for sending a signal back. It is required when implementing the "Send Message with Asynchronous Response – Push and Webhook Pull" exchange pattern.

The API Endpoints section documents under what conditions the field should be used. The header field MUST not appear more than once in an HTTP request message. If an HTTP request message contains multiple Signal-Webhook header fields, the receiver MUST consider the HTTP request message invalid. An invalid HTTP request message MUST be rejected using the "Invalid Message Format" signal message (see Signal Message section).

The information conveyed through this field applies only to the HTTP message that contains it.

Intermediaries MUST NOT modify the field's value and the field name MUST NOT be listed in the Connection header field.

No special security considerations apply to the information provided in the field.

7.4.2 User Message

The User Message is a payload-agnostic message container that contains the common message fields that are part of the HTTP Header and payload-specific metadata for each included payload. Following the API Core Profile, its structure MUST always be a multipart/mixed, containing at least one multipart subpart as payload (see Figure 19: User Message High-Level Structure).

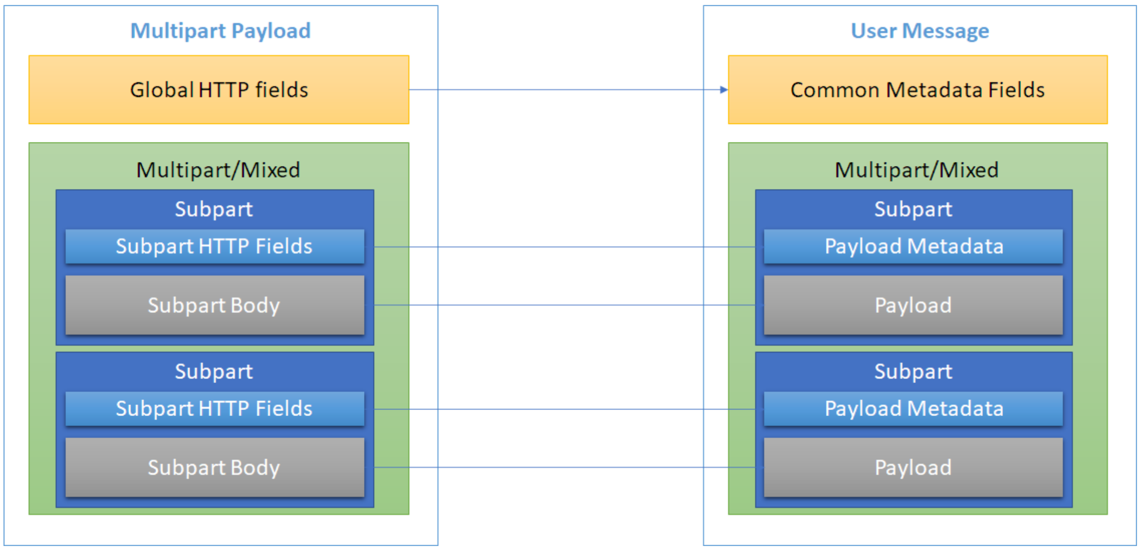

Figure 19 User Message High-Level Structure

The User Message structure follows the structure depicted in Figure 19: User Message High-Level Structure. The Common Metadata Fields are provided in the Global HTTP Fields of the Multipart Payload, as described in the Common Message Fields section of the profile.

Each User Message consists of one or more multipart subparts. Each subpart represents a payload. Each payload has its own payload metadata that are specific to this payload:

- Payload content-type
- Payload content-length
- Payload content-disposition
- Payload signature (Edel-Payload-Sig)

These payload metadata are represented as subpart HTTP header fields, with the payload itself being the subpart body.

The payload signature is expressed using the JAdES detached profile, as profiled in the API Core Profile. The following snippet provides an example of a User Message containing two payloads:

```
POST /my-service/my-action/dde12f67-c391-4851-8fa2-c07dd8532efd HTTP/1.1
Content-Type: multipart/mixed; boundary=REST-API-BOUNDARY
Content-Length: 5142342
Authorization: Bearer
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
Original-Sender: 1111:123456789
Original-Sender-Token:
eyJhbGciOiJIUzINiIsnRI6IkpXVCJ9.eyJzdWIiOiIxM0NTY3ODkwIiwibmIiwiaWF0IoxNTE2MjM5MDIyf
Q.SflKxwRJSMeKKF2QT4fwpMePOk6yJV_adQssw5c
Final-Recipient: 9999::333222111
Timestamp: 2021-03-11T07:00:27Z
Digest: sha-
256=eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ=
Edel-Message-Sig:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

--REST-API-BOUNDARY
Content-Disposition: name="datafile1"; filename="r.pdf"
Content-Type: application/pdf
Edel-Payload-Sig:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

R0lGODlhIAAgAPcAAP///+/v7/f39+/n5/fv7//39/fn597OzufW1tbGxs69vffe3u/W1salpZQQ
EIQYEJwQCJQYCJwYCM6clL2Ee5wpGIwhEK0YANatpdalnHMYCKUhCJQYAKUYAM6Ec5QhCJwhCPfW
zu/Oxta1rb17a6UhAK0hAPe9rb2Ec7V7a71rUpwpCL0xCJwhAOfOxta9tcatpdallM6cjMaUhK1C
IZwxELU5EK0xCLUxCN61pcacjK1rUrVaOaVKKZxCIZwxCKUxCL05CLUxAM6llO+1nMZzUr1rSrVS
KaVCGK1CGLVCEJQxCK05CLU5CIwpAKUxAK0xAM69tffWxrVrSrVaMbVKGMZKEL1CCLU5AP/v597O
xufGtcallNaljLWEa96ce9aUc717Wq1aMcZjMbVaKbVSIa1KGL1SGJw5CK1CCLVCCMZKCM5KCMZC
APfn3tbGvc57Ss5aGMZSENZaEL1KCM5SCLVCAMZKAPfezvfOte/GrdatlN6cc9aEUr1rOc5jIdZj
GL1SEM5aELVKCMZSCNZaCL1KAOfOvc57Qs5jGNZjEM5aCMZSAN5aAO/OtefGrcZjGM5jEN5rELVS
--REST-API-BOUNDARY
Content-Disposition: name="datafile2"; filename="g.gif"
Content-Type: image/gif
Edel-Payload-Sig:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

GIF87a............,...........D..;
--REST-API-BOUNDARY--
```

7.4.3 Signal Message

Signal messages are messages representing confirmations of message reception, an error state after message submission or a notification that a message response is ready to be retrieved. Use of signal messages is required and depends on the messaging pattern being implemented.

Contrary to User Messages, Signal messages MUST NOT use a multipart media type for structuring the HTTP request or response.

Signal messages derive and further extend Problem+JSON [RFC7807] to contain messaging-specific domain attributes. These attributes are:

- **instance:** The instance attribute uses the relative URI structure to provide the message metadata like the service, action and message UUID for reference.
- **digest:** The message digest of the received HTTP message for which the signal was created. Digest MUST be formatted following [RFC3230] (and [DRAFT-IETF-HTTP-DGST-HDR] that will obsolete [RFC3230]). Used in combination with the signature, it provides non-repudiable evidence of reception.

The following block provides the JSON Schema for the Signal Message:

```
{
  "title": "A Problem Details object (RFC 7807) defined by the ISA² IPS\
            REST API Core Profile",
  "description": "A Problem Details object (RFC 7807) with ISA² IPS REST\
                  API extensions, used for signals (responses) to\
                  messages",
  "$id": "https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/messaging-api-specification/components/schemas/signal-
message.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "type": {
      "type": "string",
      "format": "uri",
      "description": "An URI reference that identifies the problem type.\
                      When dereferenced, it SHOULD provide human-readable\
                      documentation for the problem type (e.g. using\
                      HTML).",
      "example": "https://github.com/isa2-api4ips/rest-api-
profile/blob/main/messaging-api-specification/signal.md#message-accepted"
    },
    "title": {
      "type": "string",
      "description": "A short summary of the problem type, written in\
                      English and readable for engineers (usually not\
                      suited for non technical stakeholders and not\
                      localized).",
      "example": "Message Accepted"
    },
    "status": {
      "type": "integer",
      "format": "int32",
      "description": "The HTTP status code generated by the origin server\
                      for this occurrence of the problem.",
      "minimum": 200,
      "exclusiveMaximum": 600,
      "example": 202
    },
    "detail": {
      "type": "string",
      "description": "A human-readable explanation specific to this\
                      occurrence of the problem."
    },
    "instance": {
      "type": "string",
      "format": "uri-reference",
      "description": "A URI reference that identifies the specific\
                      occurrence of the problem. It may or may not yield\
                      further information if dereferenced."
    },
    "digest": {
      "type": "string",
      "description": "The digest of the received message using the\
                      notation proposed in 'Digest Fields'\
                      (https://datatracker.ietf.org/doc/html/draft-ietf-
httpbis-digest-headers).",
      "example": "sha-256=4REjxQ4yrqUVicfSKYNO/cF9zNj5ANbzgDZt3/h3Qxo=,"
    }
  },
  "required": ["title", "type", "status", "instance"],
  "additionalProperties": false
}
```

The Messaging API Specification mandates the use of Problem+JSON to send back error responses. It further mandates the same for signalling a successful accepted submission by a server as well as a readiness of a message response. To this end, it extends the scope of [RFC7807], as it uses the same schema. The precision definition of how it should be used is defined in the following section.

The following block provides an example of a successful Signal Message (Message Accepted) sent as an acknowledgement:

```
HTTP/1.1 202 Accepted
Content-Type: application/problem+json; charset=utf-8
Message-Id: 1814964a-5a1c-4c4b-839f-c4629be5db0c
Timestamp: 2021-03-11T07:00:27Z
Digest: sha-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Edel-Message-Sig:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_a
dQssw5c
{
    "type": "https://github.com/isa2-api4ips/rest-api-
profile/blob/main/messaging-api-specification/signal.md#message-accepted",
    "title": "Message Accepted",
    "status": 202,
    "instance": "/my-service/my-action/dde12f67-c391-4851-8fa2-
c07dd8532efd",
    "digest": "sha-
256=eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MD
IyfQ"
}
```

7.4.3.1 Predefined Signals

The following pre-defined signals MUST be supported by both the client and the server implementing the Messaging API Specification:

No	Signal Title	Signal Status (code)	Signal Type	Signal Description
1	Message Accepted	202	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#message-accepted	Sent when the message is properly validated. It may include a status monitor that can provide the user with an estimate of when the request will be fulfilled (see [RFC7231])
2	Validation Failed	400	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#message-validation-failed	Sent when the message fails the validation process
3	Invalid or Duplicate Message ID	400	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#invalid-message-id	Sent when the MessageId is not valid
4	Invalid Message Signature	400	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#invalid-message-signature	Sent when the message signature cannot be verified
5	Invalid Addressing	400	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#invalid-addressing	Sent when the Original Sender or Final Recipient(s) cannot be resolved
6	Invalid Message Format	400	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#invalid-format	Sent when the message format does not adhere to the specification
7	Pull Error: No final recipient configured for the pulling user	400	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#pull/no-final-recipient	Sent when the server cannot resolve/match the pulling user to a final recipient

8	Pull Error: No Message Found	404	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#pull/no-message-found	Sent when no message is found that maps to the pull request
9	Pull Error: Unauthorized	401	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#pull/unauthorized	Sent when the pull request is unauthorized
10	Message Response is ready	201	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#message-ready	An HTTP Request following [RFC7807] MUST be sent when a message response is ready to be retrieved

For signals number 1-9, the HTTP Response code MUST match the Signal Status (code) defined in the above table.

The following pre-defined signal SHOULD be supported by servers implementing the Messaging API Specification for 500 server errors that do not affect the generation, signature and submission of such a signal:

11	Server Error	500	https://github.com/isa2-api4ips/rest-api-profile/blob/main/messaging-api-specification/signal.md#server-error	Sent when a server error occurred that does not affect the signalling process

Otherwise a simple 500 HTTP Response code will be returned.

7.4.3.2 Signature profile for User and Signal Messages

The Messaging API Specification follows the the API Core Profile on Message-Level Security. The Messaging API Specification adds the following headers in the calculation of the signature, by mandating that they MUST be added in the `pars` array of the `SigD` object of the JAdES detached signature if they exist in the message:

- Original-Sender
- Original-Sender-Token
- Final-Recipient
- Message-Id
- Service
- Action
- Timestamp

7.4.4 Message Reference

The Message Reference is a response provided by the resource server when multiple messages could be retrieved from an API operation, e.g. getting all messages for a specific service or for a specific combination of service and action. The Message Reference follows the following JSON Schema:

```
{
  "title": "A Message Reference object defined by the ISA² IPS REST API\
            Messaging API Specification",
  "description": "A Message Reference object to be used when multiple\
                  messages could be retrieved from an API operation",
  "$id": "https://raw.githubusercontent.com/isa2-api4ips/rest-api-
profile/main/messaging-api-specification/components/schemas/message-reference-
list.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": [
    "MessageReferenceList"
  ],
  "properties": {
    "MessageReferenceList": {
      "type": "array",
      "items": {
        "title": "Items",
        "type": "object",
        "required": [
          "service",
          "action",
          "messageId",
          "href"
        ],
        "properties": {
          "service": {
            "description": "The service the message belongs to",
            "type": "string"
          },
          "action": {
            "description": "The action the message belongs to",
            "type": "string"
          },
          "messageId": {
            "description": "The unique identifier of a message",
            "type": "string",
            "format": "uuid"
          },
          "href": {
            "description": "The direct link for getting the specific\
                          message",
            "type": "string",
            "format": "uri-reference"
          }
        }
      }
    }
  }
}
```

The following block shows an example of a Message Reference:

```
{
  "MessageReferenceList": [
    {
      "service": "my-service",
      "action": "my-action",
      "messageId": "dde12f67-c391-4851-8fa2-c07dd8532efd",
      "href": "/my-service/my-action/dde12f67-c391-4851-8fa2-c07dd8532efd"
    },
    {
      "service": "my-service",
      "action": "my-action",
      "messageId": "68258b84-7806-4446-971f-3c8ddeb7b093",
      "href": "/my-service/my-action/68258b84-7806-4446-971f-3c8ddeb7b093"
    }
  ]
}
```

7.5 API ENDPOINTS

The profile makes use of the semantics of resource patterns, following the REST API Design principles as described in the API Core Profile, using conformant HTTP Methods, HTTP Request and Response Fields, and URL fields for the definition of the Submission (Push) endpoints and Request for Reception (Pull) endpoints of the Messaging API Specification. It defines three groups of API endpoints, each containing a set of endpoints for a specific purpose:

- **Message Submission Endpoints**: used for submitting messages from the client to the server. It consists of:
  - **Message Submission Endpoint**: used for submitting asynchronous messages.
  - **Message Submission with Synchronous Response Message Endpoint**: used for submitting messages that expect a synchronous response.
- **Response Submission Endpoints**: used for submitting a message which is a response to a message. It consists of:

  - **Response Message Submission Endpoint**: used for submitting asynchronous response messages.
  - **Webhook Response Message Submission**: used for submitting asynchronous response messages to a webhook server.
  - **Webhook Signal Submission Endpoint**: used for signalling that a response message is available.
- **Message Pull Endpoints**: used for pulling messages from a server asynchronously. It consists of:
  - **Get Message Reference List Endpoint**: used for getting a listing of retrievable messages, using references.
  - **Get Message Endpoint**: used for retrieving a message from a server.
- **Response Message Pull Endpoints**: used for pulling messages from a server asynchronously, that are responses to a previously sent message. It consists of:
  - **Get Response Message Reference List Endpoint**: used for getting a listing of retrievable response messages, using references.
  - **Get Response Message Endpoint**: used for retrieving a response message from the server.

The profile defines **all endpoints as optional,** as which endpoints are needed is completely dependent on the Message Exchange Pattern being implemented. Each Message Exchange Pattern as defined in the Message Exchange Patterns section can be fully implemented only when specific endpoints are implemented either by the message receiver or the message response receiver. The following table shows a mapping between the Message Exchange Patterns and the endpoint(s) they require. Table cells indicate the role of the party, with respect to the **initial message**, which needs to provide the endpoint:

Mapping: Message Exchange Patterns to Endpoints	Message Submission	Message Submission with Sync Response Message	Response Message Submission	Webhook Response Message Submission	Webhook Signal Submission	Get Message Reference List	Get Message	Get Response Message Reference List	Get Response Message
Send Message with No Response – Push	Message Receiver								
Send Message with No Response – Pull						Message Sender	Message Sender		
Send Message with Synchronous Response		Message Receiver							
Send Message with Asynchronous Response – Push and Pull	Message Receiver							Message Receiver	Message Receiver
Send Message with Asynchronous Response – Push and Webhook Pull	Message Receiver				Message Sender			Message Receiver	Message Receiver
Send Message with Asynchronous Response – Push and Webhook Push	Message Receiver			Message Sender					
Send Message with Asynchronous Response – Pull and Push			Message Sender			Message Sender	Message Sender		

Table 2 Mapping: Message Exchange Patterns to Endpoints. Table cells indicate the role of the party, with respect to the initial message, which needs to provide the endpoint.

7.5.1 Message Submission Endpoints

7.5.1.1 Authorisation of HTTP Requests

The server responsible for implementing the endpoints defined in this section MUST ensure that the (OAuth) access token used to authorize the message submission matches the OpenID Connect Token of the JSON Web Token received in the Original-Sender-Token field of the submitted message. The approach to ensure the matching is out of scope of this specification. One option would be to compare the identity from a JWT-type access token to that included in the aforementioned Original-Sender-Token field.

7.5.1.2 Message Submission Endpoint

The **Message Submission** endpoint is the main endpoint of the Messaging API. It provides the endpoint to which a client sends the message, as created by the original sender. Table 3 provides an overview of the HTTP Multipart body and fields defined and required for the implementation of this endpoint.

Resource Attributes	Resource Value		
HTTP Method	`POST`		
URL Pattern	`/messaging/{service}/{action}/{messageId}`		
HTTP Body (Content-Type)	`multipart/mixed` containing the `boundary` directive		
Successful HTTP Response (Content-Type)	`application/problem+json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service the message is submitted to	URL Safe String	Mandatory

action	A representation of the action related to the service the message is submitted to	URL Safe String	Mandatory
messageId	The identifier of the message being submitted. It MUST be generated by the client submitting the message	UUID v4 String	Mandatory
Global HTTP Request Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender	String	Mandatory
Original-Sender-Token	The ID Token proving the identity of the Original Sender	Compact JWT	Mandatory
Final-Recipient	The representation of the Final Recipient(s)	String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the sent message	JAdES Compliant Compact Detached JWS	Optional
Response-Webhook	The URL to which the Server will send the response. The URL MUST include the URI resource fragment up to and including the `response` resource path. The `rService`, `rAction` and `rMessageId` MUST be added by the Server.	URL	Optional
Signal-Webhook	The URL to which the Server will send the signal that the response is ready. The URL MUST include the complete URL up to and including the `signal` resource path.	URL	Optional
Multipart HTTP Request Subpart Header Fields	**Resource Value**	**Format**	**Optionality**

Edel-Payload-Sig	The detached JAdES signature signing the subpart of the multipart of the sent message	JAdES Compliant Compact Detached JWS	Optional
Content-Disposition	The Content-Disposition header, declaring the subpart as an attachment	fixed value: 'Attachment'	Mandatory
Content-Type	The content type of the subpart of the multipart message	One of IANA Media Types	Mandatory
HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the signal message	JAdES Compliant Compact Detached JWS	Optional

Table 3 Field and attribute overview of the Message Submission endpoint

The endpoint implements the message submission which is part of the several "Message Submission with No Response" and the "Message Submission with Asynchronous Response" message exchange patterns. It expects a multipart message, following the User Message section, and sends back a response following the Problem+JSON json schema, as defined in the Signal Message section.

The URL Fields `service` and `action` MUST be provided at design time by the receiving server, through its OpenAPI Document. The `messageId` MUST be kept as a URL Field that will be provided by the client submitting the message.

The Client MUST provide the Original-Sender and the Original-Sender-Token HTTP Header Fields expressed as a compact JWT which can be either an OpenID Connect Token or a simple JSON Web Token signed by a trusted certificate. When the token is an OpenID Connect Token, it must be provided by the OpenID Connect Identity Provider stated in the OpenAPI Document of the Messaging API. It MUST also provide the final recipient of the message in the Final-Recipient HTTP Header Field. The Final-Recipient Header MUST follow the Recipient Addressing Schemes section. The structure and format of the final recipient are **domain-specific** and **out of scope** of this specification.

The Client, when possible, MAY provide one or more signatures following the API Core Profile specification on the Message And Payload Level Security, creating a JAdES Compliant Compact Detached JWS.

The Server responds with a Signal Message as defined in the Signal Message section, both for successful transmission and error transmission.

7.5.1.3 Message Submission with Synchronous Response Message Endpoint

The **Message Submission with Synchronous Response Message** endpoint is the synchronous alternative to the main endpoint of the Messaging API. It provides the endpoint to which a client sends the message, as created by the original sender, and for which a synchronous response message is expected. Table 4 provides an overview of the HTTP Multipart body and fields, for both the request and the response, defined and required for the implementation of this endpoint.

Resource Attributes	Resource Value		
HTTP Method	`POST`		
URL Pattern	`/messaging/{service}/{action}/{messageId}/sync`		
HTTP Body (Content-Type)	`multipart/mixed` containing the `boundary` directive		
Successful HTTP Response (Content-Type)	`multipart/mixed` containing the `boundary` directive		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service the message is submitted to	URL Safe String	Mandatory
action	A representation of the action related to the service the message is submitted to	URL Safe String	Mandatory

messageId	The identifier of the message being submitted. It MUST be generated by the client submitting the message	UUID v4 String	Mandatory
Global HTTP Request Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender	String	Mandatory
Original-Sender-Token	The ID Token proving the identity of the Original Sender	Compact JWT	Mandatory
Final-Recipient	The representation of the Final Recipient(s)	String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the sent message	JAdES Compliant Compact Detached JWS	Optional
Multipart HTTP Request Subpart Header Fields	**Resource Value**	**Format**	**Optionality**
Edel-Payload-Sig	The detached JAdES signature signing the subpart of the multipart of the sent message	JAdES Compliant Compact Detached JWS	Optional
Content-Disposition	The Content-Disposition header, declaring the subpart as an attachment	fixed value: 'Attachment'	Mandatory
Content-Type	The content type of the subpart of the multipart message	One of IANA Media Types	Mandatory
HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender	String	Mandatory
Original-Sender-Token	The ID Token proving the identity of the Original Sender	Compact JWT	Mandatory

Final-Recipient	The representation of the Final Recipient(s)	String	Mandatory
Message-Id	The identifier of the response message	UUID v4 String	Mandatory
Service	A representation of the service the response message	String	Mandatory
Action	A representation of the action related to the service of the response message	String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the sent message	JAdES Compliant Compact Detached JWS	Optional
Multipart HTTP Response Subpart Header Fields	**Resource Value**	**Format**	**Optionality**
Edel-Payload-Sig	The detached JAdES signature signing the subpart of the multipart of the response message	JAdES Compliant Compact Detached JWS	Optional
Content-Disposition	The Content-Disposition header, declaring the subpart as an attachment	fixed value: 'Attachment'	Mandatory
Content-Type	The content type of the subpart of the multipart message	One of IANA Media Types	Mandatory

Table 4 Field and attribute overview of the Message Submission with Synchronous Response Message endpoint

The endpoint implements the message submission which is part of the "Message Submission with Synchronous Response" message exchange pattern. It expects a multipart message, following the User Message section, and sends back a response following the User Message section when successful or the Problem+JSON json schema when an error occurs, as defined in the Signal Message section.

The URL Fields `service` and `action` MUST be provided at design time by the receiving server, through its OpenAPI Document. The `messageId` MUST be kept as a URL Field that will be provided by the client submitting the message.

The Client MUST provide the Original-Sender and the Original-Sender-Token HTTP Header Fields expressed as a compact JWT which can be either an OpenID Connect Token or a simple JSON Web Token signed by a trusted certificate. When the token is an OpenID Connect Token, it must be provided by the OpenID Connect Identity Provider stated in the OpenAPI Document of the Messaging API. It MUST also provide the final recipient of the message in the Final-Recipient HTTP Header Field. The Final-Recipient Header MUST follow the Recipient Addressing Schemes section. The structure and format of the final recipient are **domain-specific** and **out of scope** of this specification.

The Client, when possible, MAY provide one or more signatures following the API Core Profile specification on the Message And Payload Level Security, creating a JAdES Compliant Compact Detached JWS.

The Server responds with:

- A User Message as defined in the User Message section, on a successful transmission.
- A Signal Message as defined in the Signal Message section, on an error transmission.

7.5.2 Response Message Submission Endpoints

The endpoints defined in this section enable pushing or signalling the availability of responses to initial messages to a Sever or Webhook Server.

The Webhook Server is a server that is coupled with the Client and that is limited to providing the webhook endpoints for response and/or signal submission.

As an initial message can potentially have multiple final recipients, there may be multiple response messages for a given initial message, each having its own response message identifier. The response messages may be issued under specific services and actions that are potentially different from those of the initial message.

7.5.2.1 Authorisation of HTTP Requests

The server responsible for implementing the endpoints defined in this section MUST ensure that the (OAuth) access token used to authorize the response message submission matches:

- the OpenID Connect Token or the JSON Web Token received in the Original-Sender-Token field of the submitted response message;
- (one of) the Final Recipient(s) of the initial message to which the submitted response message is replying.

The approach to ensure the matching is out of scope of this specification.

7.5.2.2 Response Message Submission Endpoint

The **Response Message Submission** endpoint is the endpoint used for sending response messages in reply to a previously submitted message. It provides the endpoint to which a client sends the response message, as created by the final recipient. Table 3 provides an overview of the HTTP Multipart body and fields defined and required for the implementation of this endpoint.

Resource Attributes	Resource Value		
HTTP Method	`POST`		
URL Pattern	`/messaging/{service}/{action}/{messageId}/response/{rService}/{rAction}/{rMessageId}`		
HTTP Body (Content-Type)	`multipart/mixed` containing the `boundary` directive		
Successful HTTP Response (Content-Type)	`application/problem+json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service of the **initial message** the response message responds to	URL Safe String	Mandatory
action	A representation of the action related to the service of the **initial message** the response message responds to	URL Safe String	Mandatory
messageId	The identifier of the **initial message** the response message responds to	UUID v4 String	Mandatory
rService	A representation of the service the **response message** is submitted to	URL Safe String	Mandatory

rAction	A representation of the action related to the service the **response message** is submitted to	URL Safe String	Mandatory
rMessageId	The identifier of the **response message** being submitted. It MUST be generated by the client submitting the response message	UUID v4 String	Mandatory
Global HTTP Request Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender of the **response message.** It MUST match (one of) the Final Recipient(s) of the initial message	String	Mandatory
Original-Sender-Token	The ID Token proving the identity of the Original Sender of the **response message**	Compact JWT	Mandatory
Final-Recipient	The representation of the Final Recipient of the **response message**. It MUST match the Original Sender of the initial message	String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the sent response message	JAdES Compliant Compact Detached JWS	Optional
Multipart HTTP Request Subpart Header Fields	**Resource Value**	**Format**	**Optionality**
Edel-Payload-Sig	The detached JAdES signature signing the subpart of the multipart of the sent response message	JAdES Compliant Compact Detached JWS	Optional
Content-Disposition	The Content-Disposition header, declaring the subpart as an attachment	fixed value: 'Attachment'	Mandatory

Content-Type	The content type of the subpart of the multipart message	One of IANA Media Types	Mandatory
HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the signal message	JAdES Compliant Compact Detached JWS	Optional

Table 5 Field and attribute overview of the Response Message Submission endpoint

The endpoint implements the response message submission which is part of the "Message Submission with Asynchronous Response – Pull and Push" message exchange pattern. It expects a multipart message, following the User Message section, and sends back a response following the Problem+JSON json schema, as defined in the Signal Message section.

The URL Fields `service`, `action`, `rService` and `rAction` MUST be provided at design time by the receiving server, through its OpenAPI Document. The `messageId` and `rMessageId` MUST be kept as URL Fields that will be provided by the client submitting the response message.

The Client MUST provide the Original-Sender and the Original-Sender-Token HTTP Header Fields expressed as a compact JWT which can be either an OpenID Connect Token or a simple JSON Web Token signed by a trusted certificate. When the token is an OpenID Connect Token, it must be provided by the OpenID Connect Identity Provider stated in the OpenAPI Document of the Messaging API. It MUST also provide the final recipient of the message in the Final-Recipient HTTP Header Field. The Final-Recipient Header MUST follow the Recipient Addressing Schemes section. The structure and format of the final recipient are **domain-specific** and **out of scope** of this specification.

The Client, when possible, MAY provide one or more signatures following the API Core Profile specification on the Message And Payload Level Security, creating a JAdES Compliant Compact Detached JWS.

The Server responds with a Signal Message as defined in the Signal Message section, both for successful transmission and error transmission.

7.5.2.3 Webhook Response Message Submission Endpoint

The **Webhook Response Message Submission** endpoint is the webhook endpoint used for sending response messages in reply to a previously submitted message. It provides the endpoint to which a server sends the response message, as created by the final recipient. Table 6 provides an overview of the HTTP Multipart body and fields defined and required for the implementation of this endpoint.

Resource Attributes	Resource Value		
HTTP Method	`POST`		
URL Pattern	`/messaging-webhook/{messageId}/response/{rService}/{rAction}/{rMessageId}`		
HTTP Body (Content-Type)	`multipart/mixed` containing the `boundary` directive		
Successful HTTP Response (Content-Type)	`application/problem+json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
messageId	The identifier of the **initial message** the response message responds to	UUID v4 String	Mandatory
rService	A representation of the service the **response message** is submitted to	URL Safe String	Mandatory
rAction	A representation of the action related to the service the **response message** is submitted to	URL Safe String	Mandatory
rMessageId	The identifier of the **response message** being submitted. It MUST be generated by the server submitting the response message	UUID v4 String	Mandatory

Global HTTP Request Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender of the **response message**. It MUST match (one of) the Final Recipient(s) of the initial message	String	Mandatory
Original-Sender-Token	The ID Token proving the identity of the Original Sender of the **response message**	Compact JWT	Mandatory
Final-Recipient	The representation of the Final Recipient of the **response message**. It MUST match the Original Sender of the initial message	String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the sent response message	JAdES Compliant Compact Detached JWS	Optional
Multipart HTTP Request Subpart Header Fields	**Resource Value**	**Format**	**Optionality**
Edel-Payload-Sig	The detached JAdES signature signing the subpart of the multipart of the sent response message	JAdES Compliant Compact Detached JWS	Optional
Content-Disposition	The Content-Disposition header, declaring the subpart as an attachment	fixed value: 'Attachment'	Mandatory
Content-Type	The content type of the subpart of the multipart message	One of IANA Media Types	Mandatory
HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory

Edel-Message-Sig	The detached JAdES signature signing the signal message	JAdES Compliant Compact Detached JWS	Optional

Table 6 Field and attribute overview of the Webhook Response Message Submission endpoint

The endpoint implements the webhook response message submission which is part of the "Message Submission with Asynchronous Response – Push and Webhook Push" message exchange pattern. It expects a multipart message, following the User Message section, and sends back a response following the Problem+JSON json schema, as defined in the Signal Message section.

The URL Fields `rService` and `rAction` MUST be provided at design time by the server receiving the initial message, through its OpenAPI Document. The `messageId` and `rMessageId` MUST be kept as URL Fields that will be provided by the server submitting the response message.

The Client MUST provide the Original-Sender and the Original-Sender-Token HTTP Header Fields expressed as a compact JWT which can be either an OpenID Connect Token or a simple JSON Web Token signed by a trusted certificate. When the token is an OpenID Connect Token, it must be provided by the OpenID Connect Identity Provider stated in the OpenAPI Document of the Messaging API. It MUST also provide the final recipient of the message in the Final-Recipient HTTP Header Field. The Final-Recipient Header MUST follow the Recipient Addressing Schemes section. The structure and format of the final recipient are **domain-specific** and **out of scope** of this specification.

The Server MAY provide one or more signatures following the API Core Profile specification on the Message And Payload Level Security, creating a JAdES Compliant Compact Detached JWS.

The Webhook Server responds with a Signal Message as defined in the Signal Message section, both for successful transmission and error transmission.

7.5.2.4 Webhook Signal Submission Endpoint

The **Webhook Signal Submission** endpoint is the webhook endpoint used for signalling the availability of a response message in reply to a previously submitted message. It provides the endpoint to which a server signals the availability of the response message. Table 7 provides an overview of the HTTP Multipart body and fields defined and required for the implementation of this endpoint.

Resource Attributes	Resource Value
HTTP Method	`POST`
URL Pattern	`/messaging-webhook/{messageId}/response/signal`

HTTP Body (Content-Type)	`application/problem+json`		
Successful HTTP Response (Content-Type)	`N/A`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
messageId	The Identification of the **initial message** the response message responds to	UUID v4 String	Mandatory
Global HTTP Request Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender of the **response message.** It MUST match (one of) the Final Recipient(s) of the initial message	String	Mandatory
Original-Sender-Token	The ID Token proving the identity of the Original Sender of the **response message**	Compact JWT	Optional
Final-Recipient	The representation of the Final Recipient of the **response message**. It MUST match the Original Sender of the initial message	String	Mandatory
Timestamp	The timestamp of the signal generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the sent signal message	JAdES Compliant Compact Detached JWS	Optional
HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Mandatory

Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the signal message	JAdES Compliant Compact Detached JWS	Optional

Table 7 Field and attribute overview of the Webhook Signal Submission endpoint

The endpoint implements the webhook signal submission which is part of the "Message Submission with Asynchronous Response – Push and Webhook Pull" message exchange pattern. It expects a single-part message, following the Signal Message section, and sends back a response following the Problem+JSON json schema, as defined in the Signal Message section.

The signal "Message Response is ready", as defined in the Predefined Signals section, MUST be used in the HTTP Request.

The Client MUST provide the Original-Sender and the Original-Sender-Token HTTP Header Fields expressed as a compact JWT which can be either an OpenID Connect Token or a simple JSON Web Token signed by a trusted certificate. When the token is an OpenID Connect Token, it must be provided by the OpenID Connect Identity Provider stated in the OpenAPI Document of the Messaging API. It MUST also provide the final recipient of the message in the Final-Recipient HTTP Header Field. The Final-Recipient Header MUST follow the Recipient Addressing Schemes section. The structure and format of the final recipient are **domain-specific** and **out of scope** of this specification.

The Server MAY provide a signature following the API Core Profile specification on the Message-Level Security, creating a JAdES Compliant Compact Detached JWS.

The Webhook Server responds with a Signal Message as defined in the Signal Message section, both for successful transmission and error transmission.

7.5.3 Message Pull endpoints

Following the light context constraints, the specification needs to provide a mechanism for a Client to receive messages in cases where the Original Sender is on the Server side. In such cases, the resource server must implement the operations defined hereunder.

7.5.3.1 Authorisation of HTTP Requests

The server responsible for implementing the endpoints defined in this section MUST ensure that the (OAuth) access token used to authorize the message pull request matches the Final Recipient of all the messages included in the message reference list or of the full message returned. The approach to ensure the matching is out of scope of this specification. One option would be to use the access token to retrieve the identity of the user making the pull request from the authorization server and compare it to the aforementioned Final Recipient.

7.5.3.2 Get Message Reference List Endpoint

This endpoint returns a list of message references available for pulling, following the Message Reference schema. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value		
HTTP Method	`GET`		
URL Pattern	`/messaging`		
HTTP Request Body (Content-Type)	`N/A`		
Successful HTTP Response (Content-Type)	`application/json`		
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Conditional
Timestamp	The timestamp of the signal message generation	Date	Conditional
Edel-Message-Sig	The detached JAdES signature signing the message	JAdeS Compliant Compact Detached JWS	Optional

The endpoint implements the querying mechanism in support of message pulling which is part of the "Send Message with No Response – Pull" and the "Send Message with Asynchronous Response – Pull and Push" message exchange patterns.

The Server responds with:

- A list of message references, following the Message Reference schema, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Message-Id and Timestamp are mandatory only in case the Server responds with a Signal Message.

7.5.3.3 Get Message Reference List for service Endpoint

This endpoint returns a list of message references available for pulling for a specific `service`, following the Message Reference schema. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value		
HTTP Method	`GET`		
URL Pattern	`/messaging/{service}`		
HTTP Request Body (Content-Type)	`N/A`		
Successful HTTP Response (Content-Type)	`application/json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service of the message(s) to be pulled	URL Safe String	Mandatory
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Conditional
Timestamp	The timestamp of the signal message generation	Date	Conditional
Edel-Message-Sig	The detached JAdES signature signing the message	JAdeS Compliant Compact Detached JWS	Optional

The endpoint implements the querying mechanism in support of message pulling which is part of the "Send Message with No Response – Pull" and the "Send Message with Asynchronous Response – Pull and Push" message exchange patterns.

The Server responds with:

- A list of message references, following the Message Reference schema, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Message-Id and Timestamp are mandatory only in case the Server responds with a Signal Message.

7.5.3.4 Get Message Reference List for service and action Endpoint

This endpoint returns a list of message references available for pulling for a specific `service` and `action`, following the Message Reference schema. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value		
HTTP Method	`GET`		
URL Pattern	`/messaging/{service}/{action}`		
HTTP Request Body (Content-Type)	`N/A`		
Successful HTTP Response (Content-Type)	`application/json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service of the message(s) to be pulled	URL Safe String	Mandatory
action	A representation of the action related to the service of the message(s) to be pulled	URL Safe String	Mandatory
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**

Message-Id	The identifier of the signal message	UUID v4 String	Conditional
Timestamp	The timestamp of the signal message generation	Date	Conditional
Edel-Message-Sig	The detached JAdES signature signing the message	JAdeS Compliant Compact Detached JWS	Optional

The endpoint implements the querying mechanism in support of message pulling which is part of the "Send Message with No Response – Pull" and the "Send Message with Asynchronous Response – Pull and Push" message exchange patterns.

The Server responds with:

- A list of message references, following the Message Reference schema, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Message-Id and Timestamp are mandatory only in case the Server responds with a Signal Message.

7.5.3.5 Get Message Endpoint

This endpoint returns the **message** filed under a specific `service` and `action`, following the format defined in the User Message section. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value
HTTP Method	`GET`
URL Pattern	`/messaging/{service}/{action}/{messageId}`
HTTP Request Body (Content-Type)	`N/A`
Successful HTTP Response (Content-Type)	`multipart/mixed` containing the `boundary` directive

Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service of the message being pulled	URL Safe String	Mandatory
action	A representation of the action related to the service of the message being pulled	URL Safe String	Mandatory
messageId	The identifier of the message being pulled. A valid identifier MUST be provided by the client pulling the message	UUID v4 String	Mandatory
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender	String	Conditional
Original-Sender-Token	The ID Token proving the identity of the Original Sender	Compact JWT	Conditional
Final-Recipient	The representation of the Final Recipient(s)	String	Conditional
Message-Id	The identifier of the message	UUID v4 String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the message	JAdES Compliant Compact Detached JWS	Optional
Multipart HTTP Response Subpart Header Fields	**Resource Value**	**Format**	**Optionality**
Edel-Payload-Sig	The detached JAdES signature signing the subpart of the multipart of the message	JAdES Compliant Compact Detached JWS	Optional

Content-Disposition	The Content-Disposition header, declaring the subpart as an attachment	fixed value: 'Attachment'	Mandatory
Content-Type	The content type of the subpart of the multipart message	One of IANA Media Types	Mandatory

The endpoint implements the message pulling mechanism which is part of the "Send Message with No Response – Pull" and the "Send Message with Asynchronous Response – Pull and Push" message exchange patterns.

The Server responds with:

- A User Message as defined in the User Message section, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Original-Sender, Original-Sender-Token and Final-Recipient response header fields are mandatory only in case the Server responds with a User Message.

7.5.4 Response Message Pull endpoints

The endpoints defined in this section enable Clients that cannot accommodate a webhook-based push mechanism to request messages that are responses to initial messages sent by Original Senders through the Client.

For obtaining a response message, the Client needs to be aware of the following:

- The service and action of the initial message
- The message identifier of the initial message
- The service and action of the response message
- The message identifier of the response message

As an initial message can potentially have multiple final recipients, there may be multiple response messages for a given initial message, each having its own response message identifier. The response messages may be issued under specific services and actions that are potentially different from those of the initial message.

To facilitate response message discovery, the several API endpoints have been defined.

7.5.4.1 Authorisation of HTTP Requests

The server responsible for implementing the endpoints defined in this section MUST ensure that the (OAuth) access token used to authorize the response message pull request matches the Final Recipient of all the response messages included in the message reference list or of the full response message returned. The approach to ensure the matching is out of scope of this specification. One option would be to use the access token to retrieve the identity of the user making the pull request from the authorization server and compare it to the aforementioned Final Recipient.

7.5.4.2 Get Response Message Reference List Endpoint

This endpoint returns a list of response message references available for pulling, following the Message Reference schema, representing responses to a previous message sent by the original sender. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value		
HTTP Method	`GET`		
URL Pattern	`/messaging/{service}/{action}/{messageId}/response`		
HTTP Request Body (Content-Type)	`N/A`		
Successful HTTP Response (Content-Type)	`application/json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service of the **initial message** the response message(s) to be pulled pertain(s) to	URL Safe String	Mandatory

action	A representation of the action related to the service of the **initial message** the response message(s) to be pulled pertain(s) to	URL Safe String	Mandatory
messageId	The identifier of the **initial message** the response message(s) to be pulled pertain(s) to	UUID v4 String	Mandatory
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Conditional
Timestamp	The timestamp of the signal message generation	Date	Conditional
Edel-Message-Sig	The detached JAdES signature signing the message	JAdES Compliant Compact Detached JWS	Optional

The endpoint implements the querying mechanism in support of response message pulling which is part of the "Send Message with Asynchronous Response – Push and Pull" and the "Send Message with Asynchronous Response – Push and Webhook Pull" message exchange patterns.

The Server responds with:

- A list of response message references, following the Message Reference schema, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Message-Id and Timestamp are mandatory only in case the Server responds with a Signal Message.

7.5.4.3 Get Response Message Reference List for service Endpoint

This endpoint returns a list of response message references available for pulling for a specific `service`, following the Message Reference schema, representing responses to a previous message sent by the original sender. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value
HTTP Method	`GET`

URL Pattern	`/messaging/{service}/{action}/{messageId}/response/{rService}`		
HTTP Request Body (Content-Type)	`N/A`		
Successful HTTP Response (Content-Type)	`application/json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service of the **initial message** the response message(s) to be pulled pertain(s) to	URL Safe String	Mandatory
action	A representation of the action related to the service of the **initial message** the response message(s) to be pulled pertain(s) to	URL Safe String	Mandatory
messageId	The identifier of the **initial message** the response message(s) to be pulled pertain(s) to	UUID v4 String	Mandatory
rService	A representation of the service of the **response message(s)** to be pulled	URL Safe String	Mandatory
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Conditional
Timestamp	The timestamp of the signal message generation	Date	Conditional
Edel-Message-Sig	The detached JAdES signature signing the message	JAdES Compliant Compact Detached JWS	Optional

The endpoint implements the querying mechanism in support of response message pulling which is part of the "Send Message with Asynchronous Response – Push and Pull" and the "Send Message with Asynchronous Response – Push and Webhook Pull" message exchange patterns.

The Server responds with:

- A list of response message references, following the Message Reference schema, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Message-Id and Timestamp are mandatory only in case the Server responds with a Signal Message.

7.5.4.4 Get Response Message Reference List for service and action Endpoint

This endpoint returns a list of response message references available for pulling for a specific `service` and `action`, the Message Reference schema, representing responses to a previous message sent by the original sender. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value		
HTTP Method	`GET`		
URL Pattern	`/messaging/{service}/{action}/{messageId}/response/{rService}/{rAction}`		
HTTP Request Body (Content-Type)	`N/A`		
Successful HTTP Response (Content-Type)	`application/json`		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**

service	A representation of the service of the **initial message** the response message(s) to be pulled pertain(s) to	URL Safe String	Mandatory
action	A representation of the action related to the service of the **initial message** the response message(s) to be pulled pertain(s) to	URL Safe String	Mandatory
messageId	The identifier of the **initial message** the response message(s) to be pulled pertain(s) to	UUID v4 String	Mandatory
rService	A representation of the service of the **response message(s)** to be pulled	URL Safe String	Mandatory
rAction	A representation of the action related to the service of the **response message(s)** to be pulled	URL Safe String	Mandatory
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Message-Id	The identifier of the signal message	UUID v4 String	Conditional
Timestamp	The timestamp of the signal message generation	Date	Conditional
Edel-Message-Sig	The detached JAdES signature signing the message	JAdES Compliant Compact Detached JWS	Optional

The endpoint implements the querying mechanism in support of response message pulling which is part of the "Send Message with Asynchronous Response – Push and Pull" and the "Send Message with Asynchronous Response – Push and Webhook Pull" message exchange patterns.

The Server responds with:

- A list of response message references, following the Message Reference schema, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Message-Id and Timestamp are mandatory only in case the Server responds with a Signal Message.

7.5.4.5 Get Response Message Endpoint

This endpoint returns the **message** filed under a specific `service` and `action`, following the format defined in the User Message section, representing a response to a previous message sent by the original sender. The following table summarises the endpoint operation parameters:

Resource Attributes	Resource Value		
HTTP Method	`GET`		
URL Pattern	`/messaging/{service}/{action}/{messageId}/response/{rService}/{rAction}/{rMessageId}`		
HTTP Request Body (Content-Type)	`N/A`		
Successful HTTP Response (Content-Type)	`multipart/mixed` containing the `boundary` directive		
Error HTTP Response (Content-Type)	`application/problem+json`		
URL Fields	**Resource Value**	**Format**	**Optionality**
service	A representation of the service of the **initial message** the response message being pulled pertains to	URL Safe String	Mandatory
action	A representation of the action related to the service of the **initial message** the response message being pulled pertains to	URL Safe String	Mandatory
messageId	The identifier of the **initial message** the response message being pulled pertains to	UUID v4 String	Mandatory
rService	A representation of the service of the **response message** being pulled	URL Safe String	Mandatory

rAction	A representation of the action related to the service of the **response message** being pulled	URL Safe String	Mandatory
rMessageId	The identifier of the **response message** being pulled. A valid identifier MUST be provided by the client pulling the resposne message	UUID v4 String	Mandatory
Global HTTP Response Header Fields	**Resource Value**	**Format**	**Optionality**
Original-Sender	The representation of the Original Sender of the **response message**	String	Conditional
Original-Sender-Token	The ID Token proving the identity of the Original Sender of the **response message**	Compact JWT	Conditional
Final-Recipient	The representation of the Final Recipient(s) of the **response message**	String	Conditional
Message-Id	The identifier of the message	UUID v4 String	Mandatory
Timestamp	The timestamp of the message generation	Date	Mandatory
Edel-Message-Sig	The detached JAdES signature signing the message	JAdES Compliant Compact Detached JWS	Optional
Multipart HTTP Response Subpart Header Fields	**Resource Value**	**Format**	**Optionality**
Edel-Payload-Sig	The detached JAdES signature signing the subpart of the multipart of the message	JAdES Compliant Compact Detached JWS	Optional
Content-Disposition	The Content-Disposition header, declaring the subpart as an attachment	fixed value: 'Attachment'	Mandatory
Content-Type	The content type of the subpart of the multipart message	One of IANA Media Types	Mandatory

The endpoint implements the querying mechanism in support of response message pulling which is part of the "Send Message with Asynchronous Response – Push and Pull" and the "Send Message with Asynchronous Response – Push and Webhook Pull" message exchange patterns.

The Server responds with:

- A User Message as defined in the User Message section, on success.
- A Signal Message as defined in the Signal Message section, on error.

The Original-Sender, Original-Sender-Token and Final-Recipient response header fields are mandatory only in case the Server responds with a User Message.

8 High-Security Enhancement

8.1 Introduction

The API Core Profile defines guidelines on how Message-Level Security, Payload Security and Identification flows MUST be implemented if needed. The High-Security Enhancement makes their implementation mandatory, further restricting the algorithms that can be used, and forbids the use of flows that are considered less safe for use from a security perspective.

8.2 OpenID Connect Flows

Given that they provide a great enhancement in overall interoperability with minimal risk, the API Core Profile allows OpenID Connect flows that are generally considered less safe.

This enhancement requires that the Hybrid Flow MUST NOT be used. Thus the section on Profiled OpenID Connect Flows MUST NOT be taken into consideration for the High-Security Enhancement.

8.3 Security

8.3.1 Transport Level Security

This enhancement mandates the use of Transport Layer Security 1.3 [RFC8446]. Thus, SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1 and TLS 1.2 MUST NOT be used. The following algorithms MUST be used with TLS 1.3:

Key exchange	Certificate Verification	Bulk Encryption	Hashing
ECDHE	ECDSA	AES_256_GCM	(HMAC-)SHA-384

8.3.2 Message Level Security & Payload Security

This enhancement enforces the use of both Message-Level Security and Payload Security. The enhancement also modifies the list of allowed algorithms:

- The ECDSA Algorithm with SHA-384 and P-384 Curve MUST be supported, with a key length of at least 256 bits. The value "ES384" for the "alg" parameter MUST be used in this case as defined in [RFC7518].
- The EdDSA Algorithm [RFC8032] using one of the curves defined in [RFC7748] SHOULD be supported and is RECOMMENDED for use, with a key length of at least 256 bits. The value "EdDSA" for the "alg" parameter MUST be used in this case and the curve shall be encoded in the "crv" parameter as defined in [RFC8037].

Any other algorithms MUST NOT be used.

9 DISCOVERABILITY ENHANCEMENT

9.1 INTRODUCTION

The API Core Profile defines a minimal set of guidelines on how APIs should support discoverability. The Discoverability Enhancement extends this set to mandate further discoverability guidelines. Thus, an API conforming to the Discoverability Enhancement MUST implement both the guidelines defined in the Discoverability section of the API Core Profile, by providing all the OpenAPI attributes prescribed therein, and the additional ones defined in the Discoverability Enhancement.

9.2 DISCOVERABILITY ENHANCEMENT

The discoverability enhancement mandates the use of the following additional attributes that MUST be present in the OpenAPI Document:

- The description, using `info.description` property;
- The documentation URL using the `externalDocs` properties;
- Information on the `servers` property, pointing to all the known deployed instances of the API as shown in the servers object example in the [OAS-V3]. It is RECOMMENDED that a staging URL is equally provided for users to be able to test the API.

The following example provides a summary of the fields required for enhanced API discoverability:

```
info:
  title: The API
  description: This API provides access to citizen resources
  x-edelivery:
    publisher:
      name: The API Publishing Organization
      url: http://www.organization.org/
    lifecycle:
      maturity: deprecated
      deprecated_at: 2020-12-31
      sunset_at: 2021-12-31

servers:
- url: https://api.the-server.com/v2
  description: Basepath serving version 2.x.y of the API

externalDocs:
  description: "API User guide"
  url: https://example.com/guide.pdf
```

REFERENCES

[BCP14] Key words for use in RFCs to Indicate Requirement Levels *and* Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. https://www.rfc-editor.org/info/bcp14

[DRAFT-IETF-HTTP-DGST-HDR] Digest Fields. (work in progress) https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-digest-headers

[DRAFT-IETF-HTTPAPI-DPRC-HDR] The Deprecation HTTP Header Field. (work in progress) https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-deprecation-header

[DRAFT-IETF-HTTPSBIS-MSG-SIGS] HTTP Message Signatures. (work in progress) https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-message-signatures

[DRAFT-IETF-OAUTH-JWT-INTROSPECTION] JWT Response for OAuth Token Introspection. (work in progress) https://datatracker.ietf.org/doc/html/draft-ietf-oauth-jwt-introspection-response

[EDELIVERY-AS4-PROFILE] eDelivery AS4 Profile Version 1.15 https://ec.europa.eu/digital-building-blocks/wikis/x/RqbXGw

[ENISA-CRYPTO-2020] ENISA Good Practises in Cryptography – Primitives and Schemes, December 2020. *(Limited availability)*

[ETSI-JADES] ETSI TS 119 182-1 V1.1.1 (2021-03) Electronic Signatures and Infrastructures (ESI); JAdES digital signatures built on JSON Web Signatures; Part 1: Building blocks and JAdES baseline signatures. https://www.etsi.org/deliver/etsi_ts/119100_119199/11918201/01.01.01_60/ts_11918201v010101p.pdf

[FIELDING-2000] Fielding, R. T., "Architectural Styles and the Design of Network-based Software Architectures", 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[HTTP-STATUS-CODES-REG] Hypertext Transfer Protocol (HTTP) Status Code Registry. https://www.iana.org/assignments/http-status-codes

[ISA-CORE-VOC]. https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/e-government-core-vocabularies

[OAS-V3] OpenAPI Specification version 3.1.0 https://spec.openapis.org/oas/v3.1.0

[OIDC-CORE] OpenID Connect 1.0 Core Specification. https://openid.net/specs/openid-connect-core-1_0.html

[OIDC-DRAFT-ID-ASSUR] OpenID Connect for Identity Assurance 1.0. https://openid.net/specs/openid-connect-4-identity-assurance-1_0.html

[RFC2046] Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. https://datatracker.ietf.org/doc/html/rfc2046

[RFC2119] Key words for use in RFCs to Indicate Requirement Levels. https://datatracker.ietf.org/doc/html/rfc2119

[RFC2234] Augmented BNF for Syntax Specifications: ABNF. https://datatracker.ietf.org/doc/html/rfc2234

[RFC2387] The MIME Multipart/Related Content-type. https://datatracker.ietf.org/doc/html/rfc2387

[RFC3230] Instance Digests in HTTP. https://datatracker.ietf.org/doc/html/rfc3230

[RFC3339] Date and Time on the Internet: Timestamps. https://datatracker.ietf.org/doc/html/rfc3339

[RFC3986] Uniform Resource Identifier (URI): Generic Syntax. https://datatracker.ietf.org/doc/html/rfc3986

[RFC5246] The Transport Layer Security (TLS) Protocol Version 1.2. https://datatracker.ietf.org/doc/html/rfc5246

[RFC5789] PATCH Method for HTTP. https://datatracker.ietf.org/doc/html/rfc5789

[RFC5988] Web Linking. https://datatracker.ietf.org/doc/html/rfc5988

[RFC6749] The OAuth 2.0 Authorization Framework. https://datatracker.ietf.org/doc/html/rfc6749

[RFC7240] Prefer Header for HTTP. https://datatracker.ietf.org/doc/html/rfc7240

[RFC7231] Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://datatracker.ietf.org/doc/html/rfc7231

[RFC7515] JSON Web Signature (JWS). https://datatracker.ietf.org/doc/html/rfc7515

[RFC7518] JSON Web Algorithms (JWA). https://datatracker.ietf.org/doc/html/rfc7518

[RFC7519] JSON Web Token (JWT). https://datatracker.ietf.org/doc/html/rfc7519

[RFC7521] Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants. https://datatracker.ietf.org/doc/html/rfc7521

[RFC7522] Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants. https://datatracker.ietf.org/doc/html/rfc7522

[RFC7523] JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. https://datatracker.ietf.org/doc/html/rfc7523

[RFC7636] Proof Key for Code Exchange by OAuth Public Clients. https://datatracker.ietf.org/doc/html/rfc7636

[RFC7662] OAuth 2.0 Token Introspection. https://datatracker.ietf.org/doc/html/rfc7662

[RFC7748] Elliptic Curves for Security. https://datatracker.ietf.org/doc/html/rfc7748

[RFC7807] Problem Details for HTTP APIS. https://datatracker.ietf.org/doc/html/rfc7807

[RFC8032] Edwards-Curve Digital Signature Algorithm (EdDSA). https://datatracker.ietf.org/doc/html/rfc8032

[RFC8037] CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE). https://datatracker.ietf.org/doc/html/rfc8037

[RFC8174] Key words for use in RFCs to Indicate Requirement Levels. https://datatracker.ietf.org/doc/html/rfc8174

[RFC8446] The Transport Layer Security (TLS) Protocol Version 1.3. https://datatracker.ietf.org/doc/html/rfc6749

[RFC8594] The Sunset HTTP Header Field. https://datatracker.ietf.org/doc/html/rfc8594

[RFC9068] JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens. https://datatracker.ietf.org/doc/html/rfc9068

[SEMVER] Semantic Versioning 2.0.0. https://semver.org/spec/v2.0.0.html

CHANGELOG

9.3 VERSION 1.0

- Global changes:
    - Made editorial changes to improve content clarity
- Changes to API Core Profile section:
    - Removed non-normative material in several sections
    - Introduced 'Client Authentication Framework Profile' for machine clients
    - Specified further mandatory elements to include in the signature when Message-Level Security is used
    - Replaced `info.x-edel-lifecycle` with `info.x-edelivery.lifecycle`
    - Replaced `info.x-edel-publisher` with `info.x-edelivery.publisher`
    - Extended approach to pagination with cursor-based pagination and the recommended use of the Link header
- Changes to API Documentation section:
    - Mandatory documentation endpoint `/openapi` replaced with `/openapi.yaml` or `/openapi.json`
    - Removed requirement to use "DeprecatedOperation" tag
    - JSON schema names switched to CamelCase
    - JwsCompactDetached JSON schema definition is more restrictive
- Changes to Messaging API Specification section:
    - Message Exchange Patterns functional roles renamed from Original Sender/Final Recipient to Submitter/Receiver or Requester/Responder
    - Two new headers introduced:
        - Original-Sender-Token to carry ID token instead of Original-Sender
        - Message-Id to carry the message id in a response message or in a signal message
    - Due diligence followed for the HTTP fields introduced by the specification
    - Clarifications introduced with regard to Timestamp precision
    - New predefined signal Server Errors
    - Message-Id and Timestamp HTTP headers now mandatory for most HTTP responses
    - Get Response Message Reference List endpoints now allow the Edel-Message-Sig header to be present in the HTTP Response

9.4 Version 0.3 - Release Candidate

- Changes to REST API Core Profile:
    - Made editorial changes to improve content clarity
    - Introduced profiling of JAdES-based JSON Web Signatures for Message-Level Security and Payload Security
    - Renamed prefix of custom HTTP Headers from "x-" to "edel-" naming scheme
    - Added new subsections on REST API Design and Single and Multipart Resource Representations in the section Common Semantics
- Added initial version of the API Documentation, using OpenAPI 3.1
- Added initial version of the Messaging API Specification

9.5 Version 0.2 - Partial Draft

- The initial release of the ISA^2 IPS REST API Profile. Contains the first iteration of the REST API Core Profile with the High Security and Discoverability Enhancements