

CTP Data Model and API, rev. 2.13

October 2015

Contents

1	Introduction	1
2	Terms and definitions	3
3	Overview of CTP	5
3.1	Principles of the CTP API	5
3.2	Introducing the CTP data model	6
4	The CTP Data Model	9
4.1	JSON representation	10
4.1.1	API Resources URLs and identifiers	10
4.1.2	Resource URLs	10
4.1.3	Identifiers	11
4.1.4	JSON notations used in the specification	11
4.1.5	Common properties of CTP JSON objects	11
4.1.5.1	The JSON "self" property	12
4.1.5.2	The JSON "scope" property	12
4.1.5.3	the JSON "name" property	12
4.1.5.4	the JSON "annotation" property	12
4.1.6	Collections	13
4.2	Resources	15
4.2.1	CTP entry point (CtpBase)	15
4.2.1.1	Description	15
4.2.1.2	Encoding	16
4.2.2	Service-views	16
4.2.2.1	Definition	16
4.2.2.2	Encoding	17
4.2.3	Assets	18
4.2.3.1	Definition	18
4.2.3.2	Encoding	18
4.2.4	Attributes	19

4.2.4.1	Definition	19
4.2.4.2	Encoding	19
4.2.5	Metrics	20
4.2.5.1	Definition	20
4.2.5.2	Encoding	20
4.2.6	Measurements	21
4.2.6.1	Definition	21
4.2.6.2	Encoding	24
4.2.7	Triggers	25
4.2.7.1	Definition	25
4.2.7.2	Encoding	27
4.2.8	Log entries	27
4.2.8.1	Definition	27
4.2.8.2	Encoding of normal log entries	28
4.2.8.3	Encoding of error log entries	29
4.3	Service classes and asset classes	29
4.4	Relating measurement results and metrics	30
5	The API	35
5.1	Application of RESTful principles	35
5.1.1	HTTP methods	35
5.1.2	Application of HATEOAS	35
5.1.3	Content type	35
5.1.4	Status codes	36
5.1.5	Error response bodies	36
5.2	Specification of the API calls	36
5.2.1	GET {CtpBase}	37
5.2.1.1	Description	37
5.2.1.2	Request query string	37
5.2.1.3	Request body	37
5.2.1.4	Response body	37
5.2.2	GET {CtpBase}/serviceViews	37
5.2.2.1	Description	37
5.2.2.2	Request query string	37
5.2.2.3	Request body	37
5.2.2.4	Response body	38
5.2.3	GET {CtpBase}/serviceViews/{id}	38
5.2.3.1	Description	38
5.2.3.2	Request query string	38

5.2.3.3	Request body	38
5.2.3.4	Response body	38
5.2.4	GET {CtpBase}/serviceViews/{id}/dependencies	38
5.2.4.1	Description	38
5.2.4.2	Request query string	38
5.2.4.3	Request body	39
5.2.4.4	Response body	39
5.2.5	GET {CtpBase}/serviceViews/{id}/assets	39
5.2.5.1	Description	39
5.2.5.2	Request query string	39
5.2.5.3	Request body	39
5.2.5.4	Response body	39
5.2.6	GET {CtpBase}/assets/{id}	39
5.2.6.1	Description	39
5.2.6.2	Request query string	39
5.2.6.3	Request body	39
5.2.6.4	Response body	40
5.2.7	GET {CtpBase}/assets/{id}/attributes	40
5.2.7.1	Description	40
5.2.7.2	Request query string	40
5.2.7.3	Request body	40
5.2.7.4	Response body	40
5.2.8	GET {CtpBase}/attributes/{id}	40
5.2.8.1	Description	40
5.2.8.2	Request query string	40
5.2.8.3	Request body	40
5.2.8.4	Response body	40
5.2.9	GET {CtpBase}/attributes/{id}/measurements	41
5.2.9.1	Description	41
5.2.9.2	Request query string	41
5.2.9.3	Request body	41
5.2.9.4	Response body	41
5.2.10	GET {CtpBase}/measurements/{id}	41
5.2.10.1	Description	41
5.2.10.2	Request query string	41
5.2.10.3	Request body	41
5.2.10.4	Response body	41
5.2.11	PUT {CtpBase}/measurements/{id}?x=state`	42
5.2.11.1	Description	42

5.2.11.2	Request query string	42
5.2.11.3	Request body	42
5.2.11.4	Response body	42
5.2.12	GET {CtpBase}/metrics	42
5.2.12.1	Description	42
5.2.12.2	Request query string	42
5.2.12.3	Request body	42
5.2.12.4	Response body	43
5.2.13	GET {CtpBase}/metrics/{id}	43
5.2.13.1	Description	43
5.2.13.2	Request query string	43
5.2.13.3	Request body	43
5.2.13.4	Response body	43
5.2.14	GET {CtpBase}/serviceViews/{id}/triggers	43
5.2.14.1	Description	43
5.2.14.2	Request query string	43
5.2.14.3	Request body	43
5.2.14.4	Response body	43
5.2.15	GET {CtpBase}/triggers/{id}	44
5.2.15.1	Description	44
5.2.15.2	Request query string	44
5.2.15.3	Request body	44
5.2.15.4	Response body	44
5.2.16	POST {CtpBase}/serviceViews/{id}/triggers	44
5.2.16.1	Description	44
5.2.16.2	Request query string	44
5.2.16.3	Request body	44
5.2.16.4	Response body	44
5.2.17	DELETE {CtpBase}/triggers/{id}	44
5.2.17.1	Description	44
5.2.17.2	Request query string	45
5.2.17.3	Request body	45
5.2.17.4	Response body	45
5.2.18	GET {CtpBase}/serviceViews/{id}/logs	45
5.2.18.1	Description	45
5.2.18.2	Request query string	45
5.2.18.3	Request body	45
5.2.18.4	Response body	45
5.2.19	GET {CtpBase}/logs/{id}	46

5.2.19.1	Description	46
5.2.19.2	Request query string	46
5.2.19.3	Request body	46
5.2.19.4	Response body	46
5.3	Evaluating triggers and objectives	46
5.3.1	Evaluating objectives	46
5.3.2	Evaluating triggers	47
5.4	CTPScript: a condition description language	47
5.4.1	Basic Grammar	48
5.4.2	Definitions borrowed from other standards	49
5.4.2.1	StringLiteral	49
5.4.2.2	NumericLiteral	49
5.4.2.3	IdentifierName	49
5.4.3	Types	49
5.4.3.1	The 7 types of CTPScript	49
5.4.3.2	The typeOf() pseudo-function	50
5.4.3.3	The getField() pseudo-function.	50
5.4.4	Functions	50
5.4.4.1	toString(a): string	50
5.4.4.2	toBoolean(a): Boolean	51
5.4.4.3	toNumber(a): number	51
5.4.4.4	Array.min(): (any type)	51
5.4.4.5	Array.max(): (any type)	51
5.4.4.6	matchRegexp(r, v): boolean	52
5.4.4.7	select(s, a): table	52
5.4.4.8	timeUTC(s): number	52
5.4.5	Boolean operators , &&	53
5.4.6	Comparison Operators <, <=, >, >=, ==, !=	53
5.4.7	Multiplicative and additive binary operators: +, -, *, /, %	54
5.4.8	Identifiers	54
5.4.9	Evaluating CTPScript expressions	55
6	Security considerations	56
6.1	Four security requirements for the CTP API	56
6.2	Example of solutions	57
6.3	Recommended security mechanisms	57
7	References	58
7.1	Bibliography	58

A	API Examples	60
A.1	Step 1: Connecting to the entry point of the CTP server	62
A.2	Step 2: Discovering the service-views accessible to the customer	62
A.3	Step 3: Accessing information on the service-view	63
A.4	Step 4: Discovering the list of assets of the service-view	63
A.5	Step 5: Discovering the details of the first asset of the service-view	64
A.6	Step 8: Discovering the list of security attributes of the asset	64
A.7	Step 7: Discovering the details of the second security attribute of the asset	65
A.8	Step 8: Discovering the list of measurements available for this attribute.	65
A.9	Step 9: Discovering the details of the first measurement available for the selected security attribute	66
A.10	Step 10: Getting information about the metric used in the measurement	67
B	Accessing information about the supply chain	68
C	CTP project history	69

Revision history

Revision	Date	Comment
1.0	Oct. 2012	First draft / brainstorming document
2.0	Nov. 2012	Complete model change. Evolution to a draft standard.
2.1	Dec. 2012	Model update.
2.2	Jan 15, 2013	Transition from Google Doc to Microsoft Word format. First release for expert review.
2.3	Feb 4, 2013	Minor changes to emphasize optional “service classes”.
2.4	May 8, 2013	Clarification and minor changes based on expert review feedback.
2.5	June 15, 2013	Removed resource-group abstraction in the data model.
2.6	July 3, 2013	Clarification and minor changes based on internal feedback.
2.8	Nov. 25, 2014	Major revision after creation of Cloud Trust working group and feedback from the past 12 months.
2.9	Jan. 20, 2015	Revision after internal review and before wider release to research community outside CSA. The API was switched from XML to JSON, to acknowledge current trends in cloud API development and adopt a more agile approach.
2.10	June 30, 2015	Major revision after open public review. Abandoned MS Word for more professional tools dedicated to technical documentation.
2.11	July 16, 2015	Homogenize common attributes in CTP resources.
2.12	Sep. 9, 2015	Simplification of the data model and change in the format of items in collections.
2.13	Oct. 6, 2015	Added <i>status</i> and <i>statusUpdateTime</i> to triggers

Acknowledgments

Lead author

Alain Pannetrat – apannetrat@cloudsecurityalliance.org

Reviewers and contributors

Bernd Jaeger

Giles Hogben

Jesus Luna

Konstantinos Mantzoukas

Laura Lindsay

Martin Gilje Jaatun

Massimiliano Rak

Mauro Turtur

Michaela Iorga

Michela D’Errico

Ronald Tse

European research project support

A4Cloud - <http://www.a4cloud.eu/>

CUMULUS - <http://www.cumulus-project.eu/>

SPECS - <http://www.specs-project.eu/>

Chapter 1

Introduction

The Cloud Trust Protocol (CTP) is designed to be a mechanism by which cloud service customers can ask for and receive information related to the security of the services they use in the cloud, promoting transparency and trust.

Why CTP?

The creation of CTP was motivated by the current gaps in cloud security monitoring. Though many cloud providers offer solutions such as dashboards or ad-hoc APIs in order to enable cloud customers to monitor cloud services, these solutions have some strong shortcomings when it comes to the security of cloud services. First, they are usually proprietary and not interoperable. Second, they often target performance indicators rather than security. Third, the measurements conducted by providers are often imprecisely described in terms of parameters and scope. Fourth, each provider typically uses subtlety different metrics for similar attributes, thus making any comparison with a customer-defined baseline challenging while additionally making the ranking of competing services near impossible. The API described in this document aims to solve the first and third issues, while creating a platform tailored to support a solution to the other two remaining issues, which will require further standardization initiatives by relevant stakeholder.

This document focuses on the definition of the CTP Data Model and Application Programming Interface (API), including:

- The format of CTP messages exchanged between cloud service customers and providers.
- The modeling of concepts such as “security attributes”, “objectives”, “measurement results” and “triggers” in machine readable format.
- The means to define the scope of the service to which CTP monitoring queries apply.

However, this document does not provide a specification of the “security attributes” (and associated metrics) that are queried by CTP. Such a specification will be provided by the Cloud Security Alliance in a separate document, and will likely be influenced by upcoming standards such as [ISO_19086](#). CTP also offers implementers the choice to define and adopt their own set of security attributes and related metrics.

This document is organized as follows.

- [Section 2](#) provides some key **terms and definitions** that are used throughout this document, borrowing from relevant key standards.
 - [Section 3](#) offers a general introductory **overview of CTP**.
 - [Section 4](#) describes the **CTP data model**, defining the main concepts that are used to represent security information related to cloud services in CTP.
 - [Section 5](#) specifies the **RESTful CTP API** that implements the model described in section 4. It also specifies the CTPScript language used in “triggers” and “objectives” and describes when they should be evaluated.
 - [section 6](#) provides requirements and recommendations for securing the CTP API.
-

The document concludes with an **appendix**, which offers:

- An [example](#) of a message-level dialogue between a CTP client and a CTP server.
 - Information on the use of CTP [across a supply chain](#), where one cloud provider offering CTP uses other sub-providers that also implement CTP.
 - A brief overview of the [CTP project history](#) and the changes introduced since the previous version of the CTP model published in 2011.
 - External [References](#).
-

Chapter 2

Terms and definitions

In this section we provide some key terms and definitions used in the CTP API, which mostly originate from ISO/IEC [\[ISO 27000\]](#), with a few additions from other relevant sources.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [\[RFC_2119\]](#).

Note

This document does not use the term "**measure**" proposed in [\[ISO_27000\]](#) to avoid potential confusion arising from the multiple conflicting uses of this term, as highlighted in [\[NIST\]](#).

Asset

An asset is any tangible or intangible component of an information system that *has value to an organization* (source: derived from ISO/IEC IS 13335-1 and [\[ENISA\]](#)).

Attribute

An attribute is a *property or characteristic of an object that can be distinguished quantitatively or qualitatively by human or automated means* (source: [ISO/IEC 27000:2014](#) from ISO/IEC 15939:2007).

Security attribute

A security attribute is an attribute that describes a property related to confidentiality, integrity or availability.

Measurement

A measurement is a *process to determine a value* (source: [ISO/IEC 27000:2014](#)). In this document, a measurement describes the process resulting in the production of a *measurement result*, which is the evaluation of the value of an attribute.

Measurement parameter

A measurement parameter is a variable representing a characteristic of the measurement process.

Measurement parameter value

A measurement parameter value describes the value assigned to a measurement parameter.

Measurement result

A measurement result is a *value that expresses a qualitative or quantitative assessment of an attribute*. (Based on both [\[NIST\]](#) and [ISO/IEC 27000:2014](#) from 15939:2007).

Metric

A metric is a *standard of measurement that defines the conditions and the rules for performing the measurement and for understanding the results of a measurement*. (source: [\[NIST\]](#))

Objective

An objective is defined as *a result to be achieved*. (source: [ISO/IEC 27000:2014](#)). In this document an *objective* describes more specifically the *measurement results* that should be achieved in order to represent a desired level of security.

CTP Client

A CTP client is a software client that uses the CTP API to make queries and receive responses from a CTP Server.

CTP Server

A CTP server is a software server that implements the CTP API in order to describe the level of security of a cloud system to CTP clients, in near real-time, through a set of attributes.

Chapter 3

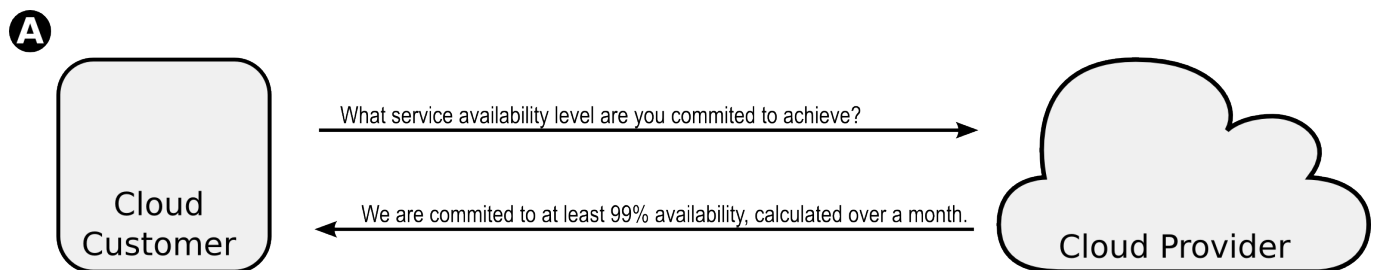
Overview of CTP

3.1 Principles of the CTP API

The CTP API is designed to be a RESTful¹ protocol that cloud service customers can use to query a cloud service provider (CSP) on current security attributes related to a cloud service such as the current level of availability of the service or information on the last vulnerability assessment. This is normally done through a classical query-response approach driven by the customer. CTP also enables customers to define “triggers” in order to receive alerts when specific conditions are met, through a standard XMPP-based notification mechanism. The CTP API additionally provides access to a log facility that can be used to store and access security events generated by triggers.

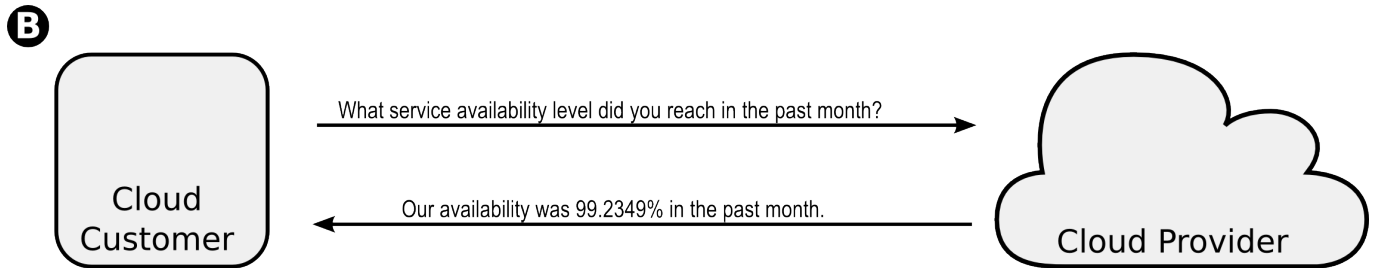
It is important to emphasize that CTP mainly proposes a unified standardized API to present measurement results related to cloud security. As such, the CTP API does not cover the actual monitoring infrastructure and related technologies that are used to gather, store and analyze events in order to produce these measurement results.

The following diagrams provide a general idea of the principles of CTP through 3 simple use cases where a cloud service customer uses CTP to query a cloud service provider about security attributes of its services. In the first figure (A), the cloud service customer uses CTP to query a cloud service provider about the service availability level that it is committed to provide. In CTP the result of this query is called an “objective” — or “service level objective” — since it describes what the provider aims to achieve, as typically described in an SLA.

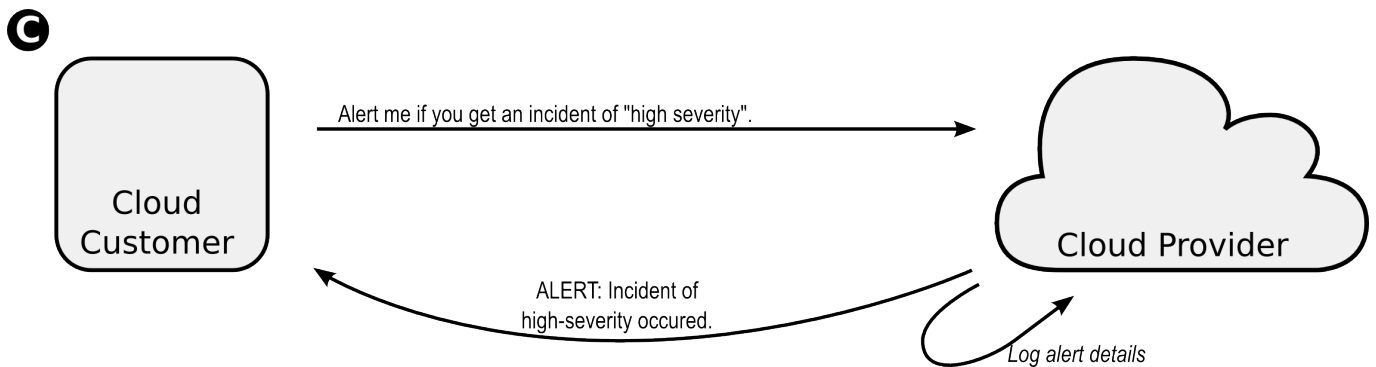


Next, in figure (B), the cloud service customer queries the cloud provider about the service availability level that was actually achieved in the past month. The result of this query is called a “measurement result” in CTP, since it describes the result of a service level measurement reported by the cloud provider. Both this measurement result and the objective in the previous example apply to the same security attribute informally called “availability” here.

¹ REST describes the Representational state transfer, a common approach for modern web services, see http://www.ics.uci.edu/~fielding/pubs/dissertation/-rest_arch_style.htm



Finally, in figure (C), the cloud service customer asks the cloud provider to send an alert when a specific condition is verified. This is called a “trigger” in CTP. In addition, the cloud provider will also log the details of this alert locally for future consultation by the customer.



Naturally, these examples leave out a lot of details that need to be addressed in CTP. Readers interested in “concrete” examples of CTP message exchanges may look at Appendix A.

3.2 Introducing the CTP data model

In order to implement the principles of the CTP API outlined in the previous paragraphs, it is necessary to define a data model that represents security attributes and related information through a set of objects that can be queried by CTP clients (e.g. cloud service customers).

Because CSPs offer very different kinds of services, they will use different approaches to describe their systems and the security they offer to cloud service customers. For example, an IaaS provider may present its service to the customer as a distinct set of “compute”, “storage” and “network” services. A SaaS provider might take a more abstract approach and simply present the service as a web service API available from a URL (much like CTP itself) with a set of security objectives associated with each functionality. The service presented to the customer is often a composition of several distinct services presented as one unique service associated with a clearly defined set of security objectives specified in a service interface and/or SLA.

In order to address such a wide range of use-cases, CTP takes an agnostic approach to the underlying technology and deployment model of a cloud, and adopts a data model that is based on the following 3 hierarchical components:

1. **Service-Views:** A service-view represents a customer’s view of a cloud service provided under the responsibility of a designated provider. In the most common case, from the perspective of a customer, there will be a 1-to-1 mapping between what is called a “cloud service” and a service-view. However in more complex use-cases it can be useful to break down a “cloud service” into a set of distinct service-views, each potentially with a distinct designated provider. A service-view encompasses a set of *assets*.
2. **Assets:** As [previously defined](#), an asset represents any physical or virtual component of a cloud information system that has a value to an organization and in particular the customer. A set of *attributes* is attached to an asset.
3. **Attributes:** As [previously defined](#), an attribute describes a characteristic of an asset that can be evaluated quantitatively or qualitatively. CTP aims to address security attributes, but the framework is generic enough to address any other kind of attribute, including notably performance indicators.

Cloud service customers can query a CTP server to learn about service-views that are exposed by the CSP. Next they can discover the cloud service assets that comprise each service-view and finally, the security attributes that are defined for each cloud service asset.

Note

CTP can also enable customers to learn about the list of cloud service providers that compose a cloud supply chain, as described in Appendix B, thereby facilitation compliance with data protection rules in some jurisdictions.

Associating a value with a security attribute requires the specification of a **measurement**, which is a process that evaluates an attribute concretely. By querying the CTP server, cloud service customers can learn about the measurements that are available for each attribute.

Each measurement is associated with a **measurement result**, which is the actual value or set of values representing the result of a measurement (i.e. numbers, strings, booleans, etc.).

In addition a measurement may be also associated with:

- **An objective:** A conditional expression that describes a constraint on possible measurement results. When the constraint is violated, the objective is considered as failed.
- **Triggers:** Like an objective, a trigger is also represented as a conditional expression on the possible measurement results associated with an attribute. When the conditional expression is true a message is sent to the cloud service customer and the CTP server logs an alert.

A measurement follows a certain number of concrete steps, collecting events, making computations, and providing results in specific units, which need to be interpreted accordingly. All these elements must be defined and documented in a **metric** specification that is accessible to the customer. Each measurement in CTP therefore references a metric, which is split in two parts as described in Section 4.2.5.

To summarize some of the key elements above, an *attribute* defines “what” we measure, a *measurement* (together with a *metric*) defines “how” we measure, and a *measurement result* represents the result of applying a *measurement* (how) to the *attribute* (what). An *objective* describes the range of measured values that are considered to reflect an agreed-upon level of security. A *trigger* describes the range of values that will generate an alert to the customer.

Example 3.1 A fictitious cloud provider

As an illustration, we can consider a fictitious example of a cloud provider “securebit.com”, which provides block storage containers that are optionally encrypted with a symmetric algorithm to provide confidentiality at the request of the customer. We could apply the CTP data model as follows:

- The block storage container service is a service-view operated by “securebit.com”.
- Each block storage unit is a distinct asset (with a distinct identifier).
- Each block storage unit has one security attribute called “confidentiality”.
- We define “symmetric-key-length” as a measurement² of the attribute “confidentiality”, where “symmetric-key-length” describes the process of evaluating the number of bits in the symmetric key used to encrypt the data on the storage unit (if no encryption is applied, “symmetric-key-length” is 0).

A customer might request all its storage blocks to be encrypted with AES-CBC using a key of 192 bits or more. In this case:

- The *objective* could be expressed as the following conditional expression³: “key_length>=192”
- Then the provider might establish and report that the *measurement result* of the actual key-length is 256 bits (which means that the objective is satisfied).

² This is a simplistic measurement process for confidentiality, used illustrative purposes only. This process would need to be more specifically detailed in an associated metric.

³ To be accurate this expression should be written as “value[0].key_length>=192” in CTP and use a better scale than bit length.

The elements briefly described so far provide a basic framework allowing cloud customers to get information about the security level of a cloud offering. However they are not sufficient to enable cloud customers to compare or rank different offerings. In order to achieve this, cloud service providers need to use CTP with a standardized set of asset categories, attributes, measurement and metrics. These standardized elements must be defined by relevant stakeholders and should derive from well-known compliance frameworks (e.g. CSA CCM, ISO 27k, PCI-DSS, etc.) or from other sector-specific requirements (see for example ongoing work in [\[ISO_19086\]](#)). In CTP, the adoption of these standards is signaled to the customer through the use of [service-classes](#) or [asset-classes](#).

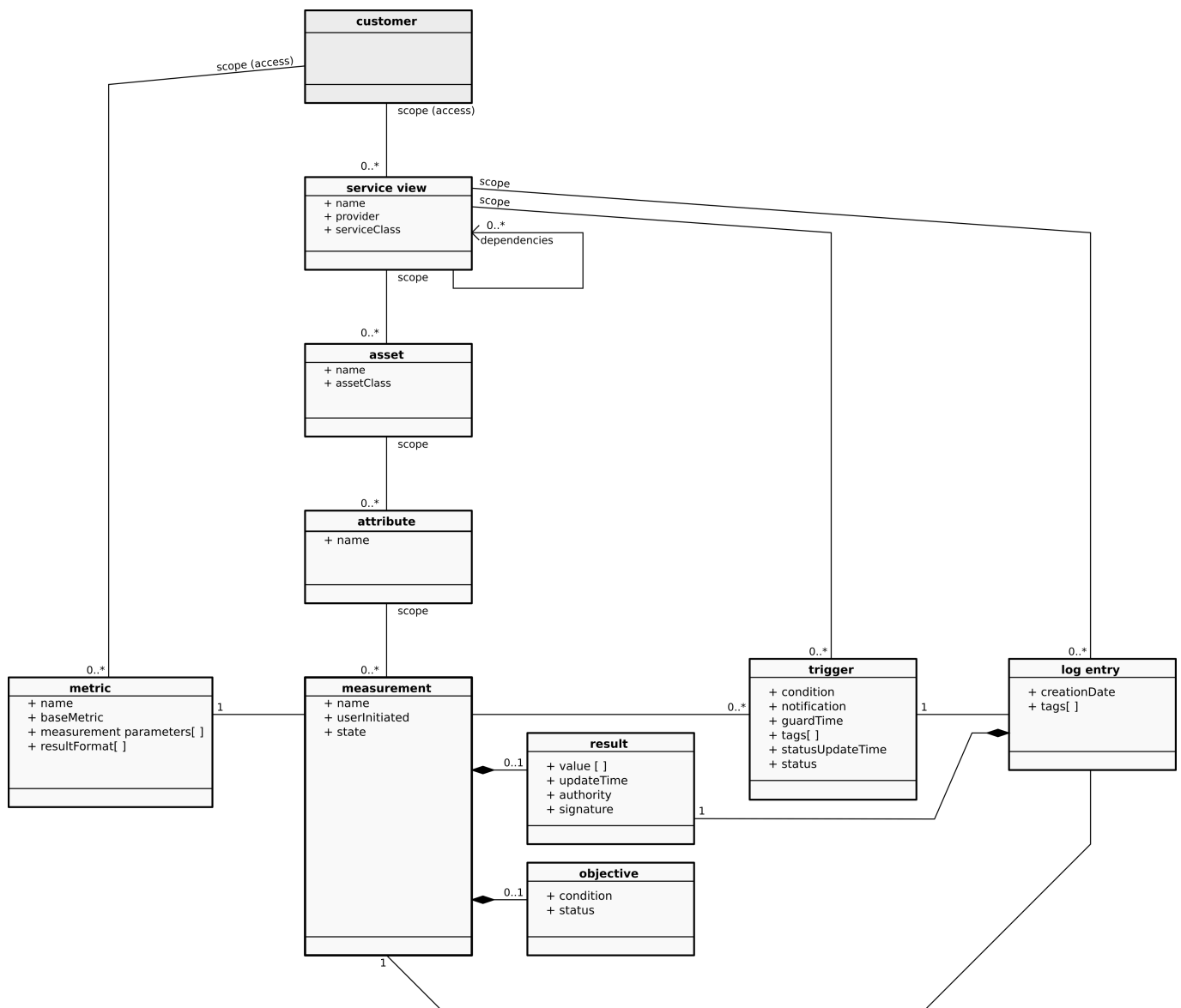
For the sake of simplicity, this brief introduction to the data model leaves out many details as well as a few important concepts (i.e. *log entries*). The following sections in this document describe the CTP data model and the related RESTful API in full detail.

Chapter 4

The CTP Data Model

This section describes the CTP data model, which is summarized in the following UML diagram.

The CTP data model



4.1 JSON representation

With the exception of the *customer*, all elements in the data model are translated into JSON resources that are made accessible by the CTP server. The *customer* is not explicitly represented as a JSON object in CTP but represents the fact that all resources in CTP are attached directly or indirectly to a specific customer.

Before detailing each component of the CTP data model along with their JSON encoding, it is necessary to introduce some conventions and notations that are used throughout the section.

4.1.1 API Resources URLs and identifiers

CTP exposes the following API resources to which HTTP methods can be applied:

- The base URL (the CTP entry point advertised to the client).
- Service-views.
- Collection of service-views.
- Collections of dependencies.
- Log entries.
- Collections of log entries.
- Assets.
- Collections of assets.
- Security attributes.
- Collections of security attributes
- Measurements (which include Measurement results and Objectives).
- Collections of measurements.
- Triggers
- Collection of triggers

4.1.2 Resource URLs

Each CTP API resource is accessible through a URL composed of two parts:

1. A base URL (the CTP API entry point)
2. A resource path (relative to the base URL)

In the rest of this document the Base URL or CTP API entry point will be referenced as `{CtpBase}`. While the `{CtpBase}` can be freely chosen by the CTP server, the CTP API resource URL must follow the rules defined in this specification.

A CTP API resource URL MUST be a fully qualified URL as defined in [\[RFC_3986\]](#).

Example 4.1 Decomposing a URL

The following example shows the decomposition of a fully qualified API resource URL:

API resource URL	<code>http://thecloud.com/serv/ctp/serviceUnit/02</code>
<code>{CtpBase}</code>	<code>http://thecloud.com/serv/ctp/</code>
Resource Path	<code>/serviceUnit/02</code>

Resource paths follow 4 possible structures:

1. `/t1` → this represents a collection of resources of type `t1` (e.g. `/serviceViews`).
2. `/t1/{id}` → this represents a single resource of type `t1` identified by the unique resource identifier `{id}` (e.g. `/serviceViews/1234`).
3. `/t1/{id}/t2` → this represents a collection of resources of type `t2` that are scoped by a particular resource identified `/t1/{id}` as previously described.
4. An empty resource path → By convention, accessing the empty resource path (`/`) will provide [general information](#) about the CTP server and protocol version.

Except for the CTP entry point `{CtpBase}`, resource paths SHALL NOT end with a terminal `/` (i.e. `/serviceViews` is a correct path, while `/serviceViews/` is incorrect and will typically result in a 404 http error).

4.1.3 Identifiers

In the remaining of this documentation, resource identifiers will be denoted as `{id}`.

All identifiers SHALL be represented with the alphabet defined in the “Base64 Encoding with URL and Filename Safe Alphabet” described in section 5 of [\[RFC_4648\]](#), with a maximum of 96 characters per identifier. CTP Server implementations MAY optionally further restrict identifiers to smaller alphabets representing a subset of the one defined in this specification, such as hexadecimal or decimal characters.

4.1.4 JSON notations used in the specification

This specification uses the standard JSON types: `string`, `number`, `object`, `array`, `boolean`, and `null`. To facilitate the description of the API, it also uses specific pseudo-types, which are simply restrictions on existing JSON types:

- `datetime`: a string that represent a UTC date and time as defined in the [\[RFC_3339\]](#).
- `URL`: A string that represents a universal resource locator (URL).
- `nested object`: A JSON object that appears within another JSON object or array.
- `list`: A JSON array where all elements are of the same type.

In this specification when describing the content of a *list*, we will describe the first element and add the symbol “...” to indicate that the *list* is an array of 0 or more of those elements.



Warning

Unless otherwise specified, lists may contain 0 elements.

4.1.5 Common properties of CTP JSON objects

A few JSON properties recurrently appear in the description of CTP resources. Firstly, all CTP JSON objects in CTP contain a property `self`. Secondly, except for *collections* and *log entries*, all CTP JSON objects contain the properties `name` and `annotation`. Finally, except for the *CTP entry point*, all CTP JSON objects have a `scope` property. These common properties are described more in detail hereafter.

4.1.5.1 The JSON "self" property

All CTP API calls with the GET, PUT or POST method return a JSON response, with a JSON object that contains the property `self`, which is a URL that self-references the CTP resource being accessed. A GET query to this URL refreshes the client's representation of the CTP API resource. In a response to a POST request, this URL represents the newly created CTP API resource URL. If the GET request includes a query string, it shall also be included in the self-reference provided in the response to the request.

Self-references MUST only be provided in response bodies, never in request bodies.

4.1.5.2 The JSON "scope" property

Except for the *CTP entry point*, all resources in CTP have a scope, as illustrated in the data model and as further detailed in Section 4.2:

- An *service-view* is scoped to the *CTP entry point* (i.e. `{CtpBase}`).
- An *asset* is scoped to a specific *service-view*.
- An *attribute* is scoped to a specific *asset*.
- A *measurement* is scoped to a specific *attribute*.
- A *log entry* or a *trigger* is scoped to a specific *service-view*.

This scoping also applies to *collections* (e.g. a collection of *measurements* is scoped to a specific *attribute*).

The scope of a JSON object is described by the property `scope`, which contains a CTP URL describing the scope of the current resource.

Scoping also allows the CTP server to implement access control: each customer has only access to a specific set of *service-views*, which in turn gives them access to a specific set of *assets*, *log entries* or *triggers*, and so and so forth.

4.1.5.3 the JSON "name" property

Except for *collections* and *log entries*, all CTP JSON resources contain a property `name`. This property is designed to hold an alternate identifier for the resource that is meaningful to the customer or provider. This alternate identifier can hold any valid string and is not bound by the format that is imposed on the CTP resource identifiers that appear in resource URLs as defined in Section 4.1.3. This alternate identifier can be used for example to represent the unique identifier of a cloud resource (e.g. a virtual machine id), a standardised name for an attribute or a metric, etc. The CTP server does not interpret this value but it can be used as a search criteria in a query to a collection, as detailed in Section 4.1.6.

4.1.5.4 the JSON "annotation" property

Except for *collections* and *log entries*, all CTP JSON resources contain a property `annotation`. This property is designed to hold human readable text for information purpose only, the content of which is not interpreted by CTP and may be left empty.

Example 4.2 Self links, alternate identifiers and annotations

The following truncated example shows a response to a CTP GET query to the URL `https://example.com/ctp/serviceViews/123abc`, illustrating the self links, alternate identifiers and annotations:

```
{
  "self": "https://example.com/ctp/serviceViews/123abc",
  "scope": "https://example.com/ctp/serviceViews/0001",
  "name": "disk:9708959262-9436488858-1118416481-4582989187",
  "annotation": "Cloud storage service",
  ...
}
```

4.1.6 Collections

As described in Section 4.1.1, API queries to URLs of the form `/t1` or `/t1/{id}/t2` target collections.

The JSON representation of a collection has the following format:

```
{
  "self": URL,
  "scope": URL,
  "collectionLength": number,
  "returnedLength": number,
  "collectionType": string,
  "collection": [
    {
      "link": URL,
      "name": string
    },
    ...
  ]
}
```

Property name	Type	Description
<code>self</code>	<i>URL</i>	Self-reference
<code>scope</code>	<i>URL</i>	Reference to the scope of the collection.
<code>collectionLength</code>	<i>number</i>	The total number of elements in the collection.
<code>returnedLength</code>	<i>number</i>	The total number of elements returned in the collection in the response, which will be equal to <code>collectionLength</code> or less, depending on the use of the query string parameters <code>page</code> and <code>items</code> , as described below.
<code>collectionType</code>	<i>string</i>	A string describing the type of resource listed in the collection.
<code>collection</code>	<i>list</i>	An ordered list representing links to items in the collections.
<code>collection[].link</code>	<i>URL</i>	A URL pointing to a CTP resource.
<code>collection[].name</code>	<i>string</i>	An <i>optional</i> string representing the name associated with the resource. This property is omitted when the name associated to the resource is empty.

The property `collectionLength` describes the size of the collection, while `returnedLength` describes the actual number of elements returned in the response.

Links in the collection **MUST** be ordered. Collections of log entries **MUST** be ordered by chronological order, by reference to their timestamp (see Section 4.2.8). For other types of collections, the choice of the ordering function applicable to links in the collection is left to the implementation. However, links returned in a collection **MUST** always be ordered the same way across queries. This ordering **MUST** allow internally for the implementation to index each link in increasing order from 0 to `collectionLength-1`.

A plain GET query without a query string returns the whole collection. In that case `collectionLength` will have the same value as `returnedLength`, as shown in the example below.

Example 4.3 A GET query to a collection.

A GET query to the collection identified by the URL:

`http://example.com/ctp/serviceViews/1/assets`

provides the following result describing a collection.

```
{
  "self": "http://example.com/ctp/serviceViews/1/assets",
  "scope": "http://example.com/ctp/serviceViews/1",
  "collectionLength": 5,
  "returnedLength": 5,
  "collectionType": "assets",
  "collection": [
    {
      "link": "https://example.com/ctp/assets/1",
```

```

    "name": "asset 1"
  },
  {
    "link": "https://example.com/ctp/assets/2",
    "name": "asset 2"
  },
  {
    "link": "https://example.com/ctp/assets/3",
    "name": "asset 3"
  },
  {
    "link": "https://example.com/ctp/assets/4",
    "name": "asset 4"
  },
  {
    "link": "https://example.com/ctp/assets/5",
    "name": "asset 5"
  }
]
}

```

CTP clients can however set a limit on the size of the returned collection using the query string parameters `items` and `page`. They may also restrict the result to resources that match a specific name (see Section 4.1.5.3).

Query parameter name	Type	Description
<code>page</code>	<i>number</i>	<i>Optional</i> page number, represented as an integer greater or equal to 0.
<code>items</code>	<i>number</i>	<i>Optional</i> number of elements returned in the collection, represented as an integer strictly greater than 0.
<code>name</code>	<i>string</i>	<i>Optional</i> string to restrict result to a specific alternate identifier. Only resource that match the value of <code>name</code> are returned.

Limiting the returned results with size and page The query parameters `page` and `items` **MUST** either be both specified or both omitted.

When these parameters are specified, a GET query to the collection **MUST** only return the existing links in the collection with indices between `page*items` and `page*items+items-1`, inclusive. In this case, the `returnedLength` returned in the response will report the actual number of links returned, which is always equal or less than the value of `items`.

The following 3 examples illustrate the use of different values for `page` and `items`.

Example 4.4 Pagination of collection results - first page

Given the collection presented in the example above, a GET query to

`http://example.com/ctp/serviceViews/1/assets?page=0&items=3`

would return:

```

{
  "self": "http://example.com/ctp/serviceViews/1/assets?page=0&items=3",
  "scope": "http://example.com/ctp/serviceViews/1",
  "collectionLength": 5,
  "returnedLength": 3,
  "collectionType": "assets",
  "collection": [
    {
      "link": "https://example.com/ctp/assets/1",
      "name": "asset 1"
    },
    {
      "link": "https://example.com/ctp/assets/2",
      "name": "asset 2"
    }
  ]
}

```

```
{
  {
    "link": "https://example.com/ctp/assets/3",
    "name": "asset 3"
  }
}
```

Example 4.5 Pagination of collection results - last page

Next, a GET query to

`http://example.com/ctp/serviceViews/1/assets?page=1&items=3`

would return:

```
{
  "self": "http://example.com/ctp/serviceViews/1/assets?page=1&items=3",
  "scope": "http://example.com/ctp/serviceViews/1",
  "collectionLength": 5,
  "returnedLength": 2,
  "collectionType": "assets",
  "collection": [
    {
      "link": "https://example.com/ctp/assets/4",
      "name": "asset 4"
    },
    {
      "link": "https://example.com/ctp/assets/5",
      "name": "asset 5"
    }
  ]
}
```

Example 4.6 Pagination of collection results - beyond last page

Finally, a GET query to

`http://example.com/ctp/serviceViews/1/assets?page=2&items=3`

would return:

```
{
  "self": "http://example.com/ctp/serviceViews/1/assets?page=2&items=3",
  "scope": "http://example.com/ctp/serviceViews/1",
  "collectionLength": 5,
  "returnedLength": 0,
  "collectionType": "assets",
  "collection": [
  ]
}
```

In the last example, the returned collection is empty because the combination of the query string parameters `page` and `items` points to indices beyond the end of the collection.

4.2 Resources

4.2.1 CTP entry point (CtpBase)

4.2.1.1 Description

A query to the CTP base URL (denoted {CtpBase}) provides general information about the CTP server to a CTP client (e.g. the CTP server version number).

The CTP entry point represents the “root” in the hierarchical scoping of all CTP JSON resources accessible to a customer.

The **CTP entry point** is defined by the following elements:

1. A *version*: the CTP API version number: "1.0" in this specification.
2. A *provider*: the identity of the provider of the CTP server.
3. A reference to a *collection of service-views*: this collection represents the service-views that are accessible to the CTP client based on the credentials it provides.

4.2.1.2 Encoding

```
{
  "self": URL,
  "name": string,
  "annotation": string,
  "version": string,
  "provider": string,
  "serviceViews": URL
}
```

Property name	Type	Description
self	URL	Self-reference to {CtpBase} Contrary to all other CTP resources, this URL ends with a / character.
name	string	An optional alternate identifier for the resource as detailed Section 4.1.5.3
annotation	string	A human readable description of the CTP server (may be empty).
version	string	The version of CTP in use (i.e. the string "1.0")
provider	string	The name of the entity providing the CTP server. This entity may be different from the one listed in service-views .
serviceViews	URL	A link referencing the CTP URI where further information can be obtained about the service-views that the client is authorized to access/query based on its credentials. The link is always equal to {CtpBase}/serviceViews

4.2.2 Service-views

4.2.2.1 Definition

A service-view represents a service offered to a specific customer under the responsibility of a single provider. This service is usually described in a SLA or service interface. A service-view encompasses a set of assets.

In CTP a **service-view** is defined by 8 elements:

1. A *scope*: A reference to the CTP entry point {CtpBase}.
2. A *name*: A optional name for the service unit.
3. A *provider*: The provider that has the contractual and legal responsibility for providing the functionalities offered by the service-view.
4. A *collection of dependencies*: The list of supplementary service-views that this service-view relies upon to operate. These dependent service-views may either be operated by the same provider or by a third party. This list may be empty.
5. A *collection of assets*: The set of assets associated with the service-view.

6. An optional *service class reference*: A reference to a standardized set of security attributes and measurement contexts this service unit strictly adheres to.
7. A *collection of logs*: The logs that are accessible in this service-view.
8. A *collection of triggers*: The triggers that are part of the service-view.

The following constraints apply:

- A service-view cannot reference itself in the dependency list.

A service-view associates groups of assets that are used together to provide a service to a customer under the legal responsibility of a specific cloud service provider. In the simplest case, a service-view can be considered as synonymous to “cloud service”. However, a “cloud service” can be broken down into 2 or more “service-views” to highlight independent components that share no common functionalities within a service (e.g. a IaaS platform might be broken down into two distinct “compute” and “storage” service-views). As such, the concept of service-views is meant to be sufficiently abstract to provide the flexibility to address a variety of use-cases and is meant to apply to any layer of the cloud stack from IaaS to SaaS.

By enforcing the association between a particular customer and a particular service-view (or set of service-views), the CTP server can implement an effective access control mechanism, since all other resources are scoped directly or indirectly to a service-view.

The optional reference to a service class is used to impose restrictions on the security attributes associated with all the assets of a service-view, for the purpose of interoperability. By default, when no service class is referenced, the assets of a service-view can be associated with an arbitrary set of security attributes, using any measurement. However, when a service-view specifically references a *service class*, the list of usable attributes and their measurement contexts must strictly follow the specifications of the referenced *service class*. Cloud providers SHOULD adopt service-classes or asset-classes for the benefit of customers (see Section 4.3).

4.2.2.2 Encoding

```
{
  "self": URL,
  "scope": URL,
  "name": string,
  "annotation": string,
  "provider": string,
  "dependencies": URL,
  "assets": URL,
  "serviceClass": URL
  "logs": URL,
  "triggers": URL
}
```

Property name	Type	Description
self	URL	Self-reference to {CtpBase}/serviceViews/{id}
scope	URL	Reference to the CTP entry point that exposes this service-view, which is a URL of the form {CtpBase}
name	string	An optional alternate identifier for the resource as detailed Section 4.1.5.3
annotation	string	a human readable description of this service-view (may be empty).
provider	string	The name of the provider that is legally responsible for providing the service-view. This provider might be different from Section 5.2.1 the one offering the CTP service itself>>.
dependencies	URL	A link providing a location where information can be found regarding the dependencies of this service-view. This link is of the form {CtpBase}/serviceViews/{id}/dependencies.
assets	URL	A link providing a location where information can be found regarding all the resources of this service-view. This link is of the form {CtpBase}/serviceViews/{id}/assets.

Property name	Type	Description
serviceClass	URL	The service class applicable to all assets and attributes of this service-view. Must be set to an empty string if no service class applies.
logs	URL	A link providing a location where information can be found regarding all the logs that are stored by the CTP server for this service-view, as the results of triggers. This link is of the form {CtpBase}/serviceViews/{id}/logs.
triggers	URL	A link providing a location where information can be found regarding all the triggers that have been defined in the CTP server for this service-view. This link is of the form {CtpBase}/serviceViews/{id}/triggers.

4.2.3 Assets

4.2.3.1 Definition

An **asset** is used to represent any tangible or intangible element of a cloud information system, such as for example simple API URLs, storage, processor cores or compute instances, databases, full blown platforms, etc. A set of attributes is attached to an asset.

In CTP, an **asset** is defined by 4 elements:

1. A *scope*: A reference to the service-view this resource belongs to.
2. A *name*: A unique name describing the resource in a way that is unambiguous for the customer in the context of the service offered (e.g. “API.GetBlock()”, “storage-0458-fb04”, “http://example.com/login”).
3. A *collection of applicable attributes*: The set of (security) attributes for which measurements can be produced for this particular asset.
4. An optional *asset class reference*: A reference to a standardized set of security attributes and measurement contexts this asset strictly adheres to.

Service resources are the virtual or physical elements to which a set of security attributes apply.

Following the same principle as “service classes” in service-views, the optional reference to an asset class is used to impose restrictions on the security attributes associated with an asset of a service-view, for the purpose of interoperability. By default, when (1) no asset class is referenced and (2) when no service class is referenced at service-view level, the assets of a service-view can be associated with an arbitrary set of security attributes, using any measurement. However, when an asset specifically references an asset *class*, the list of usable attributes and their measurement contexts must strictly follow the specifications of the referenced *asset class*. Cloud providers **SHOULD** adopt service-classes or asset-classes for the benefit of customers (see Section 4.3).

For simplicity and flexibility, CTP does not impose any constraints on the definition of what constitutes an asset. In some cases an asset might describe a software service and in some other cases an instance produced by such a service. For example, both an IaaS API that creates virtual machines and the produced virtual machines could be considered as “assets” with security attributes, depending on the context and the needs of the provider and his customers. To avoid ambiguities, it is therefore important to name assets unambiguously and to convey their scope clearly to cloud service customers.

4.2.3.2 Encoding

```
{
  "self": URL,
  "scope": URL,
  "name": string,
  "annotation": string,
  "attributes": URL,
```

```

    "assetClass": URL
}

```

Property name	Type	Description
self	URL	Self-reference to {CtpBase}/assets/{id}
scope	URL	A reference to the service-view this asset belongs to.
name	string	The name of the cloud resource (e.g. "Storage", "get_block()", "http://example.com/service/").
annotation	string	A human readable description of this service resource (may be empty).
attributes	URL	A reference to the collection of security attributes defined for this service resource.
assetClass	URL	The service class applicable to the security attributes of this asset. Must be set to the empty string if undefined.

4.2.4 Attributes

4.2.4.1 Definition

A security attribute or more simply an [attribute](#) is used to represent characteristic of an asset that can be evaluated quantitatively or qualitatively and is identified with a distinct name (e.g. “availability”, “incident response”, etc.). Associating a value with a security attribute requires the specification of a measurement.

In CTP an **attribute**¹ is defined by 3 elements:

1. *A scope*: A reference to the asset to which this attribute applies.
2. *A name*: A generic name of the attribute.
3. *A collection of available measurements*: The list of possible measurements that can be used to evaluate this attribute.

The following constraints apply:

- Each attribute is associated with only one asset (scope).

In most cases, an attribute will have only one measurement: there will be only one defined process to quantify the attribute. However, for some attributes, it makes sense to have several measurements (e.g. to distinguish the monthly evaluation of availability from the hourly evaluation of availability).

4.2.4.2 Encoding

```

{
    "self": URL,
    "scope": URL,
    "name": string,
    "annotation": string,
    "measurements": URL
}

```

Property name	Type	Description
self	URL	Self-reference to {CtpBase}/attributes/{id}
scope	URL	A reference to the asset this attribute applies to.
annotation	string	A human readable description of this security attribute (may be empty).
name	string	The optional name of the security attribute (see Section 4.1.5.3).
measurements	URL	A link of the form {CtpBase}/attributes/{id}/measurements.

¹ Note, to address a point of potential confusion: in order to be able to make statements about “security attributes”, we include “attribute” as a class in our model. As a consequence, if we adopt UML terminology, the 3 elements that define a security attribute can therefore be considered as “UML attributes” of “security attributes”.

4.2.5 Metrics

4.2.5.1 Definition

A [metric](#) describes a standard of measurement, which will be referenced in [measurements](#). A metric is typically specified in an external document that describes in human readable form *the conditions and the rules for performing the measurement and for understanding the results of a measurement*. This external document is called a “reference metric” in CTP and needs to be completed with concrete machine-readable information to be useful in CTP. In particular, while the reference metric MAY describe possible measurement parameters, their [values](#) MUST be determined for the metric to be applied concretely.

In CTP a **metric** is defined by 4 elements:

1. *A name*: the name of the metric, which highlights its purpose and distinguishes it from other metrics.
2. *A link to a reference metric*: an external document that describes in human readable form the metric (i.e. *the conditions and the rules for performing the measurement and for understanding the results of a measurement*).
3. *A list of measurement parameter values*: a list of parameters that are used to quantify the value of a security attribute. This list may be empty if no parameter is needed in the measurement. Each parameter in the list is composed of three elements:
 - a. A measurement parameter name.
 - b. A measurement parameter type.
 - c. A measurement parameter value.
4. *A measurement result format*: A description of the type and format of the result of the measurement that is conducted according to the reference metric and the measurement parameter values. Measurement results are represented as a [table of values](#) and the measurement result format included in the metric is simply a description of the columns of that table, as a list where each item is composed of:
 - a. A column name.
 - b. A column type.

The following constraints apply:

- Two metrics SHOULD share the same name if and only if they use the same reference metric, the same measurement parameters and the same measurement result format.

We highlight that the “reference metric” is not explicitly defined as an object in the CTP data model: it is only defined as an external link (using a URL). This allows to reference metrics from external public standards or other documents made available by cloud providers.

The list of [measurement parameter values](#) is used to further specify or restrict how the measurement of a security attribute is conducted, as described in the reference metric. For example, when measuring availability of a service, we may wish to define the maximum delay allowed for a service request to be executed by the service provider before the request can be considered as failed. The reference metric will describe this parameter and give it a name but may not necessarily give it a prescriptive value. However, in order to apply a metric in a concrete case, this value must be specified: this is the role of the measurement parameter values in the metric. Other measurement parameters that could be used for a metric related to availability include: the measurement period (e.g. a week, a month), specific constraints (e.g. business hour specific thresholds), and sample sizes, etc. Some of these measurement parameter values may be already defined in the externally referenced reference metric but should be repeated in the measurement for information purposes.

4.2.5.2 Encoding

```

{
  "self": URL,
  "name": string,
  "annotation": string,
  "baseMetric": URL,
  "measurementParameters": [
    {
      "name": string,
      "type": string,
      "value": string or number or boolean,
    },
    ...
  ],
  "resultFormat": [
    {
      "name": string,
      "type": string,
    },
    ...
  ]
}

```

Property name	Type	Description
self	<i>URL</i>	Self-reference {CtpBase}/metrics
name	<i>string</i>	A standardized name for the metric, as specified in the document provided in referenceMetric (see Section 4.1.5.3).
annotation	<i>string</i>	A human readable description of this metric (may be empty).
baseMetric	<i>URL</i>	External link to the metric implemented by this measurement. This is a human readable document standardizing the metric.
measurementParameters	<i>list</i>	A list of measurement parameters
measurementParameters[].name	<i>string</i>	The name of the measurement parameter.
measurementParameters[].type	<i>string</i>	The type of the measurement parameter, which is either “boolean”, “number” or “string”.
measurementParameters[].value	<i>string or number or boolean</i>	The value of the measurement parameter, encoded either as a string, number or boolean (depending on the type).
resultFormat	<i>list</i>	This element provides a description of the format of the result of the measurement. Measurement results are represented as tables and this element provides the list of names of the columns that compose that table as well as their type.
resultFormat[].name	<i>string</i>	The name of the table column (e.g. “percentage”).
resultFormat[].type	<i>string</i>	A description of the type of the items in the column, which is either “boolean”, “number” or “string”.

4.2.6 Measurements

4.2.6.1 Definition

A [measurement](#) describes how a specific attribute is evaluated, using a specific [metric](#).

In CTP a **measurement** is defined by 9 elements:

1. A *scope*: A reference to the attribute to which this measurement applies.
2. A *name*: The optional name of this measurement (see Section 4.1.5.3).

3. *A reference to a metric*: A reference to the metric that is implemented by this measurement in order to quantify or qualify an attribute.
4. *A measurement result*: see [below](#). This element may be null if a result is not available.
5. *A measurement result signature*: see [below](#). This optional element represents a JSON Web Signature on the measurement result, following [\[RFC_7515\]](#).
6. *An objective*: See [below](#). This element may be null if no objective is defined.
7. *A trigger creation indicator*: An element that indicates whether or not the CTP client is allowed to create triggers for this measurement.
8. *A user-activation indicator*: An element that indicates whether or not the CTP client is responsible for activating the measurement.
9. *A state*: An indicator describing the current monitoring state of the security attribute through this measurement, which can hold one of the following 3 values:
 - a. “activated”: This specific measurement is activated and the security attribute is monitored accordingly. The attribute can be queried for measurement results and/or activate triggers, depending on the capabilities of the measurement.
 - b. “deactivated”: This specific measurement is not activated but the cloud provider has the capability to do so. The cloud service customer must first activate this measurement with the CTP API before the cloud provider is able to provide measurement results and/or triggers about the related attribute.
 - c. “pending”: This transitional state indicates that the cloud provider has not enough data yet to provide measurement results and/or activate triggers about the attribute, with this specific measurement. This is notably intended for cases where a measurement has been activated by the cloud service customer but is not “ready” to be queried yet for measurement results due to lack of sufficient monitoring data for example.

By design, a cloud service provider implementing CTP has full control over the use of *measurement*, and can apply the following restrictions:

- The CTP server can set an objective for a measurement or not.
- The CTP server can allow triggers for a measurement or not.
- The CTP server can provide CTP clients with the capacity to activate a measurement or not.

However, if the asset to which the measured attribute is attached adheres to an asset-class or if the service-view adheres to a service-class, the cloud service provider SHALL implement the capabilities defined in the external *asset class* or *service class* specification (see Section 4.3).

In some cases, a cloud provider may not wish to conduct the measurement of a security attribute of an asset on a continuous basis, but will only initiate a measurement at the request of a cloud service customer. In CTP these measurements are distinguished from others through an “user-activation” indicator that is set to “true”. Measurements that are “user-activated” are by default in “deactivated” state, which means that the related attribute is not evaluated with the specified metric. Cloud service customers can use the CTP API to activate these measurements (see Section 5.2.11). Optionally, the measurement can then be in “pending” state before reaching “activated” state, while enough data is gathered by the monitoring infrastructure for example. Once in “activated” state, these measurements can produce measurement results and activate triggers. These “user-activated” measurements are used in situations where the intervention of the cloud service customer is meaningful. This would be for example the case for a measurement that requires a vulnerability scan, which could be requested by the cloud service customer occasionally instead of being constantly conducted the provider.

The measurement results and objectives are detailed in the following subsections.

Measurement results

A measurement result appears as part of measurements and log entries.

A measurement result reflects the state of a resource at a specific time.

Example 4.7 A measurement result

Consider the following example of a measurement result: “For our storage service, the percentage of uptime was 99.92% on December 31st, 2012, for a period of one month.”

A **measurement result** is defined by 3 elements:

1. A *value*: The actual representation of the measurement result in a table format. The number of columns and rows of that table depends on the referenced measurement. In particular, the number of columns in the table corresponds to the number of columns defined in the *measurement result format* of the underlying *metric*.
2. A *timestamp*: The date and time the measurement result was produced.
3. An *authority-identifier*: An optional identifier of the entity producing the measurement result. This element is optional and is mainly intended to be used when a signature is added.

In the general case, the entity producing the measurement result will be the cloud provider, but other alternatives could be envisaged including trusted third parties or TPM modules. In these alternative cases the entity **SHOULD** be identified in the measurement result.

Additionally, the authority may cryptographically sign the content of the measurement result, for authentication and integrity purposes. The signature applies to the full content of the measurement result, using the JSON Web Signature (JWS) as defined in [\[RFC_7515\]](#). The string representing the signature in JWS Compact Serialization format is provided in another property of the measurement. Providing a signature is left to the choice of the cloud service provider, unless otherwise dictated by an existing [service-class or asset-class](#).

Objectives

An objective is an optional part of a measurement.

An **objective** describes an expected result with regards to a security attribute. This expected results is expressed as a constraint on the measurement results of a security **attribute**. It describes the security level that the cloud provider is committed to achieve, as usually described to cloud service customers in a service interface and/or SLA.

Example 4.8 An objective

For example, as an objective, a cloud provider might commit to restrict the *location of data processing* in a resource to the set of 3 countries composed of Australia, Belgium and Canada.

An **objective** is defined by 2 elements:

1. A *condition*: A representation of an objective as a boolean expression describing a constraint on the expected result of the measurement of the underlying security attribute.
2. A *status*: A value describing the result of the evaluation of the condition, which can take one of the following 3 values:
 - true: if the condition evaluates to true.
 - false: if the condition evaluates to false.
 - error: if an error occurs in the evaluation of the condition.

In CTP, *conditions* in objectives are specified using conditional expressions that are described in a small language called **CTP-Script** that is similar to JavaScript Expressions. The result of a measurement is expressed as an array variable that can be used in a CTPScript expression to describe a condition. If this CTPScript expression does not evaluate correctly due to an error, then the status of the objective is set to ‘error’. If the CTPScript expression evaluates correctly to the boolean ‘true’, then the objective is verified to be true, otherwise it is false.

Example 4.9 An objective in CTPScript

The following CTPScript expression might describe an objective related to cryptographic key length used in a SSL/TLS endpoint:

```
value[0].tls_symmetric_key_length >= 128
```

Some objectives can take a more complex form. Consider a measurement that describes distinguishes availability between “business hours” and “other hours” The following CTPScript expression might describe another objective related to availability:

```
value[0].business_hours_uptime > 99.5 && value[0].other_hours_uptime > 98.0
```

Note that in this example, the actual definition of what constitutes “business hours” would likely be part of the measurement parameters of the [metric](#).

4.2.6.2 Encoding

```
{
  "self": URL,
  "scope": URL,
  "name": string,
  "annotation": string,
  "metric": URL,
  "result": {
    "value": [
      nested object, ...
    ],
    "dateTime": datetime,
    "authorityId": string,
    "signature": string,
  },
  "objective": {
    "condition": string,
    "status": string,           // either "true", "false" or "error"
    "statusUpdateTime": datetime
  },
  "trigger": string,
  "userInitiated": boolean,
  "state": string             // either "deactivated", "pending" or "activated"
}
```

Property name	Type	Description
self	URL	Self-reference to {CtpBase}/measurements/{id}
scope	URL	A reference to the attribute this measurement applies to.
annotation	string	A human readable description of this measurement (may be empty).
metric	URL	A link to the metric implemented by this measurement. This link is of the form: {CtpBase}/metrics/{id}
result	nested object or null	If the measurement state is “pending” or “deactivated”, this object is null (see the state property below). Otherwise this object represents the current measurement result value, following the format described in the linked metric .
result.value	List	A list of nested objects, each representing a row in the table representing the measurement result, as defined in the metric .
result.value[]	nested object	A set of (key, value) pairs describing a single row in the table representing the measurement result.
result.updateTime	datetime	A UTC timestamp of the measurement result as defined in [RFC_3339] .
result.authorityId	string	A string representing the identity of the entity that produced the measurement result. This property MAY be null if the entity is the same as the service provider or if it is not known. This property MUST be non-empty if the property resultSignature is non-empty.

Property name	Type	Description
result.signature	<i>string or null</i>	<p>A string representing the JWS Compact Serialization of a JSON Web Signature of a serialization of the object representing the content of the result property, as defined in [RFC_7515]. This field MAY be an empty string if no signature is provided. Implementations MUST support signatures provided with the algorithm “RSASSA-PKCS1-v1_5 SHA-256”. Implementations MAY support other signature standards. The serialization of the result object is performed as follows:</p> <ul style="list-style-type: none"> • Let P represent the JSON encoding of the result object, excluding the signature property, but including all other previous fields, without any change in encoding. • Remove all non-significant white space characters around the JSON tokens of P: tabulation (U+0009), line feed (U+000A), carriage return (U+000D), and space (U+0020) • Return the resulting string P, which must start with a left curly bracket (U+007B) and end right curly bracket (U+007D) <p>The string P then represents the value of the payload used in the JSON Web Signature.</p>
objective	<i>nested object or null</i>	If this property is not <i>null</i> , then an objective has been defined for the measurement result, as specified in the following 2 properties of the objective object.
objective.condition	<i>string</i>	A CTPScript expression . This JSON string must be properly escaped
objective.status	<i>boolean</i>	The result of the evaluation of the CTPScript condition based on the current measurement result.
createTrigger	<i>URL</i>	<p>This optional link indicates where triggers for this measurement can be created with a POST request or listed with a GET request. The link is of the form {CtpBase}/serviceViews/{id}/triggers. If the link is an empty string, this indicates that the measurement cannot be used in a trigger.</p>
userInitiated	<i>boolean</i>	<p>If this field is true, then the CTP client can request to initiate the measurement through a specific API call. This will cause the state described hereafter to change accordingly. If this field is false, this indicates that the client cannot initiate the measurement.</p>
state	<i>string</i>	<p>This element represents the state of the attribute and can take 3 values only: "activated" if the value of this security attribute is measurable, "pending" if the value of this security attribute is not yet measurable, due to missing or incomplete data, and "deactivated" if the field requires to be user-activated. A measurement that is user-activated is initially in a “deactivated” state but can be activated with a specific API call. A measurement that is not user-activated must always be in the "activated" or "pending" state.</p>

4.2.7 Triggers

4.2.7.1 Definition

In order to enable cloud service customers to receive notifications when specific conditions are met, CTP allows them to define triggers. Like an objective, a trigger is a conditional expression on the measured value of a security attribute. However, contrary to objectives, triggers are created by the CTP client with a specific API call. When a trigger conditional expression is true, a

message is sent to the cloud service customer via XMPP [RFC_6120] and an alert is logged by the cloud provider. For example, a cloud client might request to be notified if the *recovery time actual* (RTA) of a resource exceeds a certain duration threshold.

A **trigger** is defined by 9 elements:

1. *A scope*: A reference to the service-view in which this trigger is defined.
2. *A name*: An optional name for this trigger.
3. *A reference measurement*: The measurement to which the trigger applies.
4. *A trigger condition*: A representation of a trigger as a conditional expression that applies to the results of the measurement of the underlying security attribute. When the condition is evaluated, [alerting rules](#) defined in this document are applied to decide whether an alert is generated or not.
5. *A XMPP notification URI*: This optional URI describes an XMPP endpoint that will receive a notification in case of an alert. Note that future versions of the CTP API may define other notification mechanisms than XMPP, which will be distinguished by a different URI format.
6. *A guard time*: The minimal time, expressed in seconds, that should elapse between two alerts generated from the same trigger. This is intended to avoid alert overflows. This delay does not apply to alerts that result from an error in the evaluation of the condition: such alerts are only generated once regardless of how long the error persists.
7. *A set of tags*: a set of keywords that will be added in the log entries produced as a result of this trigger. Tags can also be later used to sort log entries (a tag can be used for example to specify a severity level).
8. *A status*: a value that indicates the result of the last evaluation of the trigger condition, which can take one of the following 3 values:
 - true: if the condition evaluates to true.
 - false: if the condition evaluates to false.
 - error: if an error occurs in the evaluation of the condition.
9. *A status timestamp*: a timestamp representing the point in time when the trigger was last evaluated.

In CTP, *conditions* in triggers are specified using conditional expressions that are described in a small language called [CTPScript](#) that is similar to JavaScript expressions. The result of a measurement is expressed as an array variable that can be used in a CTPScript expression to describe a condition. If this CTPScript expression evaluates to the boolean ‘true’, then the [trigger evaluation rules](#) defined in this document are applied to decide whether an alert is generated or not (taking the guard time into consideration), followed by a XMPP notification sent to the cloud service customer.

Example 4.10 A trigger condition in CTPScript

The following CTPScript expression shows an example of a trigger condition related to the redundancy level of a system component:

```
value[0].redundancy <=2
```

In addition, if the CTP condition fails to evaluate correctly, due for example to an error in the CTPScript expression, then the [trigger evaluation rules](#) defined in this document are also applied to decide whether an alert is generated or not, followed by a XMPP notification sent to the cloud service customer.

There are therefore two types of alerts:

- Normal alerts: alerts that are generated when the trigger condition evaluates to ‘true’ in CTPScript.
- Error alerts: alerts that are generated when an error occurs in the evaluation of the CTPScript condition.

Each one of these types of alerts corresponds to a [different type of log](#). The XMPP notification that is sent as a result of an alert MUST contain a copy of the log entry that is created as a result of the trigger. Similarly, in case of an error, an XMPP message containing the log entry describing the error MUST be sent.

Providers implementing the CTP API MAY use their own out-of-band mechanism to validate the XMPP endpoint provided by the cloud service customer if needed. Such a validation mechanism is out of scope of this document.

4.2.7.2 Encoding

```
{
  "self": URL,
  "scope": URL,
  "name": string,
  "annotation": string,
  "measurement": URL,
  "condition": string,
  "notification": URL,
  "guardTime": number,
  "tags": [
    string, ...
  ],
  "status": string,           // either "true", "false" or "error"
  "statusUpdateTime": datetime
}
```

Property name	Type	Description
self	<i>URL</i>	Self-reference to {CtpBase}/triggers/{id}
scope	<i>URL</i>	Reference to the service-view that exposes this trigger, which is a URL of the form {CtpBase}/serviceViews/{id}
name	<i>string</i>	An optional name for this trigger (see <<name- ↔ property>>).
annotation	<i>string</i>	A human readable description of the trigger (may be empty).
measurement	<i>URL</i>	Reference to the measurement to which this trigger applies.
condition	<i>string</i>	A CTPScript expression . This JSON string must be properly escaped.
notification	<i>URL</i>	Description of the XMPP URI that will be used to notify the cloud service customer, following the structure defined in RFC 5122. Implementation must recognize XMPP URI of the form xmpp:<node>@<server><path> where <node> represent an XMPP node name, <server> an XMPP server name and <path> a possibly empty XMPP path descriptor.
guardTime	<i>number</i>	The amount of time in seconds that must elapse between 2 consecutive alerts resulting from this specific trigger. This is to avoid potential notification and log overflow in case where a condition remains true for a long time, despite updated measurement results.
tags	<i>list</i>	A list of string tags.
tags[]	<i>string</i>	A keyword that will be recorded in the log entry generated as a result of the trigger. This tag can later be used to sort log entries.
status	<i>string</i>	The result of the last evaluation of the trigger. This string can be either "true", "false" or "error".
statusUpdateTime	<i>datetime</i>	A UTC timestamp (as defined in RFC_3339) representing the point in time when the trigger condition was last evaluated.

4.2.8 Log entries

4.2.8.1 Definition

Log entries are generated by triggers and can have two different forms:

Form 1, Normal log entry:

1. *A scope*: A reference to the service-view to which this log entry belongs.
2. *A trigger reference*: A reference to the trigger that generated this log entry.
3. *A timestamp*: The date and time of the alert.
4. *A measurement result*: A measurement result corresponding to the value that verified the trigger condition (see the [encoding of measurements](#)).
5. *A set of tags*: A set of tags that is associated with the entry, as specified in the referenced trigger.

Form 2, Error log entry:

1. *A scope*: A reference to the service-view to which this log entry belongs.
2. *A trigger reference*: A reference to the trigger that generated this log entry.
3. *A timestamp*: The date and time of the error.
4. *An error message*: A message describing an error in the evaluation of a trigger condition.
5. *A set of tags*: A set of tags that contains only one item, the tag “error”.

4.2.8.2 Encoding of normal log entries

```
{
  "self": URL,
  "trigger": URL,
  "creationTime": datetime,
  "result": {
    "value": [
      nested object, ...
    ],
    "updateTime": datetime,
    "authorityId": string,
    "signature": string
  }
  "tags": [
    string, ...
  ]
}
```

Property name	Type	Description
self	<i>URL</i>	Self-reference to {CtpBase}/logs/{id}
trigger	<i>URL</i>	A reference to the trigger that generated this log entry, which is of the form {CtpBase}/triggers/{id}
creationTime	<i>datetime</i>	A timestamp for the creation of the log entry.
result	<i>nested object</i>	The measurement result that made the trigger condition true. This element is defined exactly as the <code>result</code> property in a encoded measurement .
result.value	<i>list</i>	See the encoding of measurements for details.
result.updateTime	<i>datetime</i>	See the encoding of measurements for details.
result.authorityId	<i>string</i>	See the encoding of measurements for details.
result.signature	<i>string</i>	See the encoding of measurements for details.
tags	<i>list</i>	A list of tags associated with the trigger. May be an empty list if no tag is set.
tags[]	<i>string</i>	A tag that was associated with the trigger.

4.2.8.3 Encoding of error log entries

```
{
  "self": URL,
  "trigger": URL,
  "creationTime": datetime,
  "error": string,
  "tags": [
    "error"
  ]
}
```

Property name	Type	Description
self	<i>URL</i>	Self-reference to {CtpBase}/logs/{id}
trigger	<i>URL</i>	A reference to the trigger that generated this log entry, which is of the form {CtpBase}/triggers/{id}
creationTime	<i>datetime</i>	A timestamp for the creation of the log entry.
error	<i>string</i>	A description of the error in the CTPScript evaluated in the trigger.
tags	<i>list</i>	A list of tags associated with the trigger. In the case of an error, this list contains one element.
tags[]	<i>string</i>	The string “error”.

4.3 Service classes and asset classes

Where feasible, cloud service providers SHOULD adopt service classes or asset classes, as defined in this section.

From a purely modeling perspective, CTP leaves a lot of flexibility to cloud service providers:

- Assets can be associated with an arbitrary sets of security attributes;
- Security attributes can be quantified with different measurements, using different metrics with different measurement parameter values;
- The provider can choose whether measurements can have objectives, triggers, or can be user-activated.
- The provider can choose whether measurement results are cryptographically signed or not.

In real use-cases, this flexibility can have some significant drawbacks for customers because:

- it can make it more difficult to assess the data provided by the cloud service provider if it is structured in an unfamiliar way
- it can become very hard to compare the security level of the cloud service provider with the customer’s requirements or baseline,
- it can make the comparison between offerings from different cloud service providers impossible.

One first step to address this issue is for cloud service providers to adopt standardized metrics and reference them accordingly with a well-known URL (i.e. using the `referenceMetric` property of the metric resource). While necessary, standard metrics are often not enough. For the purpose of comparing security attributes of a cloud service either with a customer-defined baseline or with another competing cloud service, it is useful to be much less flexible and to explicitly agree to use a specific set of security attributes, asset categories, metrics and measurement methods with detailed restrictions. To this end, CTP defines **service classes** and **asset classes**. Both of these concepts define a set of restrictions applicable to security attributes, metrics, measurements and assets. They differ by their scope: while an asset class applies a restriction to a particular asset, a service class applies to a service-view and describes a restriction on all assets within that service-view.

An asset class SHALL specify:

- The type of asset targeted by the class of restrictions,
- Asset naming conventions,
- Metrics that can be used to create measurements on assets. The specification of metrics includes at least:
 - Metric names,
 - The specification of (human readable) reference metrics,
 - The specification of metric measurement parameters,
 - The specification of measurement result formats
- Security attribute names and their applicable measurements/metrics,
- The mandatory capabilities of measurements that apply a particular metric (e.g. if the measurement must offer an objective, triggers, or be user-activated).
- Whether cryptographic signatures are mandatory or not for measurement results.

A service class SHALL specify:

- The type of service-view target by the class of restriction,
- A series of asset-classes that apply to all the assets defined in the type of service-view.

Service classes and **asset classes** are referenced in CTP by a standardized URL, which **MUST** point to the document that defines the previously outline information.

For example, we could imagine the definition of “asset class” for IaaS *storage* assets (or similarly *compute* assets), where a predefined set of attributes is standardized for objectives and measurement results, using a well-defined set of industry wide agreed-upon metrics (i.e. metrics, measurement result formats and measurement parameters). Such a service class could for example require “availability” to be measured with a measurement period of one month, the use of an uptime-based measurement of “availability” with a “sample window size” of 2 minutes, etc. A cloud provider that offers a storage resource API that adheres to this service class would then reference the asset class URL in the CTP representation of the asset: all security attributes (and related measurements) that pertain to that asset would be expected to follow the rules defined in the referenced asset class.

The definition of the exact content of these “service classes” or “asset classes” is sector specific and is beyond the scope of this CTP Model and API specification.

4.4 Relating measurement results and metrics

In the CTP API, measurement results are presented as part of the result returned when [querying a measurement](#). This result is an object that contains two key properties: (1) `value`, which is a table describing the actual value of the measurement result and (2) `updateTime`, which is a timestamp of the measurement result. The format of the table describing the value is not fixed: it depends on the metric referenced in the measurement. The referenced metric contains a field `resultFormat` that describes the format of the result, as illustrated on the [figure](#) below.

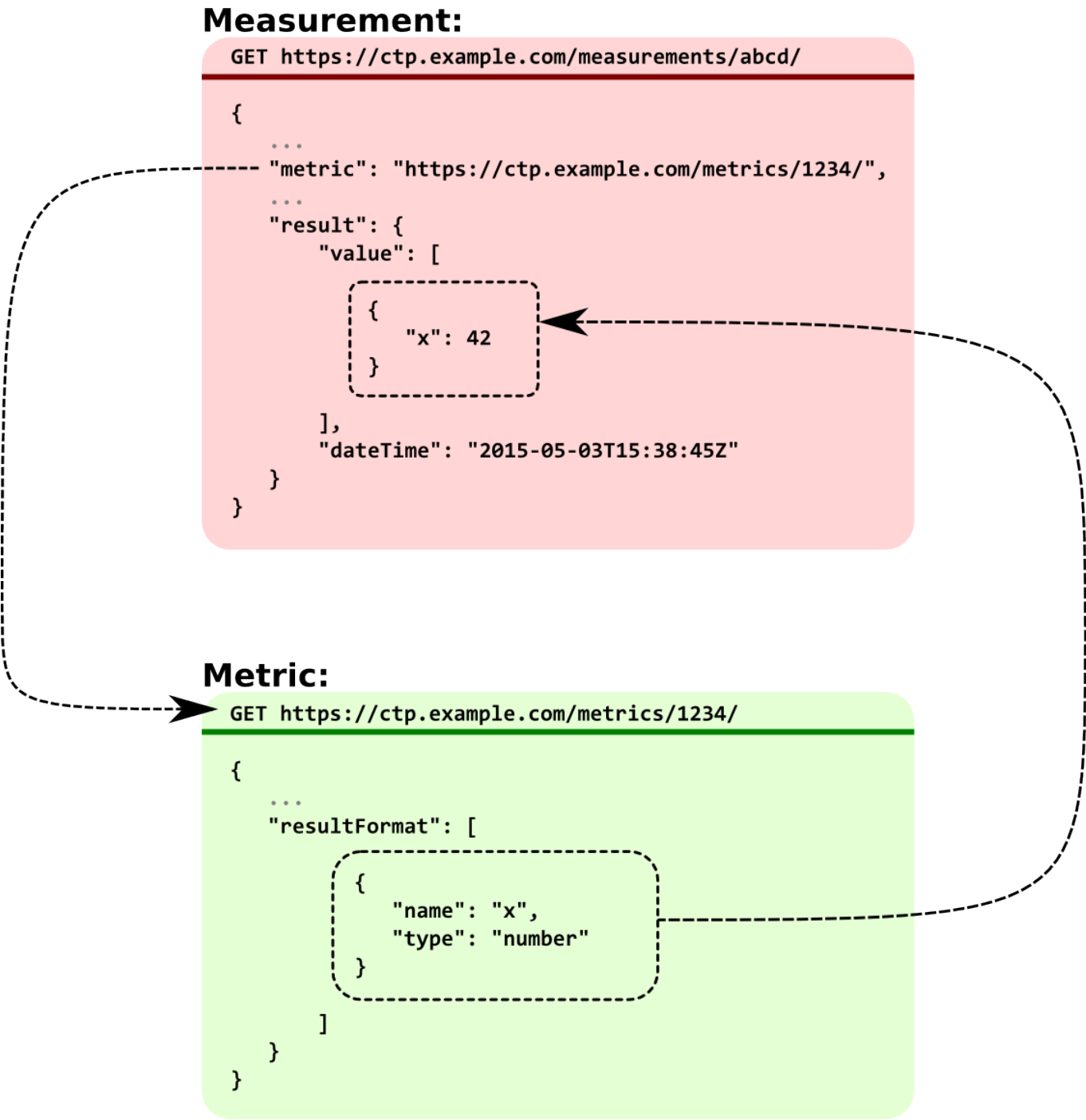


Figure 4.1: Relating measurement results format and values.

To better understand the link between the `resultFormat` property in CTP API metrics and the `value` property of a CTP API measurement, we will examine 3 examples.

Example 4.11 (A) Single value

In the simplest case, when a measurement result can be summarized into a single value (a number), the measurement result will simply be a table containing a unique column, which itself contains a unique row with one cell, as illustrated below:

Percentage
99.0

The metric corresponding to this example will look like this:

```
{
  ...
  "resultFormat": [
    {
      "name": "percentage",
      "type": "number"
    }
  ],
  ...
}
```

The measurement corresponding to this example will look like this:

```
{
  ...
  "result": {
    "value": [
      {
        "percentage": 99.0
      }
    ],
    ...
  },
  ...
}
```

Example 4.12 (B) Multi-column value

In more complex cases, a measurement result might be represented as a set of values: for example, we could imagine a security attribute describing the total number of requests made to a service (“requestTotal”) as well as the total number of requests that failed (“failedTotal”). In that case, the measurement result will be a table containing two columns named “requestTotal” and “failedTotal” respectively, with one row holding the values of these elements, as illustrated in the example below:

requestTotal	failedTotal
130	3

The metric corresponding to this example will look like this:

```
{
  ...
  "resultFormat": [
    {
      "name": "requestTotal",
      "type": "number"
    },
    {
      "name": "failedTotal",
      "type": "number"
    }
  ],
  ...
}
```

The measurement corresponding to this example will look like this:

```
{
  ...
  "result": {
    "value": [
      {
```

```

        "requestTotal": 130,
        "failedTotal": 3
    }
],
...
},
...
}

```

Example 4.13 (C) Multi-line complex values

Finally, if needed, a measurement result can be represented as a bi-dimensional table, simply extending the previous cases with multiple rows, as shown in the example below:

date	code	level
2013-01-01	1	2
2013-01-02	1	3
2013-01-03	7	1

The metric corresponding to this example will look like this:

```

{
    ...
    "resultFormat": [
        {
            "name": "date",
            "type": "string"
        },
        {
            "name": "code",
            "type": "number"
        },
        {
            "name": "level",
            "type": "number"
        }
    ],
    ...
}

```

The measurement corresponding to this example will look like this:

```

{
    ...
    "result": {
        "value": [
            {
                "date": "2013-01-01",
                "code": 1,
                "level": 2
            },
            {
                "date": "2013-01-02",
                "code": 1,
                "level": 3
            },
            {
                "date": "2013-01-03",
                "code": 7,
                "level": 1
            }
        ]
    }
}

```

```
    ],  
    ...  
  },  
  ...  
}
```

In summary, measurement results are represented as a table, which is composed of rows and named columns. CTP allows the user to query both the definition of that table and the values stored inside of that table:

- Table column names are provided as part of the metric description.
- Table values are provided as part of measurement result.

Conditions that appear in “triggers” or “objectives” are described with [CTPScript](#) (JavaScript-like) expressions that reference the measured value of an attribute in a array called “value”.

The corresponding CTPScript expression to Example 4.11 above will be a table value of one element, which itself has only one property “percentage”. A “pseudo-JavaScript” declaration of that table would be:

```
value = [ { percentage: 99.0 } ]
```

Similarly, for Example 4.12 we would have:

```
value = [ { requestTotal: 130, failedTotal: 3 } ]
```

Finally, for Example 4.13, the expression would be:

```
value = [ { date: "2013-01-01", code: 1, level: 2 },  
          { date: "2013-01-02", code: 1, level: 3 },  
          { date: "2013-01-03", code: 7, level: 7 } ]
```

Chapter 5

The API

This part of the document specifies the RESTful API of CTP and is divided in 4 parts:

- [Sub-section 5.1](#) describes the RESTful principles used throughout the CTP API.
- [Sub-section 5.2](#) specifies the API calls used in CTP, describing the available RESTful operations (GET, PUT, POST and DELETE) for each API resource and their effect.
- [Sub-section 5.3](#) describes when conditional expressions in triggers and objectives should be evaluated.
- [Sub-section 5.4](#) specifies the CTPScript language used to specify conditional expressions in objectives and triggers.

5.1 Application of RESTful principles

5.1.1 HTTP methods

A RESTful API is typically structured around “resources” which are manipulated with HTTP methods such as GET, PUT, POST and DELETE depending on the case.

In CTP the following principles are applied:

- A GET request is used to get information about a resource. GET requests never make a change to the underlying resource.
- A PUT request is used to modify part of a resource.
- A POST request is used to create a resource. The created resource **MUST** be identified with a [self-link](#) in the response.
- A DELETE request is used to delete a resource.
- All resources **MUST** have a different URL path. Query strings are not used to distinguish resources, but are used to access or modify a part of a resource through GET or PUT.

5.1.2 Application of HATEOAS

CTP only partially applies the HATEOAS (Hypertext As The Engine Of Application State) REST principle, providing descriptions of key links between resources as URLs, when relevant.

5.1.3 Content type

All requests and responses that contain a body **MUST** specify the `application/json` as content-type.

5.1.4 Status codes

As a RESTful API, CTP follows the HTTP standard and must return a relevant status code in responses to requests. In particular:

- A successful GET or PUT request SHALL return the status code **200**.
- A successful POST request SHALL return status code **201**.
- A successful DELETE request SHALL return status code **204**, with an empty body.

Unsuccessful requests MUST return appropriate status codes, and in particular:

- **400** for malformed requests (e.g. a resource identifier that uses characters outside the authorized alphabet).
- **403** for GET/PUT/POST/DELETE requests to a CTP resource that exists but is not accessible to the user.
- **404** for requests to CTP resources that do not exist.
- **405** for any request to a CTP resource with a method (GET, PUT, etc.) that is not defined in this specification.
- **409** for any PUT request on a valid CTP resource that conflicts with the current state of the resource.
- **500** for internal failure in the CTP server (e.g. database connection failure).

If the CTP server uses a HTTP authentication mechanism (see Chapter 6), then the server SHOULD return status code 401 in case of an authentication failure.

5.1.5 Error response bodies

In case of an error, in addition to setting an appropriate status code, a CTP Server SHALL send a JSON body containing a human readable error message, encoded as follows.

```
{
  "error": string
}
```

Property name	Type	Description
error	<i>string</i>	A human readable description of the error.

Note that error messages MUST not contain the self property.

5.2 Specification of the API calls

Using [previously](#) introduced notation, the following list summarizes all the valid API calls that are defined in the specification in the following sections:

Method	URL	Query string
GET	{CtpBase}/	No
GET	{CtpBase}/serviceViews	Optional
GET	{CtpBase}/serviceViews/{id}	No
GET	{CtpBase}/serviceViews/{id}/dependencies	Optional
GET	{CtpBase}/serviceViews/{id}/assets	Optional
GET	{CtpBase}/assets/{id}	No

Method	URL	Query string
GET	{CtpBase}/assets/{id}/attributes	Optional
GET	{CtpBase}/attributes/{id}	No
GET	{CtpBase}/attributes/{id}/measurements	Optional
GET	{CtpBase}/measurements/{id}	No
GET	{CtpBase}/metrics	Optional
GET	{CtpBase}/metrics/{id}	No
GET	{CtpBase}/triggers/{id}	No
GET	{CtpBase}/dependencies/{id}	No
GET	{CtpBase}/serviceViews/{id}/logs	Optional
GET	{CtpBase}/logs/{id}	No
PUT	{CtpBase}/measurement/{id}?x=state	Yes
POST	{CtpBase}/serviceViews/{id}/triggers	No
DELETE	{CtpBase}/triggers/{id}	No

5.2.1 GET {CtpBase}

5.2.1.1 Description

A **GET** request to {CtpBase} provides the information about the CTP server.

CTP servers may either make this information accessible to anybody or may restrict to authenticated CTP clients.

5.2.1.2 Request query string

None.

5.2.1.3 Request body

None.

5.2.1.4 Response body

Returns a [JSON encoded CTP server information](#).

5.2.2 GET {CtpBase}/serviceViews

5.2.2.1 Description

A **GET** request to {CtpBase}/serviceViews provides a collection of references to the service-views that are accessible to the current CTP client.

CTP servers SHALL only provide access to this collection if it applies to **service-views** that the CTP client is authorized to access.

5.2.2.2 Request query string

Uses optional [items and page](#).

5.2.2.3 Request body

None.

5.2.2.4 Response body

Returns a [collection](#), with:

- self = {CtpBase}/serviceViews/{id}/serviceViews
- collectionType = “serviceViews”

5.2.3 GET {CtpBase}/serviceViews/{id}

5.2.3.1 Description

A **GET** request to {CtpBase}/serviceViews/{id} provides information about the service-view identified by a {id}.

CTP servers SHALL only provide access to a service-view if it applies to a **service-view** that the CTP client is authorized to access.

5.2.3.2 Request query string

None.

5.2.3.3 Request body

None.

5.2.3.4 Response body

Returns a JSON encoded [service-view](#).

5.2.4 GET {CtpBase}/serviceViews/{id}/dependencies

5.2.4.1 Description

A **GET** request to {CtpBase}/serviceViews/{serviceUnit}/dependencies provides a collection of references to internal or external service-views that the current service-view relies on to provide its services.

Among these dependencies:

- This query **MUST** provide references to all service-views provided by the same provider.
- This query **SHOULD** provide references to service-views provided by third parties in the cloud supply chain, if such dependencies exist (see discussion in [Annex](#)).

The service-view dependencies provided by this query operation are unqualified: there is no distinction between service-views that are strictly necessary from service-views that are optional or used as a backup. All direct dependencies should be listed in order to provide transparency on the supply chain.

CTP servers SHALL only provide access to this collection if it applies to a **service-view** that the CTP client is authorized to access.

5.2.4.2 Request query string

Uses optional [items](#) and [page](#).

5.2.4.3 Request body

None.

5.2.4.4 Response body

Returns a [collection](#), with:

- self = {CtpBase}/serviceViews/{serviceUnit}/dependencies
- collectionType = “serviceViews” (not “dependencies”)

5.2.5 GET {CtpBase}/serviceViews/{id}/assets

5.2.5.1 Description

A **GET** request to {CtpBase}/serviceViews/{id}/assets provides a collection of references to the assets managed by a specific service-view (identified by {id}).

CTP servers SHALL only provide access to this collection if it applies to a **service-view** that the CTP client is authorized to access.

5.2.5.2 Request query string

Uses optional [items and page](#).

5.2.5.3 Request body

None.

5.2.5.4 Response body

Returns a [collection](#), with:

- self = {CtpBase}/serviceViews
- collectionType = “assets”

5.2.6 GET {CtpBase}/assets/{id}

5.2.6.1 Description

For any asset identified by a {id}, A **GET** request to {CtpBase}/assets/{id} provides information about the asset identified by a {id}.

CTP servers SHALL only provide access to an asset when it is part of a service-view that the CTP client is authorized to access.

5.2.6.2 Request query string

None.

5.2.6.3 Request body

None.

5.2.6.4 Response body

Returns a JSON encoded [asset](#).

5.2.7 GET {CtpBase}/assets/{id}/attributes

5.2.7.1 Description

A **GET** request to {CtpBase}/assets/{id}/attributes provides a collection of references to the security attributes that apply to a specific asset (identified by {id}).

CTP servers SHALL only provide access to this collection if it applies to an **asset** that the CTP client is authorized to access.

5.2.7.2 Request query string

Uses optional [items](#) and [page](#).

5.2.7.3 Request body

None.

5.2.7.4 Response body

Returns a [collection](#), with:

- self = {CtpBase}/assets/{id}/attributes
- collectionType = “attributes”

5.2.8 GET {CtpBase}/attributes/{id}

5.2.8.1 Description

A **GET** request to {CtpBase}/attributes/{id} provides the information relate to the attribute identified by a {id}.

CTP servers SHALL only provide access to an attribute when it applies to an asset that the CTP client is authorized to access.

5.2.8.2 Request query string

None.

5.2.8.3 Request body

None.

5.2.8.4 Response body

Returns a JSON encoded [attribute](#).

5.2.9 GET {CtpBase}/attributes/{id}/measurements

5.2.9.1 Description

A **GET** request to {CtpBase}/attributes/{id}/measurements provides a collection of references to the measurements that are available for a specific attribute (identified by {id}).

CTP servers SHALL only provide access to this information if the CTP client is authorized to access the identified **attribute**.

If the measurement contains an objective, it MUST be evaluated according to the [rules](#) defined in this document.

5.2.9.2 Request query string

Uses optional [items](#) and [page](#).

5.2.9.3 Request body

None.

5.2.9.4 Response body

Returns a [collection](#), with:

- self = {CtpBase}/attributes/{id}/measurements
- collectionType = "measurements"

5.2.10 GET {CtpBase}/measurements/{id}

5.2.10.1 Description

A **GET** request to {CtpBase}/measurement/{id} provides the information related to the measurement identified by a {id}.

CTP servers SHALL only provide access to a measurement when it applies to an attribute that the CTP client is authorized to access.

Measurement results MUST be formatted [according to the referenced metric](#).

5.2.10.2 Request query string

None.

5.2.10.3 Request body

None.

5.2.10.4 Response body

Returns a JSON encoded [measurement](#).

5.2.11 PUT {CtpBase}/measurements/{id}?x=state`

5.2.11.1 Description

For measurements that have the capability to be user-activated only, a **PUT** request to {CtpBase}/measurements/{id}?x=state will change the state of the measurement according to the request body, either to activate or deactivate a measurement.

CTP servers SHALL only allow this modification for a **measurement** that the CTP client is authorized to access.

This additional interface is used for cloud user-activated measurements, such as the triggering of a vulnerability scan. For example, in an IaaS setup, the cloud client could execute a HTTP **PUT** request to the “vulnerability-count” measurement for an attribute of a resource representing a particular instance he’s interested in. Once the scan is complete, its results will be accessible through the measurement result query interface described later in this document. Only a limited number of measurements are likely to be “client initiated”.

If the measurement does not have the capability to be client initiated, the server should return status code 409 (Conflict).

5.2.11.2 Request query string

Query parameter name	Type	Description
x	string	The string “state”.

5.2.11.3 Request body

```
{
  "state": string
}
```

Property name	Type	Description
state	string	Either “activated” or “deactivated”

5.2.11.4 Response body

The response body is an updated JSON representation of the [measurement](#).

5.2.12 GET {CtpBase}/metrics

5.2.12.1 Description

A **GET** request to {CtpBase}/metrics provides a collection of references to the metrics that are recognized by the CTP sever. This should include all metrics that can be used in measurements.

CTP servers may either make this collection accessible to anybody or may restrict it to authenticated users.

5.2.12.2 Request query string

Uses optional [items](#) and [page](#).

5.2.12.3 Request body

None.

5.2.12.4 Response body

Returns a [collection](#), with:

- self = {CtpBase}/metrics
- collectionType = “metrics”

5.2.13 GET {CtpBase}/metrics/{id}

5.2.13.1 Description

A **GET** request to {CtpBase}/metrics/{id} provides the information related to the metric identified by a {id}.

CTP servers may either make a metric accessible to anybody or may restrict it to authenticated users.

The relationship between measurement and metrics is detailed in Section [4.4](#).

5.2.13.2 Request query string

None.

5.2.13.3 Request body

None.

5.2.13.4 Response body

Returns a JSON encoded [metric](#).

5.2.14 GET {CtpBase}/serviceViews/{id}/triggers

5.2.14.1 Description

A **GET** request to {CtpBase}/serviceViews/{id}/triggers provides a collection of references to the triggers that are accessible in a service-view (identified by {id}).

CTP servers SHALL only provide access to this collection if it applies to **triggers** in a **service-view** that the CTP client is authorized to access.

5.2.14.2 Request query string

Uses optional [items](#) and [page](#).

5.2.14.3 Request body

None.

5.2.14.4 Response body

Returns a [collection](#), with:

- self = {CtpBase}/serviceViews/{id}/triggers
 - collectionType = “triggers”
-

5.2.15 GET {CtpBase}/triggers/{id}

5.2.15.1 Description

A **GET** request to {CtpBase}/triggers/{id} enables reading information about the trigger identified by {id}. While triggers are defined by CTP clients through a [POST request](#), this GET request allows querying the content of previously created triggers.

CTP servers SHALL only provide access to a trigger if it is part of a **service-view** that the CTP client is authorized to access. Triggers MUST be evaluated according to the rules defined in Section [5.3.2](#).

5.2.15.2 Request query string

None.

5.2.15.3 Request body

None.

5.2.15.4 Response body

Returns a JSON encoded [trigger](#).

5.2.16 POST {CtpBase}/serviceViews/{id}/triggers

5.2.16.1 Description

A trigger can be created with a **POST** request to {CtpBase}/serviceViews/{id}/triggers.

This **POST** request will only succeed for measurements that hold the capability to provide triggers (i.e. the `triggers` [property](#) in the [measurement](#) must not be null)

CTP servers SHALL only allow creating a **trigger** within a **service-view** that the CTP client is authorized to access.

5.2.16.2 Request query string

None.

5.2.16.3 Request body

The request body MUST be encoded as a normal [triggers](#) but without the self property.

5.2.16.4 Response body

Returns a JSON encoded [trigger](#).

The HTTP response code MUST be 201 if the trigger was successfully created. The self property provided in the response indicates the location of the newly created trigger, which can be accessed through GET or removed through DELETE.

5.2.17 DELETE {CtpBase}/triggers/{id}

5.2.17.1 Description

A **DELETE** request to {CtpBase}/triggers/{id} removes the specified trigger.

CTP servers SHALL only allow removing a **trigger** within a **view** that the CTP client is authorized to access

5.2.17.2 Request query string

None.

5.2.17.3 Request body

None.

5.2.17.4 Response body

None. HTTP status code 204 is returned if the delete operation succeeds.

5.2.18 GET {CtpBase}/serviceViews/{id}/logs

5.2.18.1 Description

A **GET** request to {CtpBase}/view/{id}/logs provides a collection of references to log entries that have been recorded for a client of the CTP server as the result of a trigger evaluation following the rules in Section 5.3.2. The references **MUST** be ordered chronologically. Request parameters can be used to limit the number of log entry references returned in the response.

5.2.18.2 Request query string

The request query string uses optional [items](#) and [page](#), but is extended with additional elements as follows.

Query parameter name	Type	Description
page	<i>number</i>	<i>Optional</i> page number, represented as an integer greater or equal to 0 (see Section 4.1.6).
items	<i>number</i>	<i>Optional</i> number of elements returned in the collection, represented as an integer greater than 0. (see Section 4.1.6)
oldest	<i>datetime</i>	An optional RFC_3339 UTC timestamp for the oldest log entry that can be returned. Only logs as old or newer than that date/time are returned if this parameter is specified.
newest	<i>datetime</i>	An optional RFC_3339 UTC timestamp for the newest log entry that can be returned. Only logs strictly older than that date/time are returned.
tags	<i>string</i>	An optional <i>string</i> representing a set of comma-separated tags. It selects only log entries that matching all the tags. When omitted, log entries are selected regardless of their tag.

5.2.18.3 Request body

None.

5.2.18.4 Response body

```
{
  "self": URL,
  "collectionLength": number,
  "returnedLength": number,
  "collectionType": string,
  "collection": [
    URL, ...
  ]
}
```

Property name	Type	Description
self	<i>URL</i>	Self-reference {CtpBase}/serviceViews/{id}/dependencies
collectionLength	<i>number</i>	The total number of elements in the collection.
returnedLength	<i>number</i>	The total number of elements returned in the collection in the response, which will be equal to collectionLength or less, depending on the use of the query string parameters page and items.
collectionType	<i>string</i>	The string “serviceViews” (not “dependencies”)
collection	<i>list</i>	A list of distinct links, each of the form

5.2.19 GET {CtpBase}/logs/{id}

5.2.19.1 Description

CTP exposes a RESTful interface to access individual alert records generated by triggers. A **GET** request to {CtpBase}/logs/{id} returns an individual log entry (identified by {id}), which notably contains the following information:

- A [self-reference](#) to this resource.
- A reference to the trigger that resulted in the creation of this log entry.
- The date and time the log entry was created.
- The copy of the measurement result that resulted in the trigger condition being true.
- The tag that was associated with the trigger.

CTP servers SHALL only provide access to a log entry if it is part of a service-view that the CTP client is authorized to access.

5.2.19.2 Request query string

None.

5.2.19.3 Request body

None.

5.2.19.4 Response body

Returns a JSON encoded [log entry](#), depending on the type of log entry.

5.3 Evaluating triggers and objectives

5.3.1 Evaluating objectives

An objective also applies to the result of a measurement. An objective condition may either be evaluated:

1. Each time a measurement result is updated, or
2. Each time a cloud service customer accesses the corresponding objective CTP RESTful resource.

Both approaches provide the same result from the point of view of the CTP user.

5.3.2 Evaluating triggers

A trigger condition **MUST** be evaluated each time a new measurement result is available. The processing of triggers is governed by the rules defined in the [pseudo-code](#) below, and depends on the following parameters:

- `trigger.status`: The current status of the trigger.
- `trigger.statusUpdateTime`: A timestamp of the last status update of the trigger.
- `NOW`: the current UTC date and time.

Trigger processing rules.

```
// pseudo-code

switch (trigger.status) {

  case "true":
    if ( NOW-trigger.statusUpdateTime > trigger.guardTime ) {
      Evaluate trigger.condition
      Update trigger.status and trigger.statusUpdateTime
      if (trigger.status=="error") {
        Generate error log entry
        Send XMPP notification
      }
      if (trigger.status=="true") {
        Generate normal log entry
        Send XMPP notification
      }
    } else {
      Do nothing
    }

  case "false":
    Evaluate trigger condition
    Update trigger.status and trigger.statusUpdateTime
    if (trigger.status=="error") {
      Generate error log entry
      Send XMPP notification
    }
    if (trigger.status=="true") {
      Generate normal log entry
      Send XMPP notification
    }

  case "error":
    Do nothing // note: trigger must be corrected first to remove error.
}
}
```

5.4 CTPScript: a condition description language

Conditions used in both objectives and triggers are described with CTPScript: a language that is modeled after JavaScript expressions, with many simplifications that are designed to facilitate implementation. We highlight in particular that CTPScript is limited to statements representing expressions and does not include any other language construct (such as assignments, control structures, declarations, prototypes, etc.).

CTPScript expressions mainly operate on a 4 variables, which represent the 4 fields of a [measurement result](#). The most important variable is `value`, a table of objects that represent a measured value.

Example 5.1 CTP expressions in triggers

The following are further examples of CTPScript Expression that could be provided in triggers or conditions.

```
value[0].percentage <= 99.0
```

```
(timeUTC("now")-toTimeUTC(updateTime)<3600)
```

```
matchRegex("UK", select(result, "country"))
```

5.4.1 Basic Grammar

The following rules describe the CTPScript grammar, with some further details provided in the following sub-sections. A CTPScript expression is defined by the **Expr** token in the grammar below. A CTPScript expression **Expr** is considered true if `toBoolean(Expr)` returns true, and false otherwise, taking the definition of `toBoolean` [below](#).

```
Expr:
    Literal
    Identifier
    Expr "&&" Expr
    Expr "||" Expr
    Expr "<" Expr
    Expr "<=" Expr
    Expr ">" Expr
    Expr ">=" Expr
    Expr "!=" Expr
    Expr "==" Expr
    Expr "+" Expr
    Expr "-" Expr
    Expr "/" Expr
    Expr "*" Expr
    Expr "%" Expr
    "-" Expr
    "!" Expr
    "(" Expr ")"
    FunctionCall

Literal:
    StringLiteral
    NumericLiteral
    ArrayLiteral
    ObjectLiteral
    "true"
    "false"
    "null"

Identifier:
    IdentifierName
    Identifier "[" Expr "]"
    Identifier "." IdentifierName

FunctionCall:
    Identifier "(" ")"
    Identifier "(" ParameterList ")"

ParameterList:
    Expr
    ParameterList "," Expr
```

```

ObjectLiteral:
    "{" "}"
    "{" ObjectElements "}"

ArrayLiteral:
    "[" "]"
    "[" ArrayElements "]"

ArrayElements:
    Expr
    ArrayElements "," Expr

ObjectElements:
    ObjectElementItem
    ObjectElements, ObjectElementItem

ObjectElementItem:
    IdentifierName ":" Expr
    StringLiteral ":" Expr

```

5.4.2 Definitions borrowed from other standards

CTPScripts borrow a few definitions from existing freely available standards, in particular [\[ECMA_262\]](#) version 5 (i.e. Javascript).

5.4.2.1 StringLiteral

A UTF8 string quoted with simple or double quotes as defined in ECMA 262 v5, in clause 7.8.4.

Note that CTPScript represents string in UTF-8 format, not UTF-16 as in javascript.

5.4.2.2 NumericLiteral

A double precision 64-bit format IEEE 754 floating point as defined in ECMA 262 v5, in clause 7.8.3.

5.4.2.3 IdentifierName

An IdentifierName as defined in ECMA 262 v5, in clause 7.6.

5.4.3 Types

5.4.3.1 The 7 types of CTPScript

There are 7 types defined in CTPScript:

- **string**: A UTF8 character string.
- **number**: A double precision 64-bit format IEEE 754 floating point number, which is used also to represent integers.
- **boolean**: A type that represents either true or false.
- **null**: A type that represents a null or undefined value.
- **objects**: A type that represents a list of fields as (key, value) pairs (a simplified Javascript object).
- **arrays**: A type that represents a list of values indexed by a integral number (a simplified Javascript array). CTPScript arrays have one property *length*, which represents the highest existing index in the array, plus one.

- **function**: A type that represents a function.

We call **integral number** the subset of numbers that can be represented as an integer in the range -2^{31} and $2^{31}-1$, inclusive. This type does not explicitly exist in CTPScript but is useful to define some elements in this specification.

Note that CTPScript does not provide the **undefined** type that exists in javascript.

5.4.3.2 The `typeof()` pseudo-function

For any value **a** represented in CTPScript, we define a pseudo-function `typeof(a)` which returns a string representing the type of **a** (e.g. "string", "number", "boolean", "null", "object", "array" or "function"). The implementation of the pseudo-function `typeof()` is not required: it is only defined to facilitate the description of this specification.

5.4.3.3 The `getField()` pseudo-function.

For the purpose of this specification, we define the pseudo-function `getField(a, prop)` which takes as first argument an array or object **a**, and as second argument a string or an integral number **prop**, and returns the value associated with the key **prop** in the array or object if it exists or **null** otherwise.

The implementation of the pseudo-function `getField()` is not required: it is only defined to facilitate the description of this specification.

5.4.4 Functions

CTPScript only defines 8 functions or methods, which are detailed hereafter.

Note that all CTP Arrays also define the special property `length` as specified in Section [5.4.3.1](#)

5.4.4.1 `toString(a): string`

The function `toString` converts its argument **a** to a string as follows:

1. If **a** is a string return **a**
2. If **a** is a number return a string representation of **a** as would be returned by the "%e" specifier in the C standard function `sprintf()` (as defined in ISO C99).
3. If **a** is a Boolean return "true" if **a** is true, "false" otherwise.
4. If **a** is null or an empty array, return an empty string.
5. If **a** is an object, return the string "[Object Undefined]".
6. If **a** is an (non-empty) array: 6.1. Let **r** be initialized with the value of `toString(GetField(a, 0))` 6.2. For **i** = 1 to `GetField(a, "length")-1` do 6.2.1. Concatenate the string "," to **r** 6.2.2. Concatenate the string `toString(GetField(a, i))` to **r** 6.3. Return the string **r**.
7. If **a** is a function return the string "function <id>() { [Native code] }", where <id> is replaced by the name of the function.

5.4.4.2 toBoolean(a): Boolean

The function `toBoolean` converts its argument **a** to a boolean as follows:

1. If **a** is a string return `false` if **a** is the empty string, and `true` otherwise.
2. If **a** is a number return `false` if **a** is `-0`, `+0`, or `NaN`, and `true` otherwise.
3. If **a** is a Boolean return **a**.
4. If **a** is `null`, return `false`.
5. If **a** is an object or an array return `true`.
6. If **a** is a function return `true`.

5.4.4.3 toNumber(a): number

The function `toNumber` converts its argument **a** to a number as follows:

1. If **a** is a string return the number that would be returned by applying the standard C function `atof()` to the string **a** (as defined in ISO C99).
2. If **a** is a number return **a**.
3. If **a** is a Boolean return 1 if **a** is `true`, 0 otherwise.
4. If **a** is `null`, return 0.
5. If **a** is an object or an array return `NaN`.
6. If **a** is a function return `NaN`.

5.4.4.4 Array.min(): (any type)

The method `min` returns the smallest value in an Array object as follows:

1. Let **t** be the array to which we apply the `min` method.
2. If **t** is empty return `null`.
3. Let **t[i]** define any value that verifies **t[i] <= t[j]** for any **i** ≠ **j** (using the comparison operator `<=` defined in Section 5.4.6).
4. If there is only one index **i** that verifies the property in step 3, return **t[i]**.
5. If there are several indices **i** that verify the property in step 3, return the value **t[i]** with the smallest index **i** that verifies step 3.

5.4.4.5 Array.max(): (any type)

The method `max` returns the largest value in an Array object as follows:

1. Let **t** be the array to which we apply the `max` method.
 2. If **t** is empty return `null`.
 3. Let **t[i]** define any value that verifies **t[i] >= t[j]** for any **i** ≠ **j** (using the comparison operator `>=` defined in Section 5.4.6).
 4. If there is only one index **i** that verifies the property in step 3, return **t[i]**.
 5. If there are several indices **i** that verify the property in step 3, return the value **t[i]** with the biggest index **i** that verifies step 3.
-

5.4.4.6 `matchRegexp(r, v): boolean`

The function `matchRegexp` verifies that the value `v` matches the regular expression `r`, which must be a string representing a POSIX Extended Regular Expression ¹ (POSIX ERE). The value `v` can either be a string or an array of strings. The function works as follows:

1. If `typeof(r)` is not "string" then abort raising an exception.
2. If `r` is not a correctly formed POSIX ERE, then abort raising an exception.
3. If `typeof(a)` is not "array" then goto step 5.
4. If all integral number keys `i` of `a` verify `matchRegexp(r, GetField(a, i))` then return `true`, otherwise return `false`.
5. If `typeof(a)` is not "string" then abort raising a/n exception.
6. If `s` matches `r` according to POSIX ERE matching rules then return `true`, otherwise return `false`.

5.4.4.7 `select(s, a): table`

The function `select` iterates through an array `a` of objects and returns an array that is created by selecting the key `s` in each one of the objects of `a`. The function operates as follows:

1. If `typeof(a)` is not "array" then abort raising an exception.
2. Let `t` be an array. For each index `i` representing an integral number key in `a`, define `t[i]` as `getField(getField(a, i), s)`.
3. Return the array `t`.

5.4.4.8 `timeUTC(s): number`

The function `timeUTC` takes a string `s` and converts it to a number representing the number of seconds since 00:00:00 UTC, January 1, 1970, which is referred as the "epoch".

The string `s` must be either:

- An Internet Date/Time format as defined in RFC 3339 (e.g. "2015-07-20T12:34:56Z")
- The string "now".

The function operates as follows:

1. If `typeof(s)` is not a string then abort raising an exception.
2. If `s` is the string "now", return the current UTC time expressed as the number of seconds elapsed since the "epoch".
3. If `s` is an RFC 3339 date/time string representing a UTC date before the "epoch", return the corresponding UTC time expressed as a negative number of seconds before the "epoch".
4. If `s` is an RFC 3339 date/time string representing a UTC date equal or greater than the "epoch", return the corresponding UTC time expressed as a number of seconds elapsed since the "epoch".
5. In all other cases, abort raising an exception.

¹ http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html

5.4.5 Boolean operators ||, &&

The expression `a || b` is evaluated as follows:

- If `toBoolean(a) = true` then return **a** else return **b**.

The expression `a && b` is evaluated as follows:

- If `toBoolean(a) = false` then return **a** else return **b**.

Note

this approach is similar to the one used in Javascript (ECMAScript) but is different from more traditional languages such as C or Java.

5.4.6 Comparison Operators <, <=, >, >=, ==, !=

Let `compareString(a, b)` be the pseudo-function that compares two UTF-8 strings in lexicographic order and returns (-1) if `a < b`, (1) if `a > b` and (0) if **a** and **b** are equal. Comparison is done on the basis of the UTF code unit values (not bytes).

The expression `a < b` is evaluated as follows:

1. If `typeof(a)` is "string" and `typeof(b)` is "string" return true if `compareString(a, b)` equals -1
2. Let **an** = `toNumber(a)` and **bn** = `toNumber(b)`
3. If **an** is NaN or **bn** is NaN return false.
4. Taking into account the canonical ordering of numbers, return true if **an** < **bn**, otherwise return false.

The expression `a == b` is evaluated as follows:

1. If `typeof(a)` is "string" and `typeof(b)` is "string" return true if `compareString(a, b)` equals 0
2. Let **an** = `toNumber(a)` and **bn** = `toNumber(b)`
3. If **an** is NaN or **bn** is NaN return false.
4. Taking into account the canonical ordering of numbers, return true if **an** == **bn**, otherwise return false.

The expression `a <= b` is evaluated as follows:

1. Return the result of evaluating (`a < b || a == b`)

The expression `a >= b` is evaluated as follows:

1. Return the result of evaluating (`a > b || a == b`)

The expression `a != b` is evaluated as follows:

1. Let **r** be the evaluation of `a == b`
2. If **r** is true return false, otherwise return true.

The expression `a > b` is evaluated as follows:

1. Let **r** be the evaluation of `a < b`
 2. If **r** is true return false, otherwise return true.
-

5.4.7 Multiplicative and additive binary operators: +, -, *, /, %

The expression $a + b$ is evaluated as follows:

1. If `typeof (a)` is "string" and `typeof (b)` is "string", return the string formed by appending the string **b** to the end of **a**.
2. If `typeof (a)` is number and `typeof (b)` is "number", return the numerical sum **a+b**, according to IEEE 754.
3. In all other cases, return NaN.

The expression $a - b$ is evaluated as follows:

1. If `typeof (a)` is "number" and `typeof (b)` is "number", return the numerical subtraction **a-b**, according to IEEE 754.
2. In all other cases, return NaN.

The expression $a * b$ is evaluated as follows:

1. If `typeof (a)` is "number" and `typeof (b)` is "number", return the numerical multiplication **a * b**, according to IEEE 754.
2. In all other cases, return NaN.

The expression a / b is evaluated as follows:

1. If `typeof (a)` is "number" and `typeof (b)` is "number", return the numerical division **a/b**, according to IEEE 754.
2. In all other cases, return NaN.

Note

Divisions by 0 return $-\text{Inf}$, $+\text{Inf}$, or NaN depending on the case, as specified in IEEE 754.

The expression $a \% b$ is evaluated as follows:

1. If `typeof (a)` is "number" and `typeof (b)` is "number", return a result equivalent to the computation of `fmod(a, b)` where `fmod` is a standard C function, which computes the floating-point remainder of dividing a by b (as defined in ISO C99).
2. In all other cases, return NaN.

5.4.8 Identifiers

The CTPScript only defines 4 identifiers, corresponding to the 4 properties of a measurement result as defined in Section 4.2.6.2:

- `value`
- `updateTime`
- `authorityId`
- `signature`

The most important identifier is `value` which represents an “array of objects” or in other words a bi-dimensional table. This is an immutable bi-dimensional table so in effect it acts as a constant. As detailed in the section Section 4.4, the table elements `value[0]`, `value[1]`, ..., `value[n]` represent a set of rows, while the fields of each individual `value[i]` represent the cells of this bi-dimensional table. The field names that are defined in each `value[i]` depend on the definition of the measurement, as specified in the `resultFormat` [property of a measurement](#).

The value of `updateTime` can be used in CTPScript expression to evaluate the date of the last measurement of a security attribute, notably in combination with the CTPScript function `timeUTC` (see Section 5.4.4.8).

5.4.9 Evaluating CTPScript expressions

CTPScript expressions are evaluated according to the timing rules defined in Section 5.3.

The result of the evaluation of a CTPScript expression is always one of the three following values: `false`, `true` or `error`.

The evaluation **MUST** generate the value `error` if any of the following conditions occur:

- The CTPScript contains a syntax error.
- The CTPScript calls a function that does not exist or attempts to access a field in a null variable.
- A function used in the expression generates an exception.

Implementation **SHOULD** provide descriptive error information as part of the data logged as a consequence of an error (see Section 4.2.8.3).

In all other cases, the function `toBoolean()` is applied to the result providing a final value that is equal to either `true` or `false`.

Chapter 6

Security considerations

6.1 Four security requirements for the CTP API

The CTP API can be used to provide access to two types of data:

- **Private data** representing customer-specific or sensitive data.
- **Public data** that represent generic information about a cloud service, excluding sensitive or customer-specific data.

As described in the [CTP data model](#), except for metrics, each CTP API resource can be tied directly or indirectly to a service-view through a hierarchy (e.g. an attribute is tied to an asset, which itself is tied to a service-view, which is then implicitly tied to a customer). CTP therefore uses the notion of a [service-view](#) to represent a limitation on the scope of information accessible to a cloud customer. Each customer should get a different service-view (or set of service-views) to access private data. Cloud providers MAY also define an additional shared “public” service-view to access public data, which will not require authentication.

In the remaining of this section, we only discuss the access to private data and the security considerations that it entails.

In order to protect private data, implementation **MUST** satisfy the following requirements:

Requirement 1

Each API call to non-public data **MUST** carry authentication information in order to establish that the client is authorized to access the resource that is requested.

Requirement 2

All communications between the CTP client and the CTP server **MUST** be secured by a transport mechanism that offers both confidentiality and integrity for the data exchanged, including the authentication data from requirement 1.

Requirement 3

All security mechanisms **MUST** be compatible with the stateless nature of the CTP RESTful API. Session information should not be maintained across consecutive CTP API calls.

Requirement 4

The security mechanisms **MUST** preserve the API defined in this specification, including URL formats, query strings and JSON data structures. The security mechanism **MAY** however result in the addition or modification of HTTP headers that are part of the RESTful API requests.

Implementers are free to choose any set of security mechanisms that verifies the 4 requirements above. However, for interoperability purposes, implementers are strongly encouraged to consider implementing the recommendations described in [Section 6.3](#).

6.2 Example of solutions

The following paragraphs give a brief overview of possible solutions that address the 4 security requirements of the CTP API, for informational purposes only.

Currently, the most straightforward way to implement requirement 2 is the use of SSL/TLS, with a server-side certificate.

Requirement 1 implies that the client is identified and authenticated. Identification allows distinguishing one client from another (e.g. a user name), while authentication is used to ascertain the claimed identity (e.g. a password).

Clients can be identified through several means, for example:

1. Each client can be assigned a different {CTPBase} [URL](#). (e.g. cloud service customer “foo” accesses the CTP server through <http://example.com/ctp/foo/>).
2. Each client can be identified as part of an underlying authentication mechanism through a “user name” (e.g. HTTP BASIC or DIGEST access).
3. Each client can be implicitly identified and authenticated simultaneously through a security token provided in an HTTP header (e.g. such as in OAuth2 token placed in an `Authorization:` header).

Clients can be authenticated by several means. In simple scenarios, client authentication can be performed with traditional HTTP authentication (BASIC) over SSL/TLS. For more advanced scenarios, stronger security is provided with mechanisms such as OAuth and other security-token based approaches.

All the above mechanisms also typically satisfy requirements 3 and 4. They do not require the CTP server to maintain any state between two consecutive API calls. They do not require modification of the API.

6.3 Recommended security mechanisms

In order to promote interoperability, both CTP clients and servers **SHOULD** consider implementing the following solution based on OAuth2 and TLS:

- All communications **MUST** take place over TLS version 1.2 or above, with a server side certificate, using current best practice approaches ¹.
- CTP client requests **MUST** be authenticated with a securely generated OAuth2 "bearer" token as defined in [\[RFC_6750\]](#), transmitted in the HTTP "Authorization" header of the request.
- If a CTP client attempts to access a CTP Resource without providing an access token or if the token is incorrect, the CTP server **MUST** respond with a `WWW-Authenticate` header containing a `scope` field set to the value `CTP_API_1.0`. CTP clients **MAY** use the presence of this field to detect the implementation of the security mechanism recommended in this section. CTP server **MUST NOT** set this field to the value defined above if they do not implement the security mechanism defined in this section.

The acquisition and the renewal of the OAuth2 "bearer" should be conducted according to the OAuth2 framework (See [\[RFC_6749\]](#)), and is out of scope of this document.

Let `vaikoh6gahJeekoovaiphiad` represent a OAuth2 bearer token, the following command shows a secure GET request to the URL `https://ctp.example.com/assets/02` using `curl`, a widely used command line tool [\[CURL\]](#).

```
curl -s -H "Authorization: Bearer vaikoh6gahJeekoovaiphiad" https://ctp.example.com/assets/02/ ↵
```

The CTP server will decode the header and extract the token. The token is used both as an identifier and as an authenticator.

¹ For example https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

Chapter 7

References

7.1 Bibliography

- [1] [CTP2] Ron Knode, Doug Egan. A precis for the cloudtrust protocol (v2.0). CSC, July 2010.
 - [2] [ENISA] European Union Agency for Network and Information Security, Glossary, <https://www.enisa.europa.eu/activities/risk-management/current-risk/risk-management-inventory/glossary>
 - [3] [ECMA_262] ECMA International. ECMAScript Language Specification, 5.1 edition. June 2011. Available at <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>
 - [4] [ISO_27000] ISO/IEC. ISO/IEC 27000:2014 Information technology — Security techniques — Information security management systems — Overview and vocabulary. Published.
 - [5] [ISO_19086] ISO/IEC. ISO/IEC CD 19086-1 Information technology — Cloud computing — Service level agreement (SLA) framework and Technology — Part 1: Overview and concepts. Under development.
 - [6] [NIST] NIST, Cloud Computing Service Metrics Description. Special Publication 500-307, NIST Cloud Computing Reference Architecture and Taxonomy Working Group, DRAFT.
 - [7] [RFC_4648] S. Josefsson. RFC 4648: The Base16, Base32, and Base64 Data Encodings. The Internet Engineering Task Force, October 2006.
 - [8] [RFC_2119] S. Bradner. RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. The Internet Engineering Task Force, March 1997.
 - [9] [RFC_6120] P. Saint-Andre, RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core. The Internet Engineering Task Force, March 2011.
 - [10] [RFC_3986] T. Berners-Lee, R. Fielding, L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. The Internet Engineering Task Force, January 2005.
 - [11] [RFC_7515] M. Jones, J. Bradley, N. Sakimura. RFC 7515: JSON Web Signature (JWS). The Internet Engineering Task Force, May 2015.
 - [12] [CURL] curl, a command line tool and library for transferring data with URL syntax. Available at <http://curl.haxx.se/>
 - [13] [RFC_2617] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, L. Stewart. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication. The Internet Engineering Task Force, June 1999.
 - [14] [RFC_6749] D. Hardt, Ed. The OAuth 2.0 Authorization Framework, Internet Engineering Task Force (IETF), October 2012.
-

- [15] [RFC_6750] M. Jones, D. Hardt, RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage, Internet Engineering Task Force (IETF), October 2012.
- [16] [RFC_3339] G. Klyne, C. Newman, RFC 3339: Date and Time on the Internet: Timestamps Internet Engineering Task Force (IETF), July 2002.

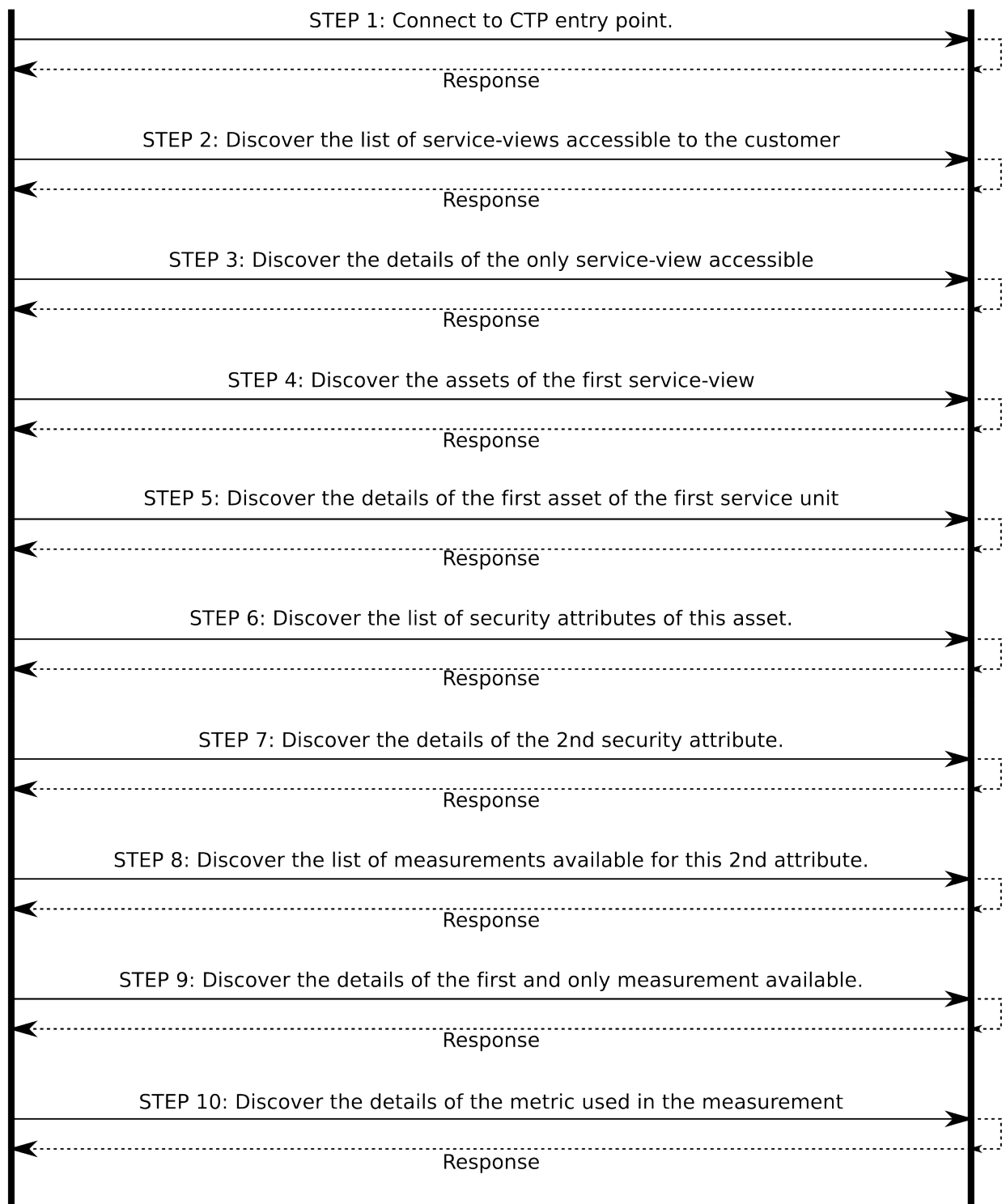
Appendix A

API Examples

This section provides the transcript of a sample exchange between a CTP client and a CTP server using the CTP API, as summarized in the following sequence diagram.

CTP Client

CTP Server



This CTP exchange between a client and a CTP server, starts from the connection to the CTP entry point and includes the discovery of service-views, cloud resources, security attributes, measurements and the creation of a trigger.

In the examples, the CTP entry point {CtpBase} is <https://ctp.cloudsecurityalliance.org/ctp>.

For the sake of simplicity the following examples do not describe the underlying security exchanges (SSL/TLS) nor the identification/authentication of the client.

A.1 Step 1: Connecting to the entry point of the CTP server

As a starting point a cloud service customer using CTP will likely connect to the CTP entry point, where he can obtain information on the version of CTP that is running as well as a pointer to a collection of service units that are under the supervision of this CTP server.

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/ HTTP/1.1
...
```

Response

```
HTTP 200 OK
...

{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/",
  "name": "ikialab cloud service",
  "annotation": "This is the CTP server of Ikialab.net",
  "version": "0",
  "provider": "net.ikialab",
  "views": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews",
  "metrics": "https://ctp.cloudsecurityalliance.org/ctp/metrics"
}
```

A.2 Step 2: Discovering the service-views accessible to the customer

The previous query returned a link to the collection of service-views exposed to the customer (<https://ctp.cloudsecurityalliance.org/ctp/serviceViews>). We can access this collection by issuing a GET request.

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/serviceViews HTTP/1.1
...
```

Response

```
HTTP 200 OK
...

{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews",
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/",
  "collectionLength": 1,
  "returnedLength": 1,
  "collectionType": "views",
  "collection": [
    {
      "link": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ",
      "name": "main"
    }
  ]
}
```

A.3 Step 3: Accessing information on the service-view

The previous example showed that the customer has only access to one service-view. We can now query the corresponding service-view for further details.

The response tells us about the provider responsible for the service unit, provides references to the dependencies of the service unit and its service resources. We note that no [service class](#) is specified.

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ HTTP/1.1
...
```

Response

```
HTTP 200 OK
...
{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ",
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/",
  "name": "main",
  "annotation": "The main service-view provided to the customer",
  "provider": "net.ikialab",
  "dependencies": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ/ ←
    dependencies",
  "assets": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ/assets ←
    ",
  "serviceClass": "",
  "logs": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ/logs",
  "triggers": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ/ ←
    triggers"
}
```

A.4 Step 4: Discovering the list of assets of the service-view

We can now query the list of assets that are part of the service-view. The query returns a list of 4 references to assets.

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ/assets HTTP/1.1
...
```

Response

```
HTTP 200 OK
...
{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ/assets",
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhbQ",
  "collectionLength": 4,
  "returnedLength": 4,
  "collectionType": "assets",
  "collection": [
    {
      "link": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhbU",
      "name": "https://webserver.example.com"
    },
    {
      "link": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fha1",

```



```
    "name": "https://database.example.com"
  },
  {
    "link": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fha2"
  },
  {
    "link": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fha3"
  }
]
}
```

A.5 Step 5: Discovering the details of the first asset of the service-view

We now query information about the first asset in the group of 4 we previously discovered. The response to the query provides us with a reference to the collection of security attributes that apply to this cloud asset (In this example, there is no [asset class](#) applicable to this cloud asset). We learn that the name of the asset is "https://webserver.example.com/", which seems to identify a SSL/TLS server.

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhaU HTTP/1.1
...
```

Response

```
HTTP 200 OK
...

{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhaU",
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/serviceViews/VIYQUT1WG628fhaQ",
  "name": "https://webserver.example.com",
  "annotation": "Webserver running on Linux Ubuntu (apache)",
  "attributes": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhaU/ ↔  
  attributes",
  "assetClass": ""
}
```

A.6 Step 8: Discovering the list of security attributes of the asset

We now obtain a collection of references to security attributes that apply to this asset, using the “collection” paradigm we have already applied in some of the previous examples.

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhaU/attributes HTTP/1.1
...
```

Response

```
HTTP 200 OK
...

{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhaU/attributes",
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhaU",
  "collectionLength": 2,
```

```
"returnedLength": 2,
"collectionType": "attributes",
"collection": [
  {
    "link": "https://ctp.cloudsecurityalliance.org/ctp/attributes/VIYQUT1WG628fhbV",
    "name": "availability"
  },
  {
    "link": "https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu",
    "name": "confidentiality-of-access"
  }
]
}
```

A.7 Step 7: Discovering the details of the second security attribute of the asset

We decide now to explore more in detail the second attribute of the asset we have selected. The response notably shows us that the security attribute offered here is “confidentiality-of-access” (which could reference to the level of confidentiality of data in transit to the asset previously identified as `https://webserver.example.com/`).

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu HTTP/1.1
...
```

Response

```
HTTP 200 OK
...

{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu",
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/assets/VIYQUT1WG628fhbU",
  "name": "csa:confidentiatity:data-in-transit"
  "annotation": "confidentiality of data in transit with SSL/TLS",
  "name": "confidentiality-of-access",
  "measurements": "https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu/ ↔
    measurements"
}
```

A.8 Step 8: Discovering the list of measurements available for this attribute.

We now query how this attribute called “confidentiality-of-access” can be measured. We find out that there is only one proposed measurement.

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu/measurements HTTP ↔
/1.1
...
```

Response

```
HTTP 200 OK
...

{
```

```

"self": "https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu/ ←
  measurements",
"scope": "https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu",
"collectionLength": 1,
"returnedLength": 1,
"collectionType": "measurements",
"collection": [
  {
    "link": "https://ctp.cloudsecurityalliance.org/ctp/measurements/VWcIa-sfTYZSm6Qw"
  }
]
}

```

A.9 Step 9: Discovering the details of the first measurement available for the selected security attribute

We now request more details on the one and only measurement that is offered here for the attribute “confidentiality-of-access”. The response is richer than in previous examples. We notably learn that the measurement is based on a metric which further details by accessing the URL <https://ctp.cloudsecurityalliance.org/ctp/metrics/VWcIa-sfTYZSm6Qv> as we will show in the [next step](#).

The response shows us that the current measurement of the “confidentiality-of-access” attribute of the resource “<https://webserver.example.com/>” is 7, which corresponds to the equivalent of a 128 bit key symmetric cryptographic algorithm, according to the “ECRYPT II” scale referenced in the metric detailed in the [next step](#). The measurement result is dated from Jan 19th 2015 in this example.

The response also shows us that the CSP is committed to maintaining a security level of at least “7” for the attribute “confidentiality-of-access” as quantified with the measurement. This is indicated by the expression `value[0].level>=7`.

Request

```

GET https://ctp.cloudsecurityalliance.org/ctp/measurements/VWcIa-sfTYZSm6Qw HTTP/1.1
...

```

Response

```

HTTP 200 OK
...
{
  "self": "https://ctp.cloudsecurityalliance.org/ctp/measurements/VWcIa-sfTYZSm6Qw",
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/attributes/VWcIa-sfTYZSm6Qu",
  "name": "",
  "annotation": "",
  "metric": "https://ctp.cloudsecurityalliance.org/ctp/metrics/VWcIa-sfTYZSm6Qv",
  "result": {
    "value": [
      {
        "level": 7
      }
    ],
    "updateTime": "2015-05-28T15:22:03.674+03:00",
    "authorityId": "",
    "signature": ""
  },
  "objective": {
    "condition": "value[0].level>=7",
    "status": "true"
  },
  "createTrigger": ""
}

```

```
"userActivated": false,  
"state": "activated"  
}
```

A.10 Step 10: Getting information about the metric used in the measurement

In the previous step, we only had a pointer to the metric used. By querying this URL, we can now get more details on the metric used in the measurement. We notably see that:

- The metric is defined in a “human readable” standard available at the (fictitious) URL <https://cloudsecurityalliance.org/ctp/metrics#csa:cryptographic-strength>.
- The metric has one parameters: a scale, which is here the ECRYPT II scale (see <http://www.keylength.com/en/3/>).
- The measurement result must be expressed as a number named "level".

Request

```
GET https://ctp.cloudsecurityalliance.org/ctp/metrics/VWcIa-sfTYZSm6Qv HTTP/1.1  
...
```

Response

```
HTTP 200 OK  
...  
  
{  
  "self": "https://ctp.cloudsecurityalliance.org/ctp/metrics/VWcIa-sfTYZSm6Qv",  
  "scope": "https://ctp.cloudsecurityalliance.org/ctp/",  
  "name": "cryptographic-strength",  
  "annotation": "",  
  "baseMetric": "https://cloudsecurityalliance.org/ctp/metrics#csa:cryptographic-strength",  
  "measurementParameters": [  
    {  
      "name": "scale",  
      "type": "string",  
      "value": "ECRYPT II"  
    }  
  ],  
  "resultFormat": [  
    {  
      "name": "level",  
      "type": "number"  
    }  
  ]  
}
```

Appendix B

Accessing information about the supply chain

Cloud services may depend on one or several others following a supply chain model, in the case where a Cloud service customer is also a CSP, which is taken into account in the design of CTP (e.g. when a SaaS providers uses a third party IaaS provider). The overall security of the service depends not only on the security of the cloud service provider at the end of the supply chain but also on the security of third party cloud service providers further down the supply chain.

Despite the fact that the security depends on several components in a supply chain, the responsibility of reporting the security level of the overall service remains on the shoulders of the “primary” CSP at the end of the supply chain: the CSP that directly faces the cloud service customer. It is the responsibility for this “primary” CSP to gain assurance about the “secondary” CSPs involved in the supply chain, possibly using CTP as well.

If the “primary” CSP uses CTP to exchange information with cloud service customers, and if this “primary” CSP itself uses CTP to exchange data with “secondary” CSPs, it may choose to “proxy” some information originating from secondary CSPs back to the cloud service customer.

This proxy approach is useful in particular to automate the discovery by the cloud service customer of all entities in the supply chain, through direct exchanges with the “primary” CSP alone using the service-view discovery API (See [dependencies](#)). Omitting some details (such as cycle avoidance), this discovery would be conducted as follows:

1. The client queries the Base URL of the “primary” CSP, and obtains the name of the provider (see the “provider” property [here](#)). Note that in practice the client is expected to know this information already.
2. The client queries the “primary” CSP for the collection of all service-views that are accessible to the client.
3. The client queries each [service-view](#) in the collection, and builds a graph representing the supply chain with the following information:
 - a. Each service-view contains a pointer to a collection of dependencies, where each dependencies is itself a service-views that is added to the list of service-views to query.
 - b. Third party providers in dependencies are identified by the fact that the “provider” field in the service-view is different from the one collected in point 1 as part of the Base URL.
 - c. Links between providers can be established by walking through the collection of dependencies (see the “dependencies” field in the service-view)

In the above discovery process, the “primary” CSP will forward queries regarding third party Service-views down the supply chain and will relay back responses to the client. In most cases, these queries can be “cached” by the cloud provider since the supply chain is expected to change infrequently.

The information that is provided by the “secondary CSPs” in the supply chain does not have to include “assets”, “attributes” or “measurements”. While the primary CSP is expected to provide all possible elements, the secondary CSPs are only expected to provide a description of service-views and their dependencies, for the sole purpose of describing the supply chain.

Providing visibility on the supply chain is often considered as a compliance requirement for CSPs with regards to EU data protection rules. CTP’s Service-view discovery API could make this process automatic.

Appendix C

CTP project history

The following paragraphs are mainly intended for readers familiar with previous versions of CTP or who want an overview of the CTP project status.

The principles of the Cloud Trust Protocol were defined by CSC in 2010 in “[A precis for the cloustrust protocol \(v 2.0\)](#)”, hereafter referred to simply as “CTP 2.0”. The goal of this work is to contribute to the advancement of CTP from the abstract framework defined in CTP 2.0 to the specification of an implementable protocol, hereafter referred to as “CTP 3.0”. To achieve this, the work on CTP 3.0 has been divided into the following tasks or deliverables:

1. Overview of the CTP Version 3.0
2. CTP Reference Specification of the Elements of Transparency (EoTs)
3. CTP Business Use Cases
4. CTP Technical Model and API
5. CTP Integration in the GRC Stack

This document is a proposal of a **CTP Technical Model and API**. Security attributes manipulated by the CTP API are specified in a separate document.

Some of key differences between CTP 2.0 and CTP 3.0 can be summarized as follows:

- *“Security attributes” instead of “Elements of Transparency” as building blocks:* From a high level perspective, the CTP 2.0 architecture classifies the information exchanged between a cloud service customer and a cloud provider in broad categories called “Elements of Transparency” (EoT), such as for example “Alerts”, “Provider capabilities and service claims” or “Results of latest vulnerability assessment”. From a technical perspective, these Elements of Transparency need to be implemented through a concrete set of mechanisms in association with a relevant data model. This document focuses on “security attributes” as a fundamental building block of the CTP API data model. Most, but not all, Elements of Transparency envisioned in CTP 2.0 can be translated in a set of security attributes, whose values will be queried by the cloud service customer. The use of word “attribute” also aims to better align CTP with the terminology of ISO/IEC 27000.
 - *Providing a separation between mechanisms and content:* In CTP 2.0, elements of transparency encompass both mechanisms and content. CTP 3.0 attempts to provide a clear distinction between security attributes and mechanisms (e.g. “availability” versus “creating triggers”).
 - *Minimizing dependencies on external standards:* CTP 2.0 relies heavily on SCAP as a foundation while CTP 3.0 is designed to be implementable on a standalone basis, with dependencies only on simple standards such JSON and several RFCs.
-