Nictiz

API Requirements

for Dutch Healthcare

Version 1.0.0

July 2022

Versions

#	Description	Status	Publication date
V0.13	First concept published on Mural	Draft	22-03-2022
V0.14	Added requirement SD010	Draft	28-03-2022
V0.15	Added paragraph 3.3 (SDK) and new requirements for documentation, discoverability, onboarding, design rules and API security	Draft	04-04-2022
V0.16	Updated requirements OB004, OB005 and TS001. Added many requirements for Lifecycle Management. Changed Semantically Standardized (SSA) to Fully Standardized (FSA). Added FHIR (Fast Healthcare Interoperability Resources) and IHE (Integrating the Healthcare Enterprise) abbreviations. Added paragraph 1.1 'Intended audience'. Added paragraph 3.2 'Specification, implementation and deployment'. Added paragraph 3.3 'Who uses an API'. Added paragraph 3.4 'What are requirements. Added paragraph 3.5 'API lifecycle'. Added paragraph 3.8 'A layered API typology'. Added paragraph 3.9 'Exchange paradigms'. Updated paragraph 3.16.2 and 3.16.3 to describe levels of standardization in more detail. Added paragraph 3.14 on the contents of API specifications and paragraph 3.15 on the contents of API documentation. Renamed RESTful API design rules to 'API design rules' and created subcategories for generic rules, SOAP based rules and RESTful rules	Draft	04-05-2022
V0.17	Added requirement tables for category 'Security'. Added substantiation for the HTTP/ WEB API restriction in paragraph 3.7 (API protocols and API styles). Adjusted paragraph 3.8 (Al layered topology of APIs) so that experience APIs can be based on/ built on System APIs as well as Process APIs. Added AG001 to AG004 requirements in the Agreements category. Added LM007 and LM008 requirements to API lifecycle management & versioning. Added paragraph 2.3 (Versioning of this specification). Renamed 'HCIM compliance' to 'Health Information Standards compliance' (paragraph 3.17) and added chapter 12 on Health Information Standards compliance	Draft	23-05-2022

V0.18	Removed ‘Changes to the length of data returned within a field’ from paragraph 3.7.3 (non-breaking changes). Added sub requirements for English translations to SD009. Added ‘intentions’ to all requirements in the Agreements category. Removed ‘measurable’ from AG002. Clarified relationship with GDPR in AG003. Added paragraph 3.5 (Relationship with (Dutch) Health Information Standards). Added requirements table to paragraph ‘API requirements categories’. Added two requirements to the ‘Health Information Standards compliance’ category	Draft	13-06-2022
V0.19	Updated paragraph ‘Fully Standardized API standardization level’. Added requirements IS001-IS003 to paragraph ‘Health Information Standards compliance’. Added abbreviations to the table in paragraph ‘Abbreviations used in this specification’	Draft	28-06-2022
V0.2	Updated requirements in the ‘Health Information Standards compliance’ category in accordance with conclusions from working group meeting 04-07-2022. Many textual adjustments. Added paragraph ‘Relationship with Dutch API Library for Healthcare’. Added requirement SD014 to Documentation & Specification	Draft	04-07-2022
V1.0.0	Added call to avoid breaking changes to paragraph ‘Breaking changes’. Various textual improvements.	Final	11-07-2022

Index

1 Introduction

This specification sets out requirements for Application Programming Interfaces (APIs) in Dutch Healthcare. The importance of complying with a common set of requirements and hence the value of this specification is to:

- Promote transparency by setting clear requirements for documentation, testability, discoverability and API onboarding procedures and agreements
- Harmonise API requirements and design between different (national) programs for Health Information Exchange and patient access to health information
- Guarantee a certain level of quality for all APIs included in the Dutch API library for healthcare
- Promote innovation through the availability of system data and by exposing application functionality.

APIs that conform to the requirements in this specification are permitted for inclusion in the Dutch API library for healthcare. APIs that are included in the Dutch API library for healthcare are more discoverable and are likely to meet the requirements in this specification. Therefore, they meet a certain level of quality.

This specification has been developed as part of the Nictiz API strategy.

1.1. | Intended audience

The intended audience for this specification is:

- Technical users and developers
- Policy makers

Notational Conventions

2

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

2.1. | Abbreviations used in this specification

The following abbreviations are used in this specification:

API	Application Programming Interface: used to allow two or more applications to 'talk to each other'
BSON	Binary JSON: an optimized version of JSON
CDA	Clinical Document Architecture: an HL7 standard
CORBA	Common Object Request Broker Architecture: a standard defined by the Object Management Group (OMG) that enables software components to communicate over a network
DCOM	Distributed Component Object Model: a Microsoft technology that allows Microsoft 'COM components' to communicate over a network
DEFLATE	A standard for data compression
EHR	Electronic Health Record
FHIR	Fast Healthcare Interoperability Resources: an HL7 standard
FSA	‘Fully standardized API' level of standardization
GDPR	General Data Protection Regulation: European privacy law
GraphQL	A query language for APIs
gRPC	A remote procedure call framework that leverages the HTTP/2 protocol
GZIP	A standard for data compression
HIE	Health Information Exchange
HTTP	Hyper Text Transfer Protocol: the protocol that fuels the internet
IETF	Internet Engineering Taskforce: the organization that creates standards for the internet
IHE	Integrating the Healthcare Enterprise: a worldwide organization for improving system interoperability in healthcare
JSON	JavaScript Serialized Object Notation: a standard for formatting data
LZ4	A standard for data compression
MedMij	Dutch standard for exchanging health data between Personal Health Record systems and healthcare provider systems

NEN	Royal Netherlands Standardization Institute
NHS	National Health Service: government funded national health services in the UK
NUTS	A Dutch foundation that develops technical standards for Health Information Exchange
OA	‘Open API' level of standardization
OASIS	Organization for the Advancement of Structured Information Standards: an international consortium that promotes the adoption of open standards in computing
oAuth	A standard/ framework for authorization
OIDC	OpenID Connect: a standard for authentication
PACS	Picture Archiving and Communication System
RFC	Request For Comments; a purely technical document published by the Internet Engineering Taskforce (IETF) or other standardization organizations. An RFC can be informational, or it can be a standard
SDK	Software Development Kit
SemVer	Semantic Versioning; a specification for versioning
SOAP	Simple Object Access Protocol: a remote procedure call framework on top of HTTP
TLS	Transport Layer Security: a standard used to secure electronic communications
TSA	‘Technically standardized API' level of standardization
TSV	Taskforce Samen Vooruit; a collaboration of vendors that develops and promotes Health Information Exchange standards for Dutch healthcare
URI	Uniform Resource Identifier; a unique sequence of characters that identifies a logical or physical resource used by web technologies.
Wabvpz	Wet Aanvullende Bepalingen Verwerking Persoonsgegevens in de Zorg: a Dutch law that contains additional privacy regulations for the electronic exchange of healthcare data
WGBO	Wet op de Geneeskundige Behandel Overeenkomst: a Dutch law that regulates the relationship between patients and care providers
WS-security	A set of standards for web service security
XDS	Cross Enterprise Document Exchange: an IHE integration profile
ZIB	Zorg Informatie Bouwsteen: the Dutch version of Health and Care Information Models (HCIM)

2.2. | Requirement identification

Requirements have unique and permanent numbers. In the event of requirements being deprecated or restructured, they are removed from the list. Therefore, gaps in the sequence can occur. New requirements will always get a new and higher number. When the new requirement supersedes an existing requirement, this will be referenced in the new requirement by adding a header “Supersedes:”.

2.3. | Versioning of this specification

This specification follows the Semantic Version 2.0.0 (SemVer) specification for versioning. This means that:

- This specification has a version consisting of three parts: MAJOR.MINOR.PATCH
- The MAJOR number is at least incremented each time requirements are added or changed in such a way that API specifications, API implementations or API deployments that complied with the previous MAJOR version of this specification, don’t comply with the latest version
- The MINOR number is at least incremented each time requirements are added or changed in such a way that API specifications, API implementations and API deployments that complied with the previous MINOR version of this specification, still comply with the new version
- The PATCH number is incremented each time textual changes occur that have no influence on the actual meaning of the requirements in this specification.

Definitions and scope

3

3.1. | What is an API?

The term 'API' refers to both a formal specification and a piece of software conforming to that specification. Both perspectives are used interchangeably, but although an API specification can exist without its actual implementation in software, the reverse is not possible. On the other hand, a specification without any implementation in actual software does not (yet) represent real value. A third possible perspective is the deployment perspective. In this case an API can be viewed as a service that is deployed by some organization to provide specific value for its clients.

No matter the perspective, the purpose of an API is to allow two or more applications to 'talk to each other'. These applications can be located on the same 'machine', or they can be on different remote machines, connected through some kind of network using some kind of communication technology.

3.2. | Specifications, implementations, and deployments

Due to the different perspectives on what an API is, this specification differentiates between API specifications, API implementations and API deployments.

An API specification is a formal specification of the API. The specification can be used to construct an API implementation (software) that complies with the specification. It can also be used to develop an API client (more software) that uses the implementation. The API specification can be viewed as a contract between the API implementation and an API client.

An API implementation is the software code that implements the specification. It is (part of) a specific software product created by a specific software developer (organization).

An API deployment is an instance of the software; the actual service deployed by some organization to provide value for its clients. A specific deployment has a specific endpoint-address (a URL in most modern APIs) that API clients use to access the API implementation.

3.3. | Who uses an API?

API specifications are used by a 'competent developer' to create an API implementation or an API client. A competent developer is a software developer that has experience using the technologies and (healthcare-specific) standards the API exploits and has basic knowledge of the healthcare domain in general and the specific value the API provides.

API implementations are used by API clients. Both are pieces of software that need to 'talk to each other'.

API deployments are used by API client deployments and form a 'service provider'/ 'service consumer' pair.

3.4. | What are API requirements?

API requirements are requirements for specifying (API specifications), implementing (API implementations), deploying (API deployments) and using (API clients) an API. In this specification the following conventions are applied when formulating requirements:

- An API requirement SHOULD not overlap with existing rules and regulations. Compliance with rules and regulations is assumed.
- Even more important, it MUST NOT contradict existing rules and regulations.
- It SHOULD be reasonably possible to apply to API requirements. A requirement should not require the impossible.

Many requirements have sub-requirements that conform to the same conventions.

Whenever possible, requirements in this specification are based on or refer to (inter)national requirements and policies, such as the Dutch National API strategy and national API design rules, international standards and RFCs and policies like the NHS Open API Architecture Policy[1].

3.5. | Relationship with (Dutch) Health Information Standards

According to the Dutch competence centre for electronic exchange of health and care information (Nictiz), a Health Information Standard is a cohesive specification of[2]:

- A use case or a combination of use cases and interaction patterns
- Dataset(s) used within these interactions
- Information models such as HCIMs (Dutch 'Zorg Informatie Bouwstenen' or ZIBs)
- Terminologies
- Communication standards such as HL7 CDA and HL7 FHIR (profiles)

As such, a Health Information Standard does not include technical details that are necessary ingredients of an API specification, such as:

- API signature and semantics
- Identification and authentication of entities
- Security and transport mechanisms such as addressing
- Other required technical parts of an API specification and API documentation as described in paragraphs 3.15 and 3.16

However, many APIs implement (parts of) Health Information Standards. An API is said to 'implement a Health Information Standard' when its specification, implementation and deployment comply with the requirements of that Health Information Standard.

1| See: https://www.england.nhs.uk/wp-content/uploads/2018/09/open-api-policy.pdf

2| See: https://www.nictiz.nl/wp-content/uploads/2021-Paper-Informatiestandaarden-Nictiz.pdf

3.6. | Relationship with the Dutch API library for healthcare

The Dutch API library for healthcare contains API specifications, implementations and deployments that meet the requirements in this specification at a particular level of standardization (see 3.18 for information on API levels of standardization). The API library promotes the findability of APIs that meet the requirements in this specification.

3.7. | API lifecycle

The API lifecycle consists of four phases:

- Create: developing the API specification and implementation
- Deploy: Deploying the API so it can be used by API clients
- Deprecate: Mark the API specification, implementation, or deployment as being removed at a future date
- Retire: remove the API specification, implementation and/or deployment

When developing (Create) an API, especially an API specification targeted at the highest level of standardization (see 'API levels of standardization'), it is important to include the viewpoints and insights of all stakeholders. API development is therefore often considered to be a community effort.

It is important to mitigate the effects of API changes on API clients. Mitigation includes clear communication when deprecating and retiring an API, but also includes preventing client applications from breaking due to changes to a deployed API.

This specification includes requirements for all phases of the API lifecycle and includes requirements for lifecycle management and API versioning.

3.7.1. | API versioning

Updating an API's version is an important measure to help API clients adapt to API changes. Most APIs use the Semantic Versioning (SemVer) scheme[3] and update the API's major version when introducing breaking changes. Non-breaking changes to the API are often reflected by an update in the API's minor version or patch version.

3.7.2. | Breaking changes

A breaking change to an API is any change that can break a client application. Usually, breaking changes involve modifying or deleting existing parts of an API or adding new required parts.
A breaking change can take place in an API specification, API implementation and in an API deployment.

3| See https://semver.org/

Examples of breaking changes are:

- Deleting a resource or operation
- Removing an allowed parameter, request field or response field
- Modifying a resource or method URI
- Modifying a field name
- Adding *required* query parameters without default values
- Introducing a new validation
- Modifying authorization
- Modifying rate-limiting
- Removing TLS (Transport Layer Security) versions or supported encryption methods
- Modifying data formats, encodings, or compression formats

If the intended functionality of an API (method) changes, this might also break the client application and therefore it is considered a breaking change. For example, if a DELETE request previously used to archive a resource but now hard deletes the resource, the change potentially breaks client functionality that is supposed to archive a resource.

Breaking changes should be avoided whenever possible.

3.7.3. | Non-breaking changes

Non-breaking changes are changes that cannot be expected to break a client application. Adding optional parts to an API is never considered a breaking change. Examples of non-breaking changes are:

- Addition of new endpoints
- Addition of new resources or operations to existing endpoints
- Addition of new fields in the following scenarios:
  - New fields in responses
  - New optional request fields or parameters
  - New required request fields that have default values
- Addition of optional query parameters
- Changes to the order of fields returned within a response
- Addition of an optional request header
- Removal of redundant request header
- Changes to the overall response size
- Changes to error messages.

Requirement LM007 states that API clients MUST be designed to handle non-breaking changes.

3.8. | Using a Software Development Kit (SDK) for easy access to APIs

While APIs can be consumed using any platform or programming language, Software Development Kits make it easier to access APIs from a particular platform and/or language of choice. An SDK is usually made up of one or more software libraries, tools, and documentation. For example, Microsoft provides SDKs for accessing Azure APIs from several different platforms, including iOS, Android, JavaScript, and .NET. Another example is the Facebook SDK for integrating Facebook features in games built on the Unity platform.

This specification does not require the provision or use of an SDK to access an API. Nevertheless, this specification does include requirements in case an SDK is provided.

3.9. | API protocols and API styles

This document specifies requirements for APIs that are built on top of the HTTP protocol. HTTP, or Hyper Text Transfer Protocol, is the protocol that fuels the World Wide Web. It is a high-level protocol that provides mechanisms for communication between clients and servers with requests and responses. The best-known HTTP client is a web browser that is used to communicate with web servers, requesting data and web pages from web servers or sending user input to web servers.

The reason for restricting this specification to HTTP-based APIs is threefold:

1. The world has been transitioning to the use of HTTP-based APIs ever since the web 2.0 era at the beginning of the 21st century. Therefore most modern API technologies that are based on the HTTP protocol and legacy technologies, such as CORBA and DCOM, are slowly but surely disappearing from the global arena

2. As a result of this, many (if not most) standards for secure communication are based on or are working in accordance with the HTTP protocol. Examples are the numerous RFCs for authorisation (like oAuth2.0) and authentication (like OpenID Connect), as well as stacks of security protocols such as WS-Security. Other examples include HTTP-header based content negotiation schemes and protocols for distributed identity management such as the Verifiable Credentials HTTP API.

3. Modern development tools (as well as developer training materials) are optimized for building applications that leverage the HTPP protocol and the various design patterns and protocols that work with HTTP or are built on top of HTTP.

APIs that use the HTTP protocol are often referred to as WEB APIs. WEB APIs are used to communicate between applications over the internet but can also be used for communication over private networks or even between applications on the same machine. WEB APIs are often divided into two groups: SOAP-based APIs and RESTful APIs. SOAP is a protocol on top of HTTP

(amongst others) while REST is an architectural style that leverages plain HTTP commands (verbs) for communication between the API client and the API server. SOAP is often associated with the WEB 1.0 era while today's modern WEB APIs are edging towards the REST architectural style. Nevertheless, both SOAP-based APIs and RESTful APIs coexist and many existing and successful initiatives for health information exchange are based on SOAP APIs. Newer WEB API technologies and styles, like GraphQL and gRPC, are rising constantly.

This specification strives to provide requirements that apply to all flavours of WEB APIs. However, its focus will be on REST APIs simply because of its growing popularity in Health IT. For example, the popular HL7 FHIR API specification is based on the REST architectural style.

3.10. | A layered typology of APIs

It is common to differentiate between three categories of APIs: system APIs, process APIs and convenience or experience APIs[4]. This differentiation is based on design principles like 'separation of concerns' and the need for reusability of specifications and software components such as APIs.

System APIs are atomic APIs. They expose 'raw' access to data and functionality from a system of record. They cannot be divided into smaller parts without losing usefulness and meaningfulness: you cannot create a meaningful API that exposes the first half of a medical diagnosis or one that creates an outpatient appointment for a particular date without specifying the time. Often, System APIs are exposed for internal use only (see 'Internal and external API usage'). API specifications for System APIs should not be specific to a particular business purpose or 'use case'. Neither should they be specific to a particular type of user or a particular type of system.

In healthcare, system APIs provide access to specific data and functionalities in systems like electronic health records, medical imaging systems, Hospital Information Systems, and many others. Many FHIR resources can be used to standardize system APIs. FHIR resource specifications do not define authorization/access control or the authentication of users and clients, which allows them to be used in many different use cases for several types of users.

Process APIs provide a means of combining data and orchestrating multiple system APIs for a specific business purpose. Note that process APIs can combine system APIs from a single backend system but also from a variety of backend systems.

In healthcare, the difference between system APIs and process APIs becomes clear when you consider that many healthcare business processes require the combining of data and/or functionality from different systems. For example, transferring a patient from a hospital to a homecare provider often requires combining data from different systems, such as an EHR (Electronic Health Record) system and a PACS system amongst others. A process API that supports the transfer process (for example based on the Dutch eOverdracht specification) uses the system APIs of these 'systems of record' to expose a single coherent dataset to the API client.

4| The Dutch API strategy (https://docs.geostandaarden.nl/api/API-Strategie/) uses this differentiation which was originally proposed by MuleSoft (https://www.mulesoft.com/resources/api/types-of-apis)

The latter being a system deployed by (or on behalf of) the homecare provider. Process API specifications for healthcare often define specific methods for authenticating clients/users and define specific authorization/access control methods.

The third category of APIs, convenience APIs or experience APIs, are tailored to a specific kind of usage. For example, an experience API could be tailored to mobile devices and provide mobile-friendly ways of formatting and paginating data. Experience APIs use System APIs or process APIs but format the output in such a way that it caters to the needs of a specific (type of) API client.

In healthcare, experience APIs are often representations of business data and/or functionality to a specific Healthcare Information Exchange (HIE) standard, such as a FHIR, or IHE XDS. For example, the previous 'eOverdracht' transfer example could be expressed using two experience APIs: one based on FHIR notifications and compositions and the other on IHE DSUB notifications and XDS document exchange.

This specification covers all types of APIs and sets out requirements for reusing APIs of a lower layer.

3.10.1. Consequences for API design and specification

The layered approach to APIs has consequences for API design and specification in healthcare. Specifications of APIs that support transferring patients (process APIs), should reuse system-level API specifications that provide access to specific data. API standards for transferring patients should reuse pre-existing API standards for accessing atomic data, such as atomic FHIR resources.

API specifications for a specific (type of) client system, such as a FHIR client, should be harmonized with existing specifications for process APIs.

3.11. | Exchange patterns

APIs can be used to PUSH data from a client to a server or to PULL data from a server by a client. PUSH and PULL are examples of 'exchange patterns'. Other examples are the NOTIFY-PULL pattern (server notifies the client using an API exposed by the client after which the client pulls data from the server using an API exposed by the server) and the BROADCAST pattern (server broadcasts data to all clients that are interested in receiving that data).

This specification strives to provide requirements that are applicable to all possible exchange patterns.

3.12. | Exchange paradigms

It is common to differentiate between four different paradigms for exchanging data between systems: operations, messaging, exchanging documents and exposing resources[5].

The operations paradigm, or Remote Procedure Call (RPC) paradigm, allows a client to execute some code on a server by passing it the operation name and required (and optional) arguments. After processing the code, the server returns some content to the client. The content can be anything, from a stream of data to a static document (see document paradigm) or a single scalar value. The SOAP protocol is designed around the operations paradigm.

Messaging is associated with the 'PUSH' exchange pattern and facilitates automated transactions between systems. System A sends a message to System B with a specific intention and just enough information to justify that intention. Interaction typically occurs without human interaction: a message invokes a state change (such as a workflow state change or changing specific data) within a target system. Typical examples in healthcare are HL7v2 messages and FHIR messages.

Document exchange is of value in healthcare IT. A document is created by an author and represents a snapshot of available information at a specific time and place: documents are 'stable'. A document can be (digitally) signed by a human, stating that the document is approved by its 'verifier'. A document is first and foremost intended for human consumption (even if it's exchanged by electronic means), but it can be processed by automated systems and automated decisions can be based on its contents, especially if the document has a standardized and machine-readable (structured) format. Note that the document paradigm can be combined with the operation, messaging and resource paradigms, because documents can be the result of an operation, can be transmitted through a message and can be requested from a 'documents' resource.

In (Dutch) healthcare, documents have significant value in exchanging data between healthcare organizations. A document can be used to send data between healthcare organizations in accordance with the Dutch WGBO regulation: a healthcare professional transmits specific data (the document) to a specific healthcare provider with a specific purpose (e.g., a transfer or consultation) in accordance with healthcare quality standards or best practices. In such cases data can be transmitted without the specific consent of the patient.

As opposed to documents, resources are dynamic. A resource exposes data in response to a specific request and its content changes over time and is dependent on the specifics of the request. Resources are often associated with the PULL and NOTIFY PULL exchange patterns, because a dynamic resource can't be PUSHED to another system. In healthcare, FHIR resources are the most popular example of the resource paradigm. Because of their dynamic nature, exchange of resources across organizations often requires specific consent of the patients concerned.

This specification covers all four paradigms and sets out requirements for when to use which paradigm.

3.12.1. | Consequences for API design and specification

Different paradigms are suitable for different scenarios. API design and specification should consider that APIs designed using the resource paradigm make the API client responsible for defining the content of the data exposed by the API. The resource paradigm is very flexible from the perspective of the consumer (the API client) but consequently, the healthcare organization responsible for the data has less control over what data is exchanged in what situation.

3.13. | Internal and external API usage

Many policies, like the Dutch API strategy and the NHS Open API policy, distinguish between internal and external APIs. Indeed, European and Dutch regulations do impose such a distinction but distinguishing internal from external APIs might not always be easy and has nothing to do with technology.

Processing personal data is always subject to privacy regulations such as the European GDPR (General Data Protection Regulation). When data is exchanged between organizations, special rules apply. These rules depend on the relationship between those organizations, like the relationship between a (GDPR) ‘data controller’ and ‘data processor’. ‘Internal’ and ‘External’ are not a property of the APIs themselves but indicate different use of (sometimes the same) APIs. Although different intended uses may affect API design, our approach is to treat internal or external API design alike as much as possible.

In this specification ‘internal API usage’ is restricted to data exchange within a single data controller. ‘External API usage’ on the other hand, covers the exchange of data (using APIs) between data controllers (and their respective data processors). External API usage is subject to special regulations such as the Dutch Wabvz and the Dutch NEN7512 standard.

This specification covers both internal and external APIs.

3.14. | Unrestricted and restricted API usage

APIs can be used to provide unrestricted access to data and functionality. These kinds of APIs don’t require authorization to access API functionality and/or data and hence don’t need to know the identity of the person or organization using the API. APIs that provide unrestricted use are sometimes referred to as ‘Anonymous APIs’ or as ‘Open Data APIs’.

Especially in healthcare, most APIs expose (sensitive) personal data and hence do require authorization. These APIs restrict access to data and/or functionality to specific applications, organizations and/or users. Hence, they need to identify and authenticate API users. This is called ‘restricted usage’. APIs that provide ‘restricted usage’ are sometimes referred to as ‘identified APIs’.

In real life, even APIs that provide unrestricted use to end-users during operation, do have some restrictions in place that apply to developers. For many APIs special onboarding procedures are in place. Only after onboarding are developers provided with a so-called API key that gives them

(their software) access to the API. This allows the organization that provides the API to prevent (deliberate or undeliberate) misuse. It also allows for better statistical analysis of API use.

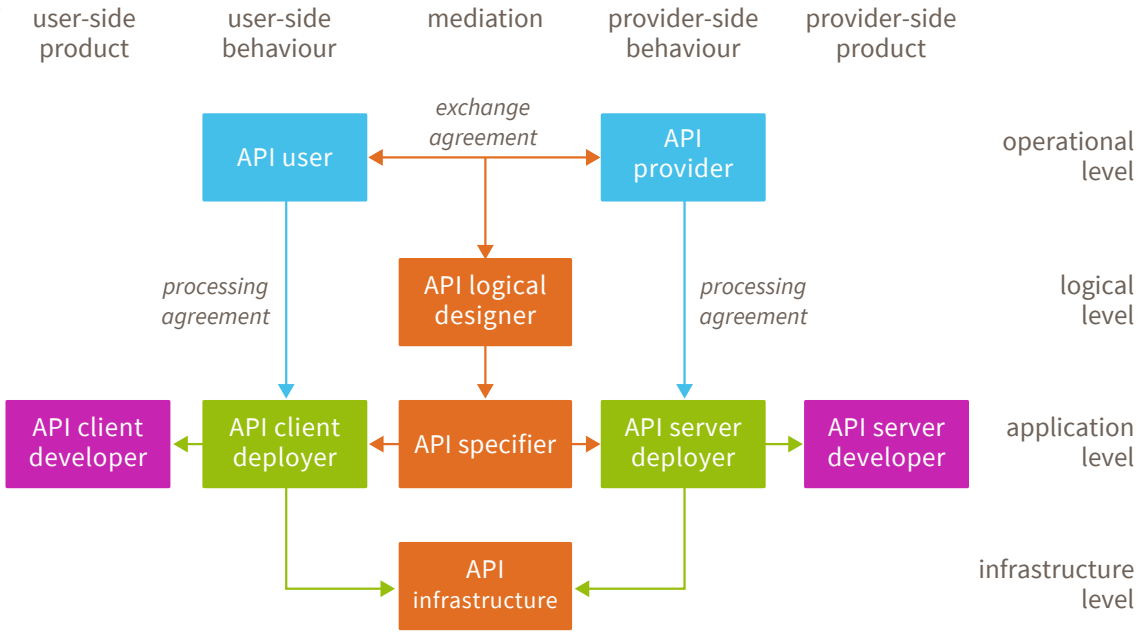
This specification covers both the restricted and unrestricted usage of APIs.

3.15. | Roles involved with the development, exploitation and use of APIs

APIs are created by developers and are consumed by software created by (other) developers. Sometimes the party responsible for creating and maintaining an API is also responsible for deploying the API. At other times, the development and deployment roles are fulfilled by different parties. The same is true for API clients. Sometimes systems that consume an API, so-called API clients, are developed and deployed by one party, sometimes different parties are responsible for developing and deploying an API client system.

Most of the time, the party responsible for developing an API is also responsible for specifying the API. In the case of standard APIs, such as the Dutch MedMij APIs, specifications for APIs are the responsibility of a (National) standards body or 'API specifier', in this case Nictiz. Sometimes, the API specifier is also responsible for verifying API conformance to the specification, sometimes designated 'API conformance verifiers' (such as notified bodies) fulfil such a role.

Because separate roles have different responsibilities throughout the lifecycle of an API, most API requirements are specific to a role. This specification recognizes nine roles involved with the creation, deployment and use of APIs. Diagram 1 sets out these nine roles.



(Diagram 1) Roles involved with specification, creation, deployment and use of APIs

Each role is briefly introduced in Table 1. Requirements in this specification are assigned to one or more roles through the 'applicable roles' attribute of each requirement.

API role	Responsibilities
API client deployer	Technical responsibilities for employing an API, as deployed by the API server deployer, and specified by the API specifier, thus implementing the final responsibilities of the API user
API client developer	Technical responsibilities for supplying software for the API client deployer
API infrastructure	Technical responsibilities for conveying specified APIs between API clients and API servers
API logical designer	Responsibilities for logically specifying both data and the operations to be implemented in the API
API provider	End responsibilities for providing the value and meaning of an API, as agreed with API users
API server deployer	Technical responsibilities for deploying an API, as specified by an API specifier, thus implementing the end responsibilities of the API provider
API server developer	Technical responsibilities for supplying software for the API server deployer
API specifier	Responsibilities for technically specifying the API so that an API server deployer knows what to deploy and an API client knows what to employ
API user	End responsibilities for using the value and meaning of an API, as agreed with the API provider

(Table 1) Roles involved with the specification, creation, deployment and use of APIs

3.16. | The contents of an API specification

An API specification MUST contain enough information for a competent developer to create an API implementation or an API client without further information. This includes:

1. Identification and authentication of people, organizations, and machines
   This does not only apply to the technical standards and specifics used to authenticate entities but also to the identifying attributes that are used and how to obtain and secure them to create a network of trust.

2. Authorization/access control
   This does not only apply to the technical standards and specifics used to authorize access to APIs, but also to the semantics of access tokens and requests for access tokens, such as the permitted values for permissions (oAUth2 scopes) and expiration requirements.

3. Protecting integrity and confidentiality
   This applies to any specifics on protecting integrity and confidentiality at both transport and message levels, including specifics on the cryptographic algorithms, key distribution and PKIs used.

4. Addressing
   This applies to specifics on addressing API endpoints and mechanisms used to distribute (updates to) addresses.

5. Content encoding
   This applies to specifics on content encoding such as the compression algorithms used and character encoding.

6. Content formatting
   Specifics on content formatting such as the use of MTOM/XOP and BSON but also healthcare-specific (data) formats.

7. Exchange patterns and exchange paradigms used

8. API signature and semantics
   All actions (methods) that are available through the API MUST be covered, as well as the legitimate data structures return (error) codes (the API signature), Including a full specification of all API requests and responses.

9. Use cases
   How to (and how not to) use the API in specific use cases.

10. References to other specifications
    Most specifications reuse other specifications such as RFCs created by IETF or W3C or Dutch information standards created by Nictiz.

Creating and maintaining the API specification is the responsibility of the API specifier role.

3.17. | The contents of API documentation

API documentation includes, but is not limited to, the API specification. Other important parts of API documentation include:

- How to obtain and use test tooling
- API onboarding and access policies
- Usage restrictions and guidelines
- Service level agreements
- Technical specifics for a particular deployment, such as the use of private networks instead of public internet
- Addresses of API endpoints

These parts of API documentation are the responsibility of the API server deployer. In many cases, the API server deployer works together with the API server developer or refers to documentation the API server developer supplies.

3.18. | API levels of standardization

Almost all APIs are based on standards such as communication standards (like the HTTP standard) and formatting standards such as XML. Nevertheless, many APIs use different communication technologies, different documentation formats and testing tools, different methods for discoverability, different data formats, unique styles and patterns and different 'content', for what seems to be similar functionality and purpose. These differences complicate the use of APIs.

This specification provides requirements for APIs with various levels of standardization. Even at the lowest level of standardization, the 'Open API level', requirements are set out that harmonize API design, development, deployment, and usage. At the middle level, the 'Technically standardized level', requirements aim at achieving technical harmonization while at the highest level, the 'Fully standardized level', requirements aim at achieving software interoperability.

Requirements in this specification are assigned to one or more levels of standardization through the 'applicable levels' attribute of each requirement. Valid values for this attribute are 'OA' (Open API), 'TSA' (Technically Standardized) and 'FSA' (Fully Standardized).

3.18.1. | 'Open API' standardization level

The NHS uses the following definition of 'Open APIs'[6]:

'Open APIs are those APIs that have been exposed to enable other systems to interact with that system, and those APIs have been sufficiently documented so that the available functionality is discoverable, fit for purpose and re-usable.'

This specification embraces the NHS definition of 'Open APIs'. It is important to disambiguate from the OpenAPI initiative (formerly Swagger) that standardizes how APIs are described.

6| https://www.england.nhs.uk/wp-content/uploads/2018/09/open-api-policy.pdf

At the ‘Open API’ level, the only technical requirements are that:

- APIs are based on the HTTP communication protocol (see API protocols and styles).
- APIs use common and state-of-the-art technologies and standards.
- APIs comply to common security and privacy guidelines and regulations

Other than that, API developers are free to use the technology of their choice and are free to create APIs using their own data formats and ‘content’. Even the purpose of these ‘Open APIs’ is defined by the organization creating them.

Requirements at this level aim to increase reusability, discoverability and quality, without restricting APIs to specific technical and/or semantical standards. This preserves agility and increases the speed at which APIs become available to API users and innovators, while at the same time providing some level of harmonization and transparency. Typical requirements at this level concern (transparency of) documentation, testability and onboarding procedures.

3.18.2. ‘Technically standardized API’ standardization level

Requirements at this level aim to increase the technical harmonization of APIs. Examples of requirements at this level are design rules, requirements for versioning and lifecycle management, security requirements, formatting requirements and transport requirements.

Many of these requirements reflect a technical choice, such as using JSON, BSON or XML. Other examples include the compression methods that are allowed (such as GZIP and DEFLATE or the less common LZ4), what security models and standards are allowed (such as oAuth2 and WS-security) and what cryptographic methods are allowed for assuring confidentiality and integrity.

In Dutch Healthcare, no single party has the authority to enforce these kinds of choices and many technical choices and standards coexist. This specification aims to harmonize technical choices by referring to specific (inter)nationally recognized standards, guidelines and best practices. Sources for these references may include, but are not limited to:

- W3C standards
- Internet Engineering TaskForce (IETF) RFCs
- OASIS standards
- International health-IT standardization efforts and standards such as HL7 and IHE
- The Dutch National Cyber Security Centre (NCSC) security guidelines
- Dutch NEN norms
- Dutch ‘afsprakenstelsels’ such as MedMij
- Technical agreements between health-IT industry partners such as made by the Dutch Taskforce Samen Vooruit (TSV, now part of NLDigital) and the NUTS foundation

For an API to comply with the requirements at this level it MUST also comply with the requirements at the ‘Open API’ standardization level.

3.18.3. | 'Fully standardized API' standardization level

Requirements at this level aim to increase software interoperability by setting requirements for standardizing all parts of APIs.

In Dutch Healthcare, no single party is designated to approve API standards. However, many national and international organizations are concerned with developing API specifications and testing their implementations. Examples of such organizations include, but are not limited to, the Royal Netherlands Standardization Institute 'NEN', the Dutch MedMij foundation, the international HL7, IHE and openEHR foundations and the Dutch NUTS community.

An API is fully standardized when:

- Its complete specification is approved as a standard by a standardization organization
- Its implementation is verified by that organization during a formal test or qualification process.

Because different standardization organizations can create different standards for the same purpose and use case, 'fully standardized' does not equal 'the only allowed way of doing things'. However, all members of the standardization organization approve the API specification and promote its implementation in real life systems. The Dutch API library for healthcare will include 'competing' fully standardized APIs from different standardization organizations, as long as:

- They fulfil the requirements for fully standardized APIs
- Their organizations are supported by a substantial number of Health IT stakeholders, such as Health IT vendors and/or healthcare providers
- Their organizations provide a formal test- or qualification process for API implementations, such as the IHE connectathon and the Nictiz MedMij qualification tests.

In exceptional cases, the Dutch government can force the use of specific standards, such as API standards. The upcoming Wegiz legislation provides the Dutch government with the means to force the use of specific standards for information exchange, including the use of specific fully standardized APIs. Other means to promote the use of specific fully standardized APIs above others are enforcing their use through common purchasing conditions.

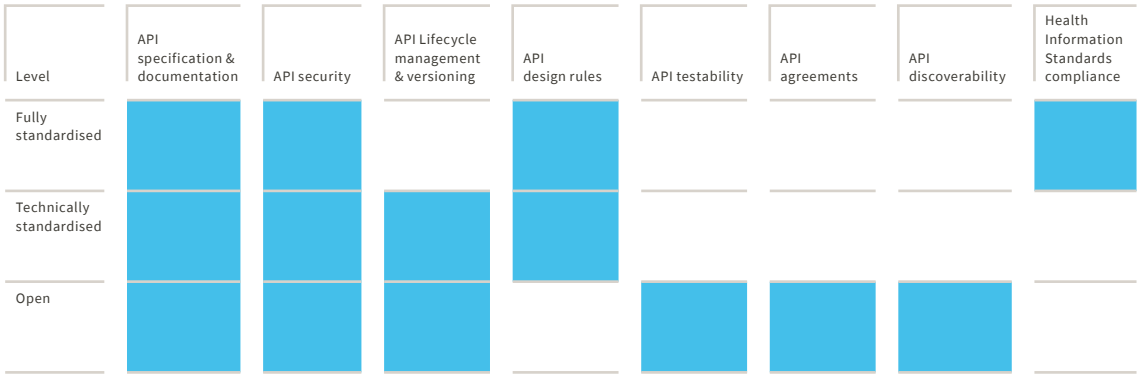
For an API to comply with the requirements at this level it MUST also comply with the requirements at the 'Open API' and 'Technically standardized' standardization levels.

3.19. | API requirement categories

All requirements in this specification fall under a specific category. This specification recognizes nine requirement categories:

- API specification & documentation
- API security
- API lifecycle management & versioning
- API Design Rules
- API testability
- API onboarding
- API agreements
- API discoverability
- Health Information Standards compliance

Some categories contain requirements on a particular level of standardisation (such as the ‘Open API standardization level’), others contain requirements on two or even all standardization levels.



(Table 2) Levels of standardization and requirement categories. Blue cells indicate the availability of requirements for a particular level and category combination

API specification & documentation

4

Code	Requirement	Applicable role(s)	Standardization levels
SD001	*API documentation MUST be publicly and freely available*	API specifier	OA, TSA, FSA
SD002	*API documentation MUST provide examples of how to use the API*	API specifier	OA, TSA, FSA
SD003	*API documentation SHOULD provide examples of input and output data*	API specifier	OA, TSA, FSA
SD004	*API documentation SHOULD include a FAQ page for API client developers*	API specifier	OA, TSA, FSA
SD005	*API documentation MAY specify cases in which API usage is not applicable*	API specifier	OA, TSA, FSA
SD006	*API server developers and/or deployers MAY be active on developer forums to assist API client developers and deployers with the correct usage of APIs*	API server developer, API server deployer	OA, TSA, FSA
SD007	*API server developers MAY provide API client developers with an SDK for easy access to deployed APIs*	API server developer	OA, TSA, FSA
SD008	*API specifications SHOULD be machine-readable and allow for automated code generation*	API server developer	OA, TSA, FSA
SD009	*API documentation MUST be published in English*	API specifier	OA, TSA, FSA
SD010	*Documentation MUST provide (references to) evidence to back any compliance claims made*	API specifier	OA, TSA, FSA
SD011	*Content relationship MUST be described in API documentation*	API specifier	TSA, FSA
SD012	*API documentation MUST describe the availability and usage of operations*	API specifier	TSA, FSA
SD013	*API versioning policy MUST be documented*	API specifier	TSA, FSA
SD014	*API specifications MUST cover the rationale behind the exchange paradigms used by the API*	API specifier	FSA

4.1. | API documentation MUST be publicly and freely available

Requirement code: SD001
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

Each API MUST be documented to the extent that a competent developer has sufficient information to make use of the API without further information. The specification MUST therefore at least cover the elements that are addressed in paragraph 3.13 (The contents of an API specification), or it MUST refer to other specifications that cover these elements.

The documentation MUST be freely available and accessible via a public website. As an exception to this requirement, documentation MAY refer to paid content published by standardization organizations. Free registration to access the API documentation is accepted.

API documentation MAY be copyright protected and further distribution without explicit permission (of the API specifier) MAY be restricted.

4.2. | API documentation MUST provide examples of how to use the API

Requirement code: SD002
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

API documentation MUST include examples for the most common use cases. The documentation MUST clearly express the value of the API (for API client developers and API users) within the context of these use cases.

When a use case involves integration of two or more APIs, the documentation MUST provide examples of how to use these APIs in collaboration.

4.3. | API documentation SHOULD provide examples of input and output data

Requirement code: SD003
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

Documentation for all API methods SHOULD contain examples of input and output data, using a supported data format such as JSON or XML.

When the API server developer includes an SDK for easy access to the API, code samples MUST be provided for using the API through the SDK.

4.4. | API documentation SHOULD include a FAQ page for API client developers

Requirement code: SD004
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

A FAQ SHOULD be made available for developers that want to use the API(s). The FAQ SHOULD be written in an actual question-and-answer format. Questions and answers SHOULD be written from the point of view of the API client developer. Questions SHOULD include the most common problems and misconceptions that API client developers run into when using the API.

4.5. | API documentation MAY specify cases in which API usage is not applicable

Requirement code: SD005
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

Documentation MAY include cases in which API usage is not applicable or not supported. In this case documentation MUST clearly state whether using the API in this way violates the API license agreement.

4.6. | API server developers and deployers MAY be active on developer forums to assist API client developers and deployers with the correct usage of APIs

Requirement code: SD006
Applicable roles: API server developer, API deployer
Standardization levels: OA, TSA, FSA

Using developer forums, experts can provide resolutions to common hiccups and increase participation and interest for API client developers.

Server developers and deployers MAY be (are encouraged to be) active participants on relevant developer forums. Solutions to problems provided through a developer forum MAY be provided without prejudice.

API documentation MAY refer to developer forums for knowledge sharing and assistance.

API server developers and deployers MAY provide their own developer forum.

4.7. | API server developers MAY provide API client developers an SDK for easy access to deployed APIs

Requirement code: SD007
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

If an SDK is provided, it MUST be documented to the extent that a competent developer has sufficient information to make use of the SDK without further information.

Charges for using the SDK are accepted.

If an SDK is provided, API client developers MUST NOT in any way be forced to use it.

Any common coding language and/or development platform is accepted.

4.8. | API specifications SHOULD be machine readable and allow for automated code generation

Requirement code: SD008
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

Using machine-readable API specification allows for automated code generation and hence saves time and avoids errors when writing API client code.

API server developers SHOULD provide machine-readable API specifications based on international standards such as OpenAPI (formerly known as Swagger) and/or FHIR StructureDefinitions/OperationDefinitions.

4.9. | API documentation MUST be published in English

Requirement code: SD009
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

API documentation MUST be available in English.
Typical Dutch terminology and names of people and organizations MUST be written down in their original Dutch form. Domain concepts MUST be translated to their corresponding official English terms instead of using literal (word-for-word) translations.

4.10. | Documentation MUST provide (references to) evidence to back up any compliance claims made

Requirement code: SD010
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

When documentation claims compliance to standards, specifications, guidelines and practices, policies or law, evidence to back up these claims MUST be provided.

Examples of evidence include official compliance certificates and statements (such as IHE integration statements and Nictiz qualifications) and independent auditor reports (such as security audit reports).

4.11. | Content relationship MUST be described in API documentation

Requirement code: SD011
Applicable roles: API specifier
Standardization levels: TSA, FSA

Some content cannot exist without its parent content. The API documentation must describe in which way the relationship is managed and used.

4.12. | API documentation MUST describe the availability and usage of operations

Requirement code: SD012
Applicable roles: API specifier
Standardization levels: TSA, FSA

API operations can be extremely useful in specific use cases, describing the operations will make them even more useful.

4.13. | API versioning policy MUST be documented

Requirement code: SD013
Applicable roles: API specifier
Standardization levels: TSA, FSA

The versioning of APIs and its policy must be documented in a clear manner. Use of versioning makes developing and debugging by API client developers easier.

4.14. | API specifications MUST cover the rationale behind the exchange paradigm used by the API

Requirement code: SD014
Applicable roles: API specifier
Standardization levels: FSA

An API specifier MUST describe the rationale behind the exchange paradigm used by the API, such as described in paragraph 3.12.

API testability

5

Code	Requirement	Applicable roles	Standardization levels
TS001	*Public test tooling MUST be freely available for test purposes*	API deployer	OA, TSA, FSA

5.1. | Public test tooling MUST be freely available for test purposes

Requirement code: TS001
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

An API server deployer MUST provide test tooling which allows API client developers and API client deployers to test their solutions without affecting production environments. This test tooling MAY be supplied and maintained by API server developers.

Test tooling and data exposed by test tooling MUST mimic real API usage. Both online and offline tools are accepted. For online test tooling, the API server deployer MUST specify what are the Service Level Agreements for the availability and response times of the test tooling.

Test tooling MUST NOT expose confidential data, including but not limited to, patient data.

The test tooling MUST be freely available and accessible via a public website. Free registration to access the test tooling is accepted.

The test tooling MAY be subject to specific onboarding procedures and policies if these policies comply with onboarding requirements in this specification.

API discoverability

6

Code	Requirement	Applicable roles	Standardization levels
DI001	*API specifications SHOULD be published in the Dutch API library for healthcare*	API specifier	OA, TSA, FSA
D002	*API implementations SHOULD be published in the Dutch API library for healthcare*	API server developer	OA, TSA, FSA
DI003	*API deployments SHOULD be published in the Dutch API library for healthcare*	API server deployer	OA, TSA, FSA

6.1. | API specifications SHOULD be published in the Dutch API library for healthcare

Requirement code: DI001
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

API specifications SHOULD be published in the Dutch API library for healthcare. For each version of the published API specification, the API specifier MUST specify the status of that version (trial implementation, production, deprecated, retired).

When publishing (a version of) an API specification, the API specifier MUST provide (a link to) all specification, documentation and qualification documents available.

API specifications that are published in the Dutch API library for healthcare MUST at least comply with all requirements for Open APIs (Open API standardization level) that apply to the API specifier role.

6.2. | API implementations SHOULD be published in the Dutch API library for healthcare

Requirement code: DI002
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

API implementations SHOULD be published in the Dutch API library for healthcare. For each version of the published API implementation, the API server developer MUST specify the status of that version (trial implementation, production, deprecated, retired).

When publishing (a version of) an API implementation, the API server developer MUST provide the name and version of the system that contains the API implementation.

API implementations that are published in the Dutch API library for healthcare MUST refer to (one or more versions of) an API specification that is published in the Dutch API library for healthcare.

API implementations that are published in the Dutch API library for healthcare MUST at least comply with all requirements for Open APIs (OA standardization level) that apply to the API server developer role.

6.3. | API deployments SHOULD be published in the Dutch API library for healthcare

Requirement code: DI003
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

API deployments SHOULD be published in the Dutch API library for healthcare. For each version of the published API deployment, the API server deployer MUST specify the status of that version (trial implementation, production, deprecated, retired).

When publishing (a version of) an API deployment, the API server deployer MUST provide (a link to) all information considering agreements and conditions for using the API, onboarding and testing procedures and endpoint information.

API deployments that are published in the Dutch API library for healthcare MUST refer to an API implementation that is published in the Dutch API library for healthcare.

API deployments that are published in the Dutch API library for healthcare MUST at least comply with all requirements for Open APIs (Open API standardization level) that apply to the API server deployer role.

API onboarding

7

Code	Requirement	Applicable roles	Standardization levels
OB001	*All API onboarding policies, criteria and procedures MUST be documented*	API deployer	OA, TSA, FSA
OB002	*API onboarding SHOULD be an online self-service process*	API deployer	OA, TSA, FSA
OB003	*API onboarding MAY require a review of the client system and API client developer organization*	API deployer	OA, TSA, FSA
OB004	*A privacy statement MUST be provided whenever API onboarding requires the API client developer to provide information on the client system and/or client developer organization*	API deployer	OA, TSA, FSA
OB005	*An API offboarding procedure MUST be provided*	API deployer	OA, TSA, FSA
OB006	*All API offboarding policies, criteria and procedures MUST be documented*	API deployer	OA, TSA, FSA

7.1. | All API onboarding policies, criteria and procedures MUST be documented

Requirement code: OB001
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

API documentation MUST include details of the onboarding process, including criteria and policies for onboarding approval and disapproval.

When the onboarding process requires a review of the client software and/or API client developer organization, the documentation MUST include details on the review process.

7.2. | API onboarding SHOULD be an online self-service process

Requirement code: OB002
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

To speed up the onboarding process, API onboarding SHOULD be an online self-service process.

An online service SHOULD be used to submit all the information required to onboard the API client developer and/or client system.

7.3. | API onboarding MAY require a review of the client system and API client developer organization

Requirement code: OB003
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

The API onboarding process MAY include a review of the client system and of the organizations that develop and/or deploy the API client.

An API server deployer MAY require an external review of the API client software, such as a security review, as part of the API onboarding procedure.

An API server deployer MAY require external certification of the Quality Management System (QMS) and/or Information Security Management System (ISMS) used by organizations that develop and/or deploy the API client as part of the API onboarding procedure.

7.4. | An Information Disclosure Statement MUST be provided whenever API-onboarding requires the API client developer to provide information on the client system and/or client developer organization

Requirement code: OB004
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

When an API server deployer requires an organization to submit information as part of the onboarding process, the API server deployer MUST provide an Information Disclosure Statement.

The Information Disclosure Statement MUST explicitly state what information is considered

confidential and what information is NOT considered confidential.

The Information Disclosure Statement MUST explicitly state what information is permanently destroyed, and what information is not, after offboarding.

The Information Disclosure Statement MUST explicitly state to which other parties information is supplied and for what purpose.

The Information Disclosure Statement MUST explicitly state any restrictions on the time period that the statement is considered in effect.

7.5. | An API offboarding procedure MUST be provided

Requirement code: OB005
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

The API server deployer MUST provide a procedure for offboarding an API client and/or organization responsible for developing and/or deploying an API client.

After offboarding, the API server deployer MUST remove any public (published) statements that indicate active onboarding by the API client and/or API client organization.

The API server deployer MAY store information that was previously submitted as part of the API onboarding process for an unlimited period of time, even after API offboarding.

7.6. | All API offboarding policies, criteria and procedures MUST be documented

Requirement code: OB006
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

API documentation MUST include details of the offboarding process, including criteria and policies for offboarding, initiated by the API server deployer.

API Lifecycle management and versioning

8

Code	Requirement	Applicable roles	Standardization levels
LM001	*API specifications MUST be marked deprecated when they are no longer recommended for use*	API specifier	OA, TSA, FSA
LM002	*API specifications MUST be marked retired when they are no longer supported*	API specifier	OA, TSA, FSA
LM003	*API implementations MUST be marked deprecated when they are no longer recommended for use*	API server developer	OA, TSA, FSA
LM004	*API implementations MUST be marked retired when they are no longer supported*	API server developer	OA, TSA, FSA
LM005	*API deployments MUST be marked deprecated when they are no longer recommended for use*	API server deployer	OA, TSA, FSA
LM006	*API deployments MUST be marked retired when they are no longer supported*	API server deployer	OA, TSA, FSA
LM007	*An API client MUST be designed to handle non-breaking changes*	API client developer	OA, TSA, FSA
LM008	*An API specification MUST comply with Semantic Versioning 2.0.0*	API specifier	OA, TSA, FSA

8.1. | API specifications MUST be marked deprecated when they are no longer recommended for use

Requirement code: LM001
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

An API specifier MUST mark an API specification as deprecated (in the Dutch API library for healthcare) when it is no longer recommended for use. Specifications marked as deprecated are still supported but support may end soon.

An API specifier MUST mark an API specification as deprecated for a period of at least one year, before it MAY be marked retired.

API server developers SHOULD replace the use of deprecated API specifications with newer versions or alternative API specifications.

API server developers MUST mark API implementations as deprecated (in the Dutch API library for healthcare) when they are based on a deprecated API specification.

8.2. | API specifications MUST be marked retired when they are no longer supported

Requirement code: LM002
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

An API specifier MUST mark an API specification as retired (in the Dutch API library for healthcare) when it is no longer supported.

API server developers MUST replace the use of retired API specifications with newer versions or alternative API specifications.

API server developers MUST mark API implementations as retired when they are based on a retired API specification.

8.3. | API implementations MUST be marked deprecated when they are no longer recommended for use

Requirement code: LM003
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

An API server developer MUST mark an API implementation as deprecated (in the Dutch API library for healthcare) when it is no longer recommended for use. Implementations marked as deprecated are still supported but support may end soon.

An API server developer MUST mark an API implementation as deprecated for a period of at least one year, before it MAY be marked retired.

API server deployers SHOULD replace deprecated API implementations with newer versions or alternative API implementations.

API server deployers MUST mark an API deployment as deprecated (in the Dutch API library for healthcare) when it is based on a deprecated implementation.

8.4. | API implementations MUST be marked retired when they are no longer supported

Requirement code: LM004
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

An API server developer MUST mark an API implementation as retired (in the Dutch API library for healthcare) when it is no longer supported.

API server deployers MUST replace the use of retired API implementations with newer versions or alternative API implementations.

API server deployers MUST mark API deployments as retired (in the Dutch API library for healthcare) when they are based on a retired API implementation.

8.5. | API deployments MUST be marked deprecated when they are no longer recommended for use

Requirement code: LM005
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

An API server deployer MUST mark an API deployment as deprecated (in the Dutch API library for healthcare) when it is no longer recommended for use. Deployments marked as deprecated are still supported but support may end soon.

An API server deployer MUST mark an API deployment as deprecated for a period of at least one year, before it MAY be marked retired.

API client developers SHOULD replace the use of deprecated API employments with newer versions or alternative API implementations.

8.6. | API deployments MUST be marked retired when they are no longer supported

Requirement code: LM006
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

An API server deployer MUST mark an API deployment as retired (in the Dutch API library for healthcare) when it is no longer supported.

API client developers MUST replace the use of retired API deployments with newer versions or alternative API deployments.

8.7. | An API client MUST be designed to handle non-breaking changes

Requirement code: LM007
Applicable roles: API client deployer
Standardization levels: OA, TSA, FSA

An API client MUST be designed to handle non-breaking changes. This includes at least the non-breaking changes described in paragraph 3.7.3.

8.8. | An API specification MUST comply with Semantic Versioning 2.0.0

Requirement code: LM008
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

An API specification MUST comply with Semantic Versioning specification (SemVer) version 2.0.0.

API agreements

9

Code	Requirement	Applicable roles	Standardization levels
AG001	*API Service Levels MUST be openly and freely available*	API server deployer	OA, TSA, FSA
AG002	*API Access Restriction Policies MUST be openly and freely available*	API server deployer	OA, TSA, FSA
AG003	*Data Processing Policies MUST be openly and freely available*	API server deployer	OA, TSA, FSA
AG004	*Commercial agreements relating to the use of APIs by API client developers and API client deployers MUST be fair and transparent*	API server deployer	OA, TSA, FSA

9.1. | API Service Levels MUST be openly and freely available

Requirement code: AG001
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA
Intention: Protect API client developers, API users and API client deployers from unpredictable service level quality

API Service Levels MUST be openly and freely available.

API Service levels MUST be documented in the form of an agreement between API server deployer, API client developer and/or API client deployer. The API server deployer MAY require API client developers and/or API client deployers to sign a Service Level Agreement before using the API.

Service Levels MAY be provided on a best-efforts basis. If service levels are provided on a best-efforts basis, the SLA documentation MUST explicitly state relevant terms such as no liability for costs or charges incurred in the event of unavailability of the API.

Service Levels MAY be provided in tiers, such as free tiers without developer support and paid tiers that include developer support.

API Service Levels MUST be documented to the extent that API client developers and API client deployers are fully informed about the duties and responsibilities of each party involved and about the remedies or penalties for breaching these duties and responsibilities.

The API Service Levels MUST at least include a description of the service(s) provided and at least the following metrics by which the service is measured:

- Availability requirements, such as days and times the API is available and any restrictions for special days
- Details on planned outage and unplanned outage, expressed as a maximum percentage of total availability times
- Response time expectations, such as a percentage of calls that return within a given amount of time
- Details on technical support and support windows such as developer support or other technical support.

The API Service Levels MUST describe any restrictions on using the API. At least the following MUST be provided:

- Usage restrictions, such as number of calls per time unit or maximum call size and consequences for exceeding these restrictions
- Any terms and conditions regarding the use of the data acquired via the API, including requirements and responsibilities for secure and lawful use of data returned by the API such as retention and destruction policies.
- Any restrictions on what persons or organizations are allowed to access the API (API access restriction policies)

If the API server deployer provides online test tooling, Service Levels for test tooling MUST be described using the same metrics for availability, outage, usage restrictions, response times and technical support. If the API Server deployer provides offline test tooling, only details on technical support MUST be provided for the offline test tool.

The API Service Level Agreement MUST detail the process and restrictions for changing the agreement and/or service levels.

9.2. | API Access Restriction Policies MUST be openly and freely available

Requirement code: AG002
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA
Intention: Prevent information blocking strategies based on arbitrary access to APIs

Any restrictions on persons or organizations to access and/or use the API MUST be transparent and documented.

The API server deployer MUST NOT restrict persons or organizations from using the API for reasons other than the documented access restriction policies.

Access Restriction Policies MAY include an accreditation process and review of the API client system, API client developer organization and/or API client deployer organization. If an accreditation process is in place, details on how accreditation is achieved MUST be provided, including criteria for exclusion.

9.3. | Data Processing Policies MUST be openly and freely available

Requirement code: AG003
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA
Intention: Protect API user, API client developer and API client deployer organizations from unwanted use of API usage data

API Server Deployer Data processing policies MUST be documented and MUST include at least the details on how API usage will be monitored and what data will be stored as part of the monitoring process, such as IP addresses of the API deployer or API user.

This requirement concerns monitoring data that is not personal data. Requirements for the processing of personal data is the concern of the European General Data Protection Regulation (GDPR) and therefore is not part of this specification. However, API server deployers MAY document details on processing personal data as part of this requirement.

9.4. | Commercial charges relating to the use of APIs by API client developers and API client deployers MUST be predictable and openly and freely available

Requirement code: AG004
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA
Intention: Prevent information blocking strategies based on non-transparent pricing

When an API server deployer charges API client developers and/or API client deployers for using the API in a production environment, any fees MUST be openly and freely available.

Fees MUST be predictable, meaning that the API client developer and the API client deployer have enough pricing information to predict the cost of API usage for at least two years after signing the agreement.

API design rules

10

Design rules focus on the technical implementation of the APIs.

10.1. | Compliance with national API design rules

The Dutch national API strategy by Geonovum[7] contains design rules[8] that were considered when writing this chapter. Unfortunately, these rules would exclude the use of FHIR, which is the most considered base for any RESTful transaction in healthcare. It also focuses on the use of RESTful interfaces, while SOAP is still used for APIs as well, especially with IHE-profiles, like XDS, XCA, etc.

10.2. | Generic

Code	Requirement	Applicable roles	Standardization levels
DR001	*Interfaces MUST be defined in English*	API specifier	TSA, FSA
DR002	*Developers MUST only apply standard HTTP methods*	API specifier	TSA, FSA
DR003	*Developers MUST adhere to HTTP safety and idempotency semantics for operations*	API specifier	TSA, FSA
DR004	*Server communication MUST remain stateless*	API server developer	TSA, FSA
DR005	*Content relationships MUST be predictably documented*	API specifier	TSA, FSA
DR006	*Operations MUST be predictably documented*	API specifier	TSA, FSA
DR007	*API version MUST be accessible*	API specifier	TSA, FSA
DR008	*APIs MUST at least support the DEFLATE and gzip compression algorithms*	API specifier	TSA, FSA
DR009	*APIs MUST use the HTTP accept-encoding header for negotiating compression*	API client developer	TSA, FSA
DR010	*APIs MUST use the HTTP content-encoding header for negotiating compression*	API server developer	TSA, FSA
DR011	*JSON formatted content SHOULD comply to RFC8259 or its successor*	API server developer	TSA, FSA

7| https://docs.geostandaarden.nl/api/API-Strategie/
8| https://publicatie.centrumvoorstandaarden.nl/api/adr/

DR012	*APIs SHOULD be based on the NOTIFY-PULL exchange pattern rather than the PUSH exchange pattern*	API specifier	TSA, FSA
DR013	*System APIs SHOULD be designed independent from specific use cases and types of client systems or user*	API specifier	FSA, TSA
DR014	*Process APIs SHOULD be designed to reuse System APIs*	API specifier	FSA, TSA
DR015	*Experience APIs SHOULD be based on Process APIs*	API specifier	FSA, TSA
DR016	*System APIs SHOULD be based on the operations, messaging, or resource paradigm rather than the document paradigm*	API specifier	FSA, TSA

10.2.1. | Interfaces MUST be defined in English

Requirement code: DR001
Applicable roles: API specifier
Standardization levels: TSA, FSA
Dutch National API strategy:

Healthcare APIs are used internationally, which is why a lot are already available in English. Conforming to this will help with uniformity and attract non-Dutch developers.

10.2.2. | Developers MUST only apply standard HTTP methods

Requirement code: DR002
Applicable roles: API server developer
Standardization levels: TSA, FSA
Dutch National API strategy: API-03

The HTTP specification (rfc7231[9]) and the later introduced PATCH method specification (rfc5789[10]) offer a set of standard methods, where every method is designed with explicit semantics. Adhering to the HTTP specification is crucial since HTTP clients and middleware applications rely on standardized characteristics. Therefore, resources must be retrieved or manipulated using standard HTTP methods.

9| https://datatracker.ietf.org/doc/html/rfc7231
10| https://datatracker.ietf.org/doc/html/rfc5789

Method	Operation	Description
`GET`	Read	Retrieve a resource representation for the given URI. Data is only retrieved and never modified.
`POST`	Create	Create a resource. The receiver generates a new URI.
`PUT`	Create/update	Create a resource with the given URI or replace (full update) a resource when the resource already exists.
`PATCH`	Update	Partially updates an existing resource. The request only contains the resource modifications instead of the full resource representation.
`DELETE`	Delete	Remove a resource with the given URI.

The following table shows some examples of the use of standard HTTP methods:

Request	Description
`GET /Patient`	Retrieves a list of patients.
`GET /Patient/12`	Retrieves an individual patient.
`POST /Patient`	Creates a new patient.
`PUT /Patient/12`	Modifies Patient #12 completely.
`PATCH /Patient/12`	Modifies Patient #12 partially.
`DELETE /Patient/12`	Deletes Patient #12.

HTTP also defines other methods, e.g., `HEAD`, `OPTIONS` and `TRACE`. For this design rule, these operations are left out of scope.

10.2.3. | Developers MUST adhere to HTTP safety and idempotency semantics for operations

Requirement code: DR003
Applicable roles: API server developer
Standardization levels: TSA, FSA
Dutch National API strategy: API-01

DR006: Adhere to HTTP safety and idempotency semantics for operations

The HTTP protocol (rfc7231[11]) specifies whether an HTTP method should be considered safe and/or idempotent. These characteristics are important for clients and middleware applications because they should be considered when implementing caching and fault tolerance strategies.

Request methods are considered *safe* if their defined semantics are essentially read-only, i.e., the client does not request, and does not expect, any state change on the origin server because of applying a safe method to a target resource. A request method is considered *idempotent* if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request.

The following table describes which HTTP methods must behave as safe and/or idempotent:

Method	Safe	Idempotent
GET	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
POST	No	No
PUT	No	Yes
PATCH	No	No
DELETE	No	Yes

10.2.4. | Server communication MUST remain stateless

Requirement code: DR004
Applicable roles: API server developer
Standardization levels: TSA, FSA
Dutch National API strategy:

One of the key constraints of the REST architectural style is stateless communication between client and server. It means that every request from client to server must contain all the information necessary to understand the request. The server cannot take advantage of any stored session context on the server as it didn't memorize previous requests. Session state must therefore reside entirely on the client.

11| https://datatracker.ietf.org/doc/html/rfc7231

To properly understand this constraint, it's important to make a distinction between two distinct kinds of state:

- *Session state:* information about the interactions of an end-user with a particular client application within the same user session, such as the last page being viewed, the login state or form data in a multi-step registration process. Session state must reside entirely on the client (e.g., in the user's browser).
- *Resource state:* information that is permanently stored on the server beyond the scope of a single user session, such as the user's profile, a product purchase or information about a building. Resource state is persisted on the server and must be exchanged between client and server (in both directions) using representations as part of the request or response payload. This is where the term *REpresentational State Transfer (REST)* originates from.

It's a misconception that there should be no state at all. The stateless communication constraint should be seen from the server's point of view and states that the server should not be aware of any *session state*.

Stateless communication offers many advantages, including:

- *Simplicity* is increased because the server doesn't have to memorize or retrieve the session state while processing requests
- *Scalability* is improved because not having to incorporate the session state across multiple requests enables higher concurrency and performance
- *Observability* is improved since every request can be monitored or analysed in isolation without having to incorporate session context from other requests
- *Reliability* is improved because it simplifies the task of recovering from partial failures since the server doesn't have to maintain, update or communicate the session state. One failing request does not influence other requests (depending on the nature of the failure of course).

In the context of REST APIs, the server must not maintain or require any notion of the functionality of the client application and the corresponding end-user interactions. To achieve full decoupling between client and server, and to benefit from the advantages mentioned above, no session state must reside on the server. Session state must therefore reside entirely on the client.

The client of a REST API could be a variety of applications such as a browser application, a mobile or desktop application or even another server serving as a backend component for another client. REST APIs should therefore be completely client agnostic.

10.2.5. | Content relationship MUST be predictably implemented

Requirement code: DR005
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

When using several content relationships within an API, adding more relationships must adhere to the same implementation choices.

E.g., if a child resource is already available through stacking of endpoint, another child resource must be made available in the same manner.

10.2.6. | Operations MUST be predictably implemented

Requirement code: DR006
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

An API can be made more accessible using operations. The operations in one API must all be implemented in the same way.

E.g., if an operation is called with a prefix, the same prefix must be used for every other operation on the same API.

10.2.7. | API version MUST be accessible

Requirement code: DR007
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

The version of an API can be useful in development and debugging, for this reason the API version must be accessible through the API.

This can be done in several ways, some examples (but not limited to these) are:

- In the URL-path (e.g., /v1/api/...)
- In a HTTP header (e.g., in Accept (request) and Content-Type (response))
- Available through a separate request (e.g., in response to /api/version)

10.2.8. | APIs MUST at least support the DEFLATE and gzip compression algorithms

Requirement code: DR008
Applicable roles: API server developer
Standardization levels: TSA, FSA

The use of compression algorithms can reduce communication size. This is especially useful when transferring large data objects.

The algorithms DEFLATE and gzip are widely used in HTTP communication.

10.2.9. | APIs MUST support the use of HTTP accept-encoding and content-encoding header fields for negotiating compression

Requirement code: DR009
Applicable roles: API client developer, API server developer
Standardization levels: TSA, FSA

The accept-encoding and content-encoding header fields are used to negotiate the compression algorithms. The API client will use the accept-encoding header field to communicate which compression algorithms are supported. The API server will use the content-encoding header field to confirm which compression algorithm is used.

10.2.10. | JSON formatted content SHOULD comply to RFC8259 *or its successor*

Requirement code: DR011
Applicable roles: API server developer
Standardization levels: TSA, FSA

The design of JSON is described by IETF in RFC 8259. Any use of JSON SHOULD be compliant with this RFC or its successor.

10.2.11. | APIs SHOULD be based on the NOTIFIED PULL exchange pattern rather than the PUSH exchange pattern

Requirement code: DR012
Applicable roles: API specifier
Standardization levels: TSA, FSA

When describing an API where data is being transferred to another party, the exchange pattern *'NOTIFIED PULL'* SHOULD be used rather than PUSH. This way, the receiving party can decide when or how the transferred data is received and processed.

10.2.12. | System APIs SHOULD be designed independent of specific use cases and types of client systems or users

Requirement code: DR013
Applicable roles: API server developer
Standardization levels: TSA, FSA

Making the system APIs independent of a specific use case encourages the reuse of these APIs.

10.2.13. | Process APIs SHOULD be designed to reuse System APIs

Requirement code: DR014
Applicable roles: API server developer
Standardization levels: TSA, FSA

Process APIs make use of the System APIs. It is encouraged to reuse already existing System APIs.

10.2.14. | Experience APIs SHOULD be based on Process APIs

Requirement code: DR015
Applicable roles: API server developer
Standardization levels: TSA, FSA

Experience APIs are by definition based on Process APIs, this SHOULD always be the case.

10.2.15. | System APIs SHOULD be based on the operations, messaging, or resource paradigm rather than the document paradigm

Requirement code: DR016
Applicable roles: API server developer
Standardization levels: TSA, FSA

To be fully atomic the document paradigm SHOULD be avoided in system APIs.

10.3 | SOAP

Code	Requirement	Applicable roles	Standardization levels
DR-S001	*SOAP APIs MAY use MTOM/ XOP for formatting binary data*	API specifier	TSA, FSA
DR-S002	*SOAP based API clients MUST support MTOM/XOP formatted binary data*	API client developer	TSA, FSA

10.3.1. | APIs MAY use MTOM/XOP for formatting binary data

Requirement code: DR-S001
Applicable roles: API specifier
Standardization levels: TSA, FSA
Dutch National API strategy:

When a SOAP message contains a large binary object, it can be optimized for processing by using MTOM/XOP. APIs MAY use MTOM/XOP to create a multipart MIME to link from the SOAP body to a mime part which holds the binary data.

10.3.2. | API clients MUST support MTOM/XOP formatting of binary data

Requirement code: DR-S002
Applicable roles: API client developer
Standardization levels: TSA, FSA
Dutch National API strategy:

Design rule DR-S001 determined that an API may use MTOM/XOP when it believes it is necessary to optimize processing of the message. To be able to use these APIs, API clients MUST support MTOM/XOP as well.

10.4. | RESTful

The REST architectural style is centred around the concept of a resource. A resource is the key abstraction of information, where every piece of information is named by assigning a globally unique URI (Uniform Resource Identifier). Resources describe *things*, which can vary between physical objects (e.g., a building or a person) and more abstract concepts (e.g., a permit or an event).

Code	Requirement	Applicable roles	Standardization levels
DR-R001	*APIs MUST use nouns to name resources*	API specifier	TSA, FSA
DR-R002	*APIs MUST use singular nouns to name collection resources*	API specifier	TSA, FSA
DR-R003	*APIs MUST hide irrelevant implementation details*	API server developer	TSA, FSA
DR-R004	*RESTful APIs MUST support both JSON and XML formatting*	API specifier	TSA, FSA
DR-R005	*RESTful API clients MUST at least support JSON or XML formatting*	API client developer	TSA, FSA
DR-R006	*RESTful APIs MAY support BSON formatting*	API specifier	TSA, FSA
DR-R007	*RESTful APIs MUST use the ACCEPT HTTP header for content negotiation*	API server developer	TSA, FSA
DR-R008	*Restful APIs MUST user the CONTENT-type HTTP header for content negotiation*	API client developer	TSA, FSA

10.4.1. | APIs MUST use nouns to name resources

Requirement code: DR-R001
Applicable roles: API specifier
Standardization levels: TSA, FSA
Dutch National API strategy: corresponds to API-05

Because resources describe things (and thus not actions), resources are referred to using nouns (instead of verbs) that are relevant from the perspective of the user of the API.

A few correct examples of nouns as part of FHIR:

- Patient
- Observation
- AllergyIntolerance

10.4.2. | APIs MUST use singular nouns to name collection resources

Requirement code: DR-R002
Applicable roles: API specifier
Standardization levels: TSA, FSA
Dutch National API strategy:

Resources can be grouped into collections, which are resources in their own right and can typically be paged, sorted and filtered. Most often all collection members have the same type, but this is not necessarily the case. A resource describing multiple things is called a collection resource. Collection resources typically contain references to the underlying singular resources.

A collection resource could still contain only one contained resource, therefore the path segment describing the name of the collection resource MUST be written in the singular form.

Example of how to collect a collection of resources in FHIR:
`https://api.example.org/Patient`
`https://api.example.org/Observation?code=http://loinc.org|29463-7`

10.4.3. | APIs MUST hide irrelevant implementation details

Requirement code: DR-R003
Applicable roles: API server developer
Standardization levels: TSA, FSA
Dutch National API strategy: API-53

An API should not expose implementation details of the underlying application. The primary motivation behind this design rule is that an API design must focus on usability for the client, regardless of the implementation details under the hood. The API, application and infrastructure need to be able to evolve independently to ease the task of maintaining backwards compatibility for APIs during an agile development process.

A few examples of implementation details:
- The API design should not necessarily be a 1-to-1 mapping of the underlying domain or persistence model
- The API should not expose information about the technical components being used, such as development platforms/frameworks or database systems
- The API should offer client-friendly attribute names and values, while persisted data may contain abbreviated terms or serializations which might be cumbersome for consumption

10.4.4. | APIs MUST support both JSON and XML formatting

Requirement code: DR-R004
Applicable roles: API server developer
Standardization levels: TSA, FSA
Dutch National API strategy:

JSON and XML both have their advantages and disadvantages. API clients can focus on either standard for all their API calls when all APIs support both. Therefore, APIs MUST support both JSON and XML formatting.

APIs MUST be able to convert consistently between the two. Converting from one to the other and back to the first MUST result in the exact same resource.

10.4.5. | API clients MUST at least support JSON or XML formatting

Requirement code: DR-R005
Applicable roles: API client developer
Standardization levels: TSA, FSA
Dutch National API strategy:

As described in DR-R004, both JSON and XML will be supported from the APIs. API clients MUST therefore at least support one of the two formats. In the end this means the focus can be on either XML or JSON for all calls to available APIs.

10.4.6. | APIs MAY support BSON formatting

Requirement code: DR-R006
Applicable roles: API server developer
Standardization levels: TSA, FSA
Dutch National API strategy:

BSON are used to communicate JSON in a binary object. This is an innovative technology that could improve performance in larger JSON objects. APIs MAY therefore support BSON formatting.

10.4.7. | APIs MUST use the Accept header for content negotiation

Requirement code: DR-R007
Applicable roles: API server developer, API client developer
Standardization levels: TSA, FSA
Dutch National API strategy:

The HTTP header Accept is used to communicate which content types can be understood by the API clients. Therefore, the API client MUST use the Accept header to communicate the content types, while the API server MUST use the value to decide the response content type.

10.4.8. APIs MUST use the Content-Type header for content negotiation

Requirement code: DR-R008
Applicable roles: API client developer, API server developer
Standardization levels: TSA, FSA
Dutch National API strategy:

The HTTP header Content-Type is used to communicate which content type is used in communication between client and server. Therefore, the API server MUST use the Content-Type header to communicate the content type, while the API client MUST use the value to decide how to process the response.

API security

11

11.1. | Generic

Code	Requirement	Applicable roles	Standardization levels
SC001	*API specifications MUST comply with Dutch NCSC guidelines for web applications*	API specifier	OA, TSA, FSA
SC002	*API implementations MUST comply with Dutch NCSC guidelines for web applications*	API server developer	OA, TSA, FSA
SC003	*API deployments MUST comply with Dutch NCSC guidelines for web applications*	API server deployer	OA, TSA, FSA
SC004	*API deployments MUST comply with Dutch NCSC guidelines for Transport Layer Security*	API server deployer	OA, TSA, FSA
SC005	*An API MUST provide audit logging conforming to NEN7513*	API server developer	OA, TSA, FSA
SC006	*Specifications for 'System APIs' MUST use authentication models that are not specific to a use case or (type of) client or user*	API specifier	TSA, FSA
SC007	*APIs MUST use fully standardized models for identification and authentication*	API specifier	FSA
SC008	*All tokens used for client authentication MUST be signed using asymmetrical encryption*	API specifier	TSA, FSA

11.1.1. | API specifications MUST comply with Dutch NCSC guidelines for web applications

Requirement code: SC001
Applicable roles: API specifier
Standardization levels: OA, TSA, FSA

The Dutch governmental organization NCSC (National Cyber Security Centre) has specified security guidelines for web applications. The API specifications MUST comply with these guidelines.

Compliance check will be limited to a confirmation by the specifier.

11.1.2. | API implementations MUST comply with Dutch NCSC guidelines for web applications

Requirement code: SC002
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

The Dutch governmental organization NCSC (National Cyber Security Centre) has specified security guidelines for web applications. The API implementations MUST comply with these guidelines.

Compliance check will be limited to a confirmation by the server developer.

11.1.3. | API deployments MUST comply with Dutch NCSC guidelines for web applications

Requirement code: SC003
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

The Dutch governmental organization NCSC (National Cyber Security Centre) has specified security guidelines for web applications. The API deployments MUST comply with these guidelines.

Compliance check will be limited to a confirmation by the server deployer.

11.1.4. | API deployments MUST comply with Dutch NCSC guidelines for Transport Layer Security

Requirement code: SC004
Applicable roles: API server deployer
Standardization levels: OA, TSA, FSA

The Dutch governmental organization NCSC (National Cyber Security Centre) has specified guidelines for use of TLS within web applications. The API deployments MUST comply with these guidelines.

The status of the used versions, algorithms, key size & choice of
groups and options SHOULD at least be sufficient. The status ‘insufficient’ is not allowed.

Compliance check will be limited to a confirmation by the server deployer.

11.1.5. | An API MUST provide audit logging conforming to NEN7513

Requirement code: SC005
Applicable roles: API server developer
Standardization levels: OA, TSA, FSA

The audit logging should conform to what is described in NEN7513, at least the content matches to what is described in the same NEN7513.

11.1.6. | Specifications for 'System APIs' MUST use authentication and authorization models that are not specific to a use case or (type of) client or user

Requirement code: SC006
Applicable roles: API specifier
Standardization levels: TSA, FSA

To ensure the disconnection between System APIs and other types of APIs, the authentication and authorization models used by System APIs cannot be specific to a use case or (type of) client or user.

11.1.7. | APIs MUST use fully standardized models for identification and authentication

Requirement code: SC007
Applicable roles: API specifier
Standardization levels: FSA

For both identification and authentication, a fully standardized model MUST be used in the API specification.

11.1.8. | All tokens used for client authentication MUST be signed using asymmetrical encryption

Requirement code: SC008
Applicable roles: API specifier
Standardization levels: TSA, FSA

Tokens that are used for client authentication MUST be signed using asymmetrical encryption.

11.1.9. | APIs MUST use generic services/functions

Requirement code: SC009
Applicable roles: API specifier
Standardization levels: TSA, FSA

APIs MUST use generic services/functions when they are nationally prescribed for use.
Examples of generic services/functions could be:
- Patient Consent
- Addressing
- Authorization models

11.2. | SOAP

Code	Requirement	Applicable roles	Standardization levels
SC-S001	*APIs SHOULD use WS-Security to ensure message confidentiality and integrity as well for adding security tokens*	API specifier	TSA, FSA
SC-S001	*APIs SHOULD use the SAML Token Security Model*	API server developer	OA, TSA, FSA

11.2.1. | APIs SHOULD use WS-Security to ensure message confidentiality and integrity for adding security tokens

Requirement code: SC-S001
Applicable roles: API specifier
Standardization levels: TSA, FSA

SOAP APIs SHOULD use WS-Security to ensure message confidentiality and integrity for adding security tokens.

11.2.2. | APIs SHOULD use the SAML Token Security Model

Requirement code: SC-S002
Applicable roles: API specifier
Standardization levels: TSA, FSA

SAML is commonly used in SOAP APIs. Therefore the SOAP APIs SHOULD use the SAML Token Security Model.

11.3. | RESTful

Code	Requirement	Applicable roles	Standardization levels
SC-R001	*APIs MUST comply with RFC7523 for client authentication and for requesting oAuth2 access tokens*	API specifier	TSA, FSA
SC-R002	*APIs MUST use oAuth2 for authorization flows*	API specifier	TSA, FSA
SC-R003	*APIs MAY use OpenID Connect to achieve Single-Sign-On when requesting oAuth access tokens whenever this doesn't conflict with existing regulations*	API specifier	TSA, FSA
SC-R004	*JWT tokens used for client authentication and authorization grants MUST comply with RFC7515 and RFC7518*	API specifier	TSA, FSA

11.3.1. | APIs MUST comply with RFC7523 or its successor for client authentication and for requesting oAuth2 access tokens

Requirement code: SC-R001
Applicable roles: API specifier
Standardization levels: TSA, FSA

When using REST for backchannel communication the APIs MUST comply with RFC7513 or its successor for client authentication and for requesting at least oAuth 2.0 access tokens.

RFC7513 describes the JSON Web Token profile for oAuth 2.0 Client Authentication and Authorization Grants.

11.3.2. | APIs SHOULD use OpenID Connect to achieve Single-Sign-On when requesting oAuth access tokens

Requirement code: SC-R002
Applicable roles: API specifier
Standardization levels: TSA, FSA

As far as existing regulations allow, OpenID Connect SHOULD be used to achieve Single-Sign-On when requesting oAuth access tokens.

11.3.3. | JWT tokens used for client authentication and authorization grants MUST comply with RFC7515 and RFC7518, or its successors

Requirement code: SC-R003
Applicable roles: API specifier
Standardization levels: TSA, FSA

When using JWT tokens for client authentication and authorization grants, they MUST comply with RFC7515 (or its successor) and RFC7518 (or its successor).

RFC7515 describes the JSON Web Signature, a data structure representing a digitally signed or MACed (Message Authentication Codes) message.

RFC7518 describes the JSON Web Algorithms

Health Information Standards compliance

12

Code	Requirement	Applicable roles	Standardization levels
IS001	*In order to be fully standardized, an API specification MUST be approved by an authoritative body*	API specifier	FSA
IS002	*In order to be fully standardized, an API implementation MUST be approved by an authoritative body during a formal testing and qualification process*	API server developer	FSA
IS003	*All API input and output data SHOULD comply with ZIB specifications*	API specifier	FSA

12.1. | In order to be fully standardized, an API specification MUST be approved by an authoritative body

Requirement code: IS001
Applicable roles: API specifier
Standardization levels: FSA

An API specification is 'fully standardized' when it is approved as a standard by an authoritative body.

12.2. | In order to be fully standardized, an API implementation MUST be approved by an authoritative body during a formal testing and qualification process

Requirement code: IS002
Applicable roles: API server developer
Standardization levels: FSA

An API implementation is fully standardized when it is approved by a national standardization organization, or by the national branch of an international standardization organization. Approval is given only as the result of a formal test or qualification process. Proof must be provided for any compliance claim made by the API server developer.

12.3. | All API input and output data SHOULD comply with ZIB specifications

Requirement code: IS003
Applicable roles: API specifier
Standardization levels: FSA

API input data, such as parameters and post data, SHOULD comply with ZIB specifications where applicable. Applicable refers to input or output data that represents general health and care concepts.

API output data, such as resources, documents or procedure results SHOULD comply with ZIB specifications where applicable.

If no ZIB specification is available that governs the contents of certain API input or output but the input or output data represents general health and care concept, the API specifier MAY submit a ZIB change request to the Nictiz ZIB centre.

Nictiz is de Nederlandse kennisorganisatie voor digitale informatievoorziening in de zorg. Nictiz ontwikkelt een visie op het zorginformatiestelsel en de architectuur die dat stelsel ondersteunt. We ontwikkelen en beheren standaarden die digitale informatievoorziening mogelijk maken en zorgen ervoor dat zorginformatie eenduidig kan worden vastgelegd en uitgewisseld. Daarnaast adviseren we en delen we kennis over digitale informatievoorziening in de zorg. Daarbij kijken we niet alleen naar Nederland, maar ook naar wat er internationaal gebeurt.

Nictiz | Postbus 19121 | 2500 CC Den Haag | Oude Middenweg 55 | 2491 AC Den Haag
070 - 317 34 50 | www.nictiz.nl

Nictiz
Betere gezondheid
door betere informatie