Customer Data Platform 1.0.0

Working Draft

Table of Contents

Committee Specification Draft 01 / Public Review Draft 01 (Working Draft)

Not Yet Published (29 March 2019)

Specification URIs:

This Version:

- http://docs.oasis-open.org/cxs/cdp/v1.0/csprd01/cdp-v1.0-csprd01.adoc (Authoritative)
- http://docs.oasis-open.org/cxs/cdp/v1.0/csprd01/cdp-v1.0-csprd01.html
- http://docs.oasis-open.org/cxs/cdp/v1.0/csprd01/cdp-v1.0-csprd01.pdf

Previous Version: N/A

Latest Version:

- http://docs.oasis-open.org/cxs/cdp/v1.0/cdp-v1.0.txt (Authoritative)
- http://docs.oasis-open.org/cxs/cdp/v1.0/cdp-v1.0.html
- http://docs.oasis-open.org/cxs/cdp/v1.0/cdp-v1.0.pdf

Technical Committee:

OASIS Context Server (CXS) TC

Chair(s):

- Serge Huber (shuber@jahia.com), Jahia Solutions Group SA
- Thomas Lund Sigdestad (tsi@enonic.com), Enonic

Editor(s):

- Thomas Lund Sigdestad (tsi@enonic.com), Enonic
- Serge Huber (shuber@jahia.com), Jahia Solutions Group SA

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- Sample server implementation: https://github.com/oasis-tcs/cxs-cdp

To run server locally:

```
git clone https://github.com/oasis-tcs/cxs-cdp
git checkout tags/v1.0.0
cd cxs-cdp/server/
npm install
npm start
```

Related work:

This specification is related to:

- GraphQL : https://graphql.org
- GraphQL specification : https://facebook.github.io/graphql/
- Apache Unomi: https://unomi.apache.org

Abstract:

This specification aims to standardize exchange of customer data across systems and silos by defining a web-based API using GraphQL. The GraphQL api is a self-documented and strongly typed interface. It is designed to be dynamically extended, and allows extensive implementation specific customization.

Status:

This document was last revised or approved by the OASIS Context Server (CXS) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=cxs#technical.

TC members should send comments on this specification to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/cxs/.

This specification is provided under the Non-Assertion Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis-open.org/committees/cxs/ipr.php).

NOTE	Any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

CDP-v1.0

Customer Data Platform Version 1.0. Edited by Thomas Lund Sigdestad & Serge Huber. 21 March 2019. OASIS Committee Specification Draft 01 / Public Review Draft 01. http://docs.oasis-open.org/cxs/cdp/v1.0/csprd01/cdp-v1.0-csprd01.html. Latest version: http://docs.oasis-open.org/cxs/cdp/v1.0/cdp-v1.0.html.

Notices

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS

Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark for above guidance.

1. Introduction

Today, virtually all business is at some point digital, and the number of systems involved and the set of data collected is growing rapidly. Each system creates new silos of customer data, spreading sensitive and personal data across both organizational and geographical borders.

Even digital savvy businesses struggle to control and utilize this information. Businesses and users also rely on such data to be accessible in real-time, and at scale - for instance to deliver personalizations. Additionally businesses now face severe legal charges if customer data is not treated according to regulatory requirements (ref GDPR).

The Customer Data Platform (CDP) specification aims to standardize exchange of customer data across systems and silos. This enables centralization of customer data - consequently giving control of the data back to the business, and the customers.

The CDP standard is defined as a web-based API using GraphQL - providing a self-documented and strongly typed interface.

It has been an explicit goal of the CXS committee to allow extensive customization of CDP deployments, in order to fit the need of each different organization. As such, the API dynamically evolves as you customize your deployment.

1.1. Terminology

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

1.2. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, http://www.rfc-editor.org/info/rfc2119.

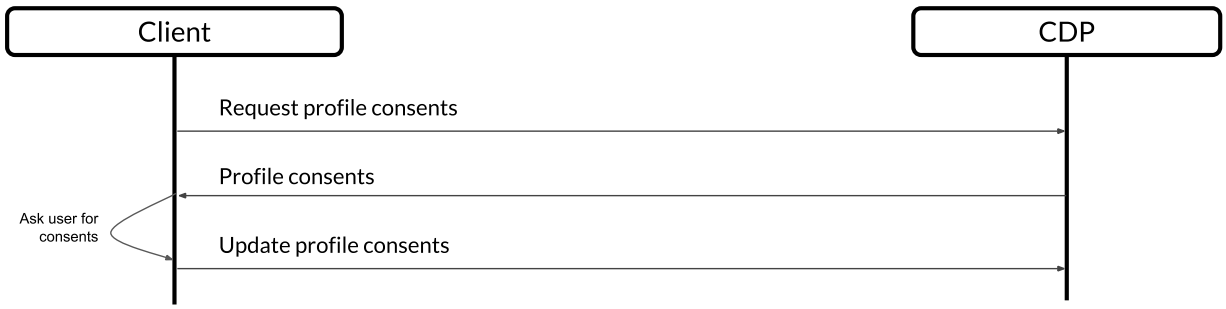
[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, http://www.rfc-editor.org/info/rfc8174.

2. Use Cases

In this section we present a selection of use cases that are relevant to the scope covered by the CDP specification. They are by no means exhaustive, but illustrate what may be achieved through the use of the standard.

2.1. Consent management

Privacy is a very important topic, especially when dealing with visitor data. For example, new legislation such as the GDPR imposes strict restrictions on how visitor data collection should be processed. It is therefore very important that the CDP specification provide standardized ways of complying with more and more stringent requirements.



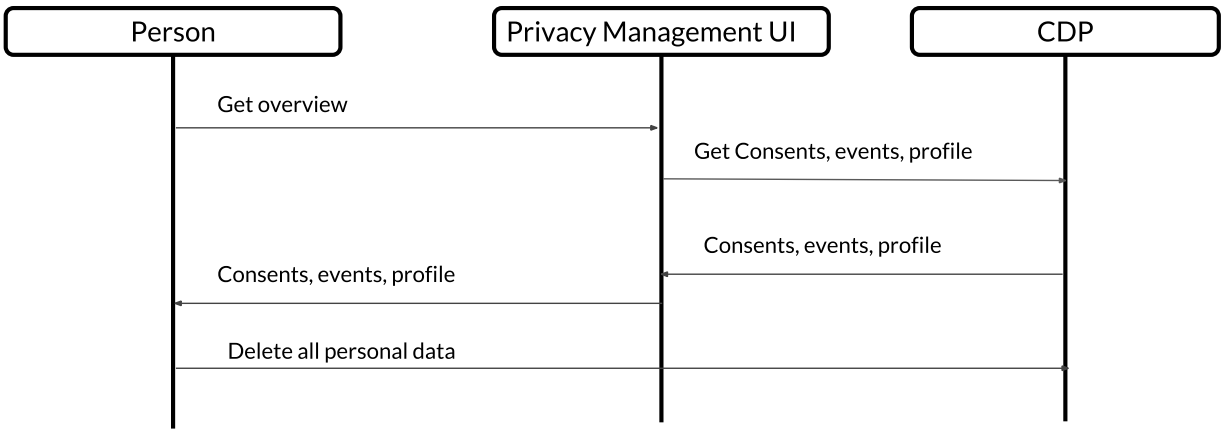
In the above use case we illustrate the support of consent management that is available in CDP-compliant systems. A visitor profile may store the state of consents (granted or not) and these may be updated by using specialized event types.

2.2. Privacy management

A core requirement for any business handling personal data is transparency. The ability to provide users with insight into what data are stored, and optionally manage their own data is essential. A CDP not only aggregates personal data from various sources, but can also manage consents and profiles. In specific cases, CDP may act as the source-of-truth across systems, and enable effective privacy management.

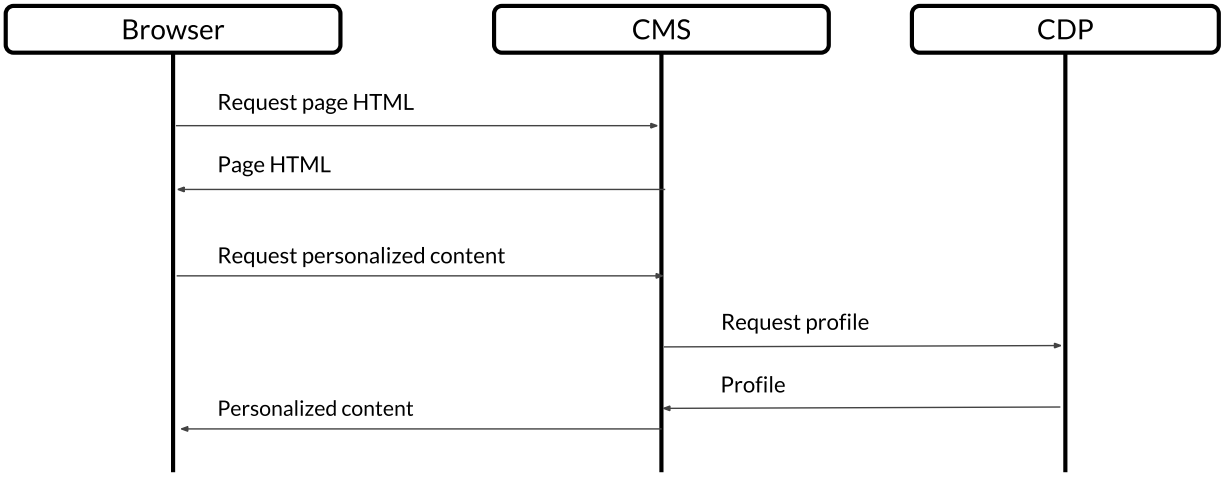
In this use case, a custom "privacy management interface" is deployed in front of the CDP. The interface should be specifically designed for the business, and require authentication.

Authenticated users can then in a controlled fashion see, delete or update their personal data. Examples or such might be events, properties or consents.



2.3. Personalization

A common use case is delivering better and more personalized experiences across applications and web sites.

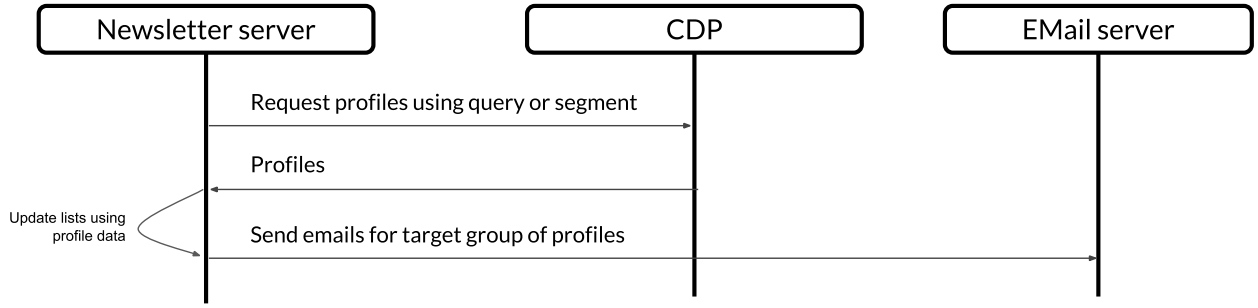


As illustrated above, the browser can interact with both a (headless or regular) Content Management System (CMS) and a CDP-compliant server to first retrieve the HTML needed to deliver the page content or bootstrap a Progressive Web Application (PWA). After this the next request to the CMS is a request for personalized content that will be customized based on the profile retrieved from the CDP. The result is personalized content for the current visitor being sent back to the browser.

This illustration is by no means the only way to implement personalization using a CDP but it serves as a simple introduction to the possibilies such a system may offer. Even native mobile applications could be integrated using this pattern.

2.4. Newsletters

This use case is relevant to users interested in delivering newsletters to the proper audience. For example it might be interesting to send a newsletter to promote a product to a group of profiles that has not purchased the product before, but it would not be a good idea to send it to people that have already purchased it.

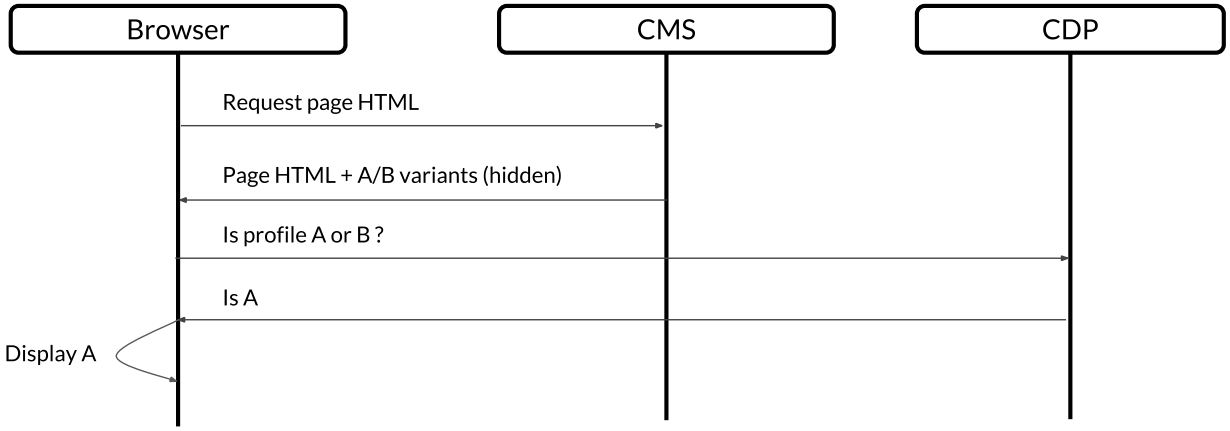


In the above illustration the newsletter server can query the CDP for a group of profile using either a query or a pre- defined segment to retrieve the subset of profiles it is interested in. Once those profiles are retrieved they may be used to update the newsletters management system lists with information coming from those exported profiles. And finally, when the newsletter is ready to be distributed, the updated lists may be used to send the emails using an email delivery server or service.

This use case could be expanded to use segmentation, campaigns and other Marketing Automation technologies that could benefit from the standardized functionalities exposed by this specification.

2.5. A/B testing

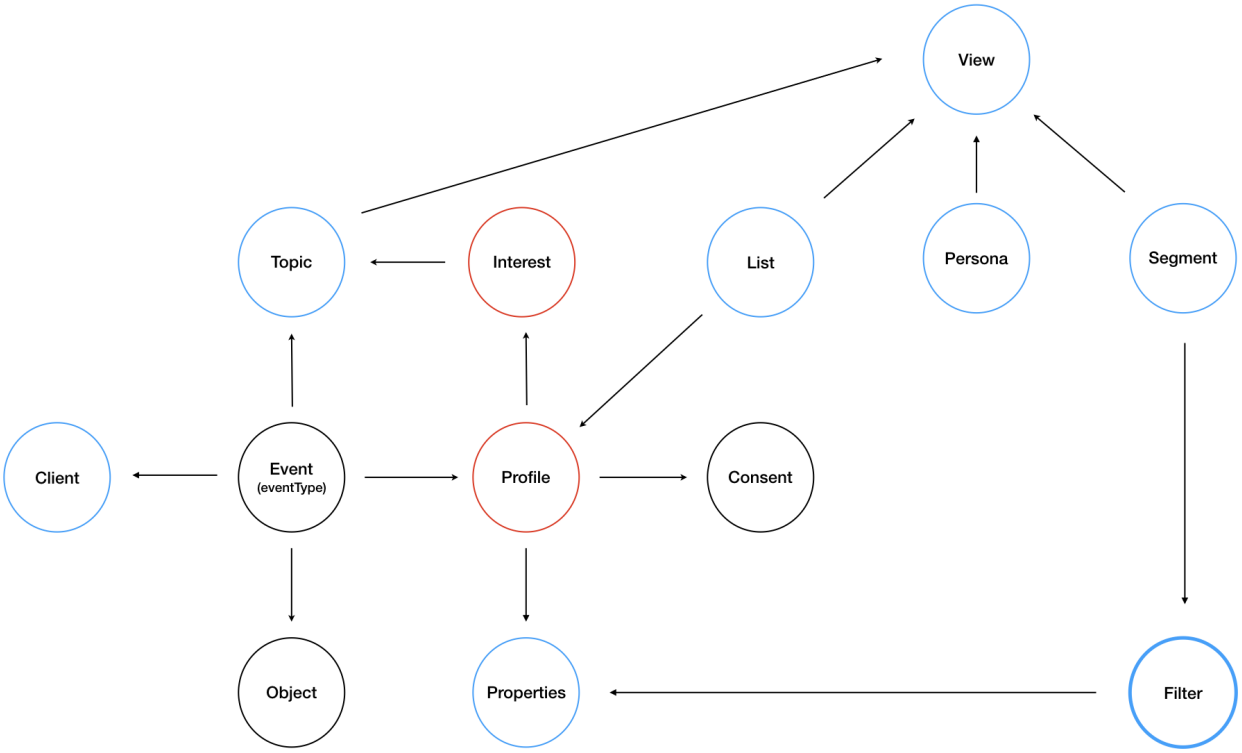
CDP systems may also be used to deliver A/B testing experiences. In this use case, the CDP server will use the visitor profile information by updating it with the variants that the visitor has been exposed to, effectively "classifying" the visitor into a sub-group.



In the above illustration, this use case is implemented by using a CMS to deliver the different variants of content that are hidden by default. After that, the CDP is asked whether the profile is in variant A or B, which might be implemented in different fashions but they will be remembered by the CDP for future displays.

3. Domain objects

Below is a short introduction to the core domain objects of the specification:



NOTE	Blue objects are typically manually configured and managed, red objects are generated.

Events

Events represent the stream of "customer behaviour" events that help the CDP build Profiles

Profiles

Representing the data of the subject, or "customer" interacting with your business Objects

Personas

Use Personas to simulate real Profiles for testing and validation

Objects

Are the physical or virtual items/persons a "customer" interacts with through events

Lists

Manually or programatically managed Lists of profiles

Segments

Segments are lists of profiles defined by Filters

Consents

Consents granted or denied by the subjects Properties: Enable the definition of custom profile Properties within a CDP deployment

Clients

Clients represent any entity connecting to CDP, either for storing or retrieving data

Views

Administrative Views are used for grouping managed entities, i.e. lists and segments

Interest

A profile’s weighted Interests for specific Topics

Topics

Represent "business areas" of the organisation deploying the CDP - i.e. a product or a location.

Filters

Filters are structured queries against other CDP domain objects

4. API

The Customer Data Platform (CDP) standard is built around a set of concepts, domain objects and services for interacting with them. This is represented through a strongly typed API defined by GraphQL Types, Queries, Mutations and Subscriptions.

Each section in the API reference will usually start with a description of the domain objects and

then include the normative GraphQL types, queries and mutations relevant to the domain objects.

This chapter describes the API in detail.

GraphQL requests are usually composed of two parts : `operations` and `variables`.

Throughout this document we will provide GraphQL request examples in the following form:

operation

GraphQL query, mutation or subscription

variables

JSON structure

Example operation

```
query getExistingProfile($profileID : CDP_ProfileIDInput) {
  cdp {
    getProfile(createIfMissing: false, profileID: $profileID) {
      cdp_profileIDs {
        id
      }
    }
  }
}
```

The above query retrieves all profileIDs for an existing profile (that's why we set the createIfMissing argument to false). We also define a variable called $profileID that must be passed in the "variables" section of the GraphQL request. Here's an example of the `variables part:

Example variables

```
{
  "profileID": {
    "clientID": "web-tracker",
    "id": "0bb99ae7-0571-4b5f-8267-978731cb62c2"
  }
}
```

As illustrated above, the variables may contain complex JSON structure that represent the values for the objects that are passed as GraphQL arguments.

4.1. GraphQL limitations and workarounds

No support for inheritance in input types

as a workaround a wrapper type is used that contains fields for all the different possible sub-object types. The CDP_PropertyInput type is an example of such a workaround

Namespacing

all types are prefixed with the `CDP_` prefix to avoid conflicts with custom-defined types. Also for types that may mix CDP standard and user-defined fields, a `cdp_` prefix has been used.

Dynamic API

as parts of the API are generated from user-defined properties or event types, removal of fields and types may happen dynamically. Implementors should advise users about this or create workarounds (such as depreciation).

4.2. Scalars

GraphQL provides several basic value types that are used extensively in this specification, for instance Int and String. However, the CDP specification is also handling other value types in a similar fashion.

The following scalars have been specifically added:

4.2.1. JSON

For values and arguments that cannot be defined structurally

```
Scalar JSON
```

4.2.2. Date

For consistent representation of dates. Based on RFC-3339, for example 1996-12-19, see https://github.com/graphql-java/graphql-java-extended-scalars for example implementation

```
Scalar Date
```

4.2.3. Time

For consistent representation of time. Based on RFC-3339, for example 16:39:57-08:00, see https://github.com/graphql-java/graphql-java-extended-scalars for example implementation

```
Scalar Time
```

4.2.4. DateTime

For consistent representation of date and time. Based on RFC-3339, for example 1996-12-19T16:39:57-08:00, see https://github.com/graphql-java/graphql-java-extended-scalars for example implementation

```
Scalar DateTime
```

4.2.5. GeoPoint

Uses a string representation of lat,lon

```
Scalar GeoPoint
```

4.3. Properties

To properly store and query data CDP needs a way to describe the data dynamically.

A Property represents data stored in a key-value format. A single property can hold a single value, or an ordered array of values. Each property has a specific valueType to limit what kind of values it may hold, such as `Identifier`, `String` and `Int`.

Below are some examples of properties:

- fullName(String) : "Jane Doe"
- birthDate(Date) : "2003-07-01"
- someInteger(Integer) : 1337
- gender(Enumeration) : FEMALE
- location(GeoPoint) : "lat,lon"
- arrayOfStrings([String]) : ["This", "is", "nice"]
- setOfProperties(Set) : {"prompt" : "hello", "response" : "yo"}
- arrayOfSet([Set]) : [{"name1": "value1", "name2" : "value2"}], [{"name1": "value1", "name2" : "value2"}]

In the case of the enumeration value type, a GraphQL enum type will be generated based on the registered possible values for the property.

The Set value type is special, as it enables nested properties and a tree-structure of properties. I.e. from the example above: "setOfProperties.response" would hold the value "yo"

The arrayOfSet given as an example above is simply a Set property type with multiple values (see occurrences defined below).

A property consists of:

Property name
: it is recommended but not mandatory to prefix the property name

Value type
: One of `Identifier`, `String`, `Int`, `Float`, `Date`, `Boolean`, `GeoPoint`, `Enumeration` and `Set`

Minimum occurrences

Minimum number of values a property may hold (array)

Maximum occurrences

Maximum number of values per property

Tags

A tag may be used to annotate the property with metadata information such as "personalData", "requiredReadAuthorization".

Since the CDP api is defined using strongly typed GraphQL, the API is dynamically updated when properties are added or changed.

4.3.1. CDP_PropertyInterface

The property interface defines the common fields for the different value types.

```
interface CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
}
```

name

must be in a format that's acceptable as a GraphQL field name (/*[A-Za-z][_0-9A-Za-z]*/) , *we highly RECOMMEND to prefix it to avoid conflicts, i.e acme_pageView, acme_click. "cdp*" is reserved.

minOccurences

Default = 0. For minOccurrences > 1 the property can hold multiple values in preserved order. minOccurences = 1 indicates the property is mandatory.

maxOccurences

Default = 1. maxOccurences = 0 indicates no limit. maxOccurences must be higher than minOccurences.

tags

System defined/generated tags. E.g: hidden, readOnly, personalData

4.3.2. CDP_PropertyInput

A property type may have different values types, but due to a limitation of GraphQL Input types it is not possible to represent this using type inheritance. As a workaround, an input type containing all possible value types is used instead, and only one of these fields is allowed to have a value corresponding to the declared property value type. All other value fields must be null. It is REQUIRED that implementations check for this and return an error if invalid values are passed.

```
input CDP_PropertyInput {
  identifier : CDP_IdentifierPropertyInput
  string : CDP_StringPropertyInput
  int : CDP_IntPropertyInput
  float : CDP_FloatPropertyInput
  date : CDP_DatePropertyInput
  boolean : CDP_BooleanPropertyInput
  geopoint : CDP_GeoPointPropertyInput
  enum : CDP_EnumPropertyInput
  set : CDP_SetPropertyInput
}
```

4.3.3. CDP_BooleanProperty

```
type CDP_BooleanProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  defaultValue : Boolean
}
```

4.3.4. CDP_BooleanPropertyInput

```
input CDP_BooleanPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  defaultValue : Boolean
}
```

4.3.5. CDP_DateProperty

```
type CDP_DateProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  defaultValue : String
}
```

4.3.6. CDP_DatePropertyInput

```
input CDP_DatePropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  defaultValue : String
}
```

4.3.7. CDP_EnumProperty

```
type CDP_EnumProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  values : [String]
}
```

4.3.8. CDP_EnumPropertyInput

```
input CDP_EnumPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  values : [String]
}
```

4.3.9. CDP_FloatProperty

```
type CDP_FloatProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  minValue : Float
  maxValue : Float
  defaultValue : Float
}
```

4.3.10. CDP_FloatPropertyInput

```
input CDP_FloatPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  minValue : Float
  maxValue : Float
  defaultValue : Float
}
```

4.3.11. CDP_GeoPointProperty

```
type CDP_GeoPointProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  defaultValue : String
}
```

4.3.12. CDP_GeoPointPropertyInput

```
input CDP_GeoPointPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  defaultValue : String
}
```

4.3.13. CDP_IdentifierProperty

```
type CDP_IdentifierProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  regexp : String
  defaultValue : String
}
```

4.3.14. CDP_IdentifierPropertyInput

```
input CDP_IdentifierPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  regexp : String
  defaultValue : String
}
```

4.3.15. CDP_IntProperty

```
type CDP_IntProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  minValue : Int
  maxValue : Int
  defaultValue : Int
}
```

4.3.16. CDP_IntPropertyInput

```
input CDP_IntPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  minValue : Int
  maxValue : Int
  defaultValue : Int
}
```

4.3.17. CDP_StringProperty

```
type CDP_StringProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  regexp : String
  defaultValue : String
}
```

4.3.18. CDP_StringPropertyInput

```
input CDP_StringPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  regexp : String
  defaultValue : String
}
```

4.3.19. CDP_SetProperty

```
type CDP_SetProperty implements CDP_PropertyInterface {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  properties : [CDP_PropertyInterface]
}
```

4.3.20. CDP_SetPropertyInput

```
input CDP_SetPropertyInput {
  name : ID!
  minOccurrences : Int
  maxOccurrences : Int
  tags : [String]
  properties : [CDP_PropertyInput]
}
```

4.4. Filters

Filters are widely used in CDP, and enable querying profiles, events, and other CDP objects. Filters are designed to be easy to use for administrators and marketeers in visual user interfaces, but also in terms of technical implementation.

Filters are essentially composed from basic property comparison expressions, and may be chained with the operators AND and OR, where AND is used by default.

For each operator available on a property's value type a GraphQL field will be generated.

As we are expressing filters through GraphQL, filters will always be strongly typed. I.e. if the property "firstName" with valueType string is available, the following filter options can be used:

```
firstName_equals
firstName_startsWith (OPTIONAL)
firstName_endsWith (OPTIONAL)
firstName_contains
firstName_regexp (OPTIONAL)
```

Below are some basic filter examples:

```
{ "firstName_equals" : "Serge" }
```

```
{ "birthDate_greaterThan" : "1970-01-01" }
```

```
{
  "location_distance" : {
    "center" : { "longitude" : 59.91273, "latitude": 10.74609 },
    "unit" : "KILOMETERS",
    "distance" : 5
  }
}
```

```
{
  "or" : [
    { "firstName_equals" : "Serge" },
    { "birthDate_greaterThan" : "1970-01-01" }
  ]
}
```

GraphQL filter fields will be generated the following way:

```
PROPERTYNAME + "_" + OPERATOR
```

The following comparison operators are available:

Table 1. Operator availability for property value types

Operators	Identifier	String	Int	Float	Date	Boolean	GeoPoint	Enumeration	Array
equals	x	x	x	x	x	x	x	x	
startsWith		x[o]							
endsWith		x[o]							

Operators	Identifier	String	Int	Float	Date	Boolean	GeoPoint	Enumeration	Array
contains		x[o]							x
regexp		x[o]							
lt			x	x	x				
lte			x	x	x				
gt			x	x	x				
gte			x	x	x				
distance							x		

[o] OPTIONAL operator

The `Array` column is a special case. It can be an array of any GraphQL type. In this case only the `contains` operator is defined in the specification, but implementations are free to offer more advanced operators for this type.

4.4.1. Ordering

OrderBy is used in combination with filters and lets you sort the result based on properties available for the returned objects.

Example:

```
"orderBy": [{
  "property": "firstName",
  "order": "ASC"
  }
]
```

4.4.2. CDP_SortOrder

Enumeration of allowed sorting operators

```
enum CDP_SortOrder {
  ASC,
  DESC,
  UNSPECIFIED
}
```

4.4.3. CDP_OrderByInput

```
input CDP_OrderByInput {
  fieldName : String
  order : CDP_SortOrder
}
```

fieldName

Specify the field to sort by, i.e. "endTime", "properties.location"

4.4.4. CDP_DateFilter

```
type CDP_DateFilter {
  after : DateTime
  before : DateTime
  includeAfter : Boolean
  includeBefore : Boolean
}
```

4.4.5. CDP_DateFilterInput

```
input CDP_DateFilterInput {
  after : DateTime
  before : DateTime
  includeAfter : Boolean
  includeBefore : Boolean
}
```

4.4.6. CDP_GeoDistanceFilterUnit

```
enum CDP_GeoDistanceFilterUnit {
  METERS,
  KILOMETERS,
  MILES
}
```

4.4.7. CDP_GeoDistanceFilter

```
type CDP_GeoDistanceFilter {
  center : GeoPoint
  unit : CDP_GeoDistanceFilterUnit
  distance : Float
}
```

4.4.8. CDP_GeoDistanceFilterInput

```
input CDP_GeoDistanceFilterInput {
  center : GeoPoint
  unit : CDP_GeoDistanceFilterUnit
  distance : Float
}
```

4.5. Clients

The CDP GraphQL API should only be accessible for specific authorized clients.

Client represent any software that interacts directly with the Customer Data Platform.

Examples of clients are:

- Cookie-based (Javascript or other) tracker for website(s)
- Integration with your CRM
- Integration with your Identity System

Each Client is responsible for uniquely identifying visitors, for instance through the use of a cookie on the website, a customer ID in the CRM or a user ID in the Identity system. The Customer Data Platform requires [profileIDs] to be unique within every client. For instance, if a client is used to track visitors across multiple websites, it should aim to re-use the same [profileID] across all of them, for the same visitor.

NOTE	The standard does not specify Queries or Mutations for creating or retrieving Clients in the CDP specification, as this is considered an implementation-specific feature. For any CDP implementation, a Client MUST be defined for it to access the API.

4.5.1. CDP_Client

```
type CDP_Client {
    id : ID!
    title : String
}
```

4.5.2. CDP_ClientInput

```
input CDP_ClientInput {
    id : ID!
    title : String
}
```

4.6. Sources

Sources are optional, but represent a way to identify the exact origin of events within a client. For instance, a web tracking script may track visitors across many different sites, but treat each site as a source. As such, sources are comparable to siteID's in Google Analytics.

Sources may be reused across clients as desired.

4.6.1. CDP_Source

```
type CDP_Source {
    id : ID!
    thirdParty : Boolean
}
```

4.6.2. CDP_SourceInput

```
input CDP_SourceInput {
    id : ID!
    thirdParty : Boolean
}
```

id

The "system" source ID is reserved for internal use by the CDP.

thirdParty

Optional, indicates that the source is a third party (useful for privacy regulations such as GDPR)

4.6.3. CDP_Query

Source related queries

```
  getSources : [CDP_Source]
```

4.6.4. CDP_Mutation

Source related mutations

```
  createOrUpdateSource(source : CDP_SourceInput) : CDP_Source
  deleteSource(sourceID : ID!) : Boolean
```

4.7. Objects

Objects are representations of anything users interact with. For example: a web page, a product or another person. Objects are used in Events to specify what the Profiles are interacting with. Objects

are also used in Optimizations.

Objects may be part of one or more collections. Collections are used to classify objects. By placing objects into collections, optimizations may execute on a reduced data set (i.e. : recommending products).

4.7.1. URIs

Objects are identified globally using URIs, that follow the URI specification (https://tools.ietf.org/html/rfc3986). Internal CDP objects may be referenced using reserved schemes, that each have their associated syntax:

- cdp_profile:client/id
- cdp_segment:view/name
- cdp_persona:view/name
- cdp_topic:view/name
- cdp_list:view/name

4.7.2. CDP_Object

```
type CDP_Object {
    uri : ID! # uri format : scheme:path, https://tools.ietf.org/html/rfc3986
    scheme : String
    path : String
    topics : [CDP_Topic]
}
```

uri

Globally unique identifier using URI syntax : https://tools.ietf.org/html/rfc3986

topics

A way of classifying objects.

4.7.3. CDP_ObjectInput

```
input CDP_ObjectInput {
    uri : ID!
}
```

4.8. Events

Events are what drives the Customer Data Platform forward. Events are collected from different Clients, such as a specific website, beacons, commerce systems or a CRM.

A single Client might still produce many different profiles for a "real person". For instance - if a

visitor uses different devices on a single web page, each device will produce a new profile, with a unique profileID.

The Customer Data Platform is essentially interested in "User behavioral events". An event could be anything from someone clicking a link, to performing a transaction or consenting to use of his/hers information. Events are streamed or delivered from authorized Clients to the Customer Data Platform.

As an example: Imagine an e-commerce site with a client that collect events from its visitors. When a visitor browses the site with his laptop, the client assigns a cookie to his/her browser and starts feeding events to the CDP API. As the visitor click on some product links, and maybe fills in a form that includes e-mail. CDP will gradually populate a profile, using the cookie value as an ID. At a later point, the same visitor picks up a different device and returns to the site. As the client cannot know this is the same individual, a new cookie is generated, and a new profile starts to build up.

A single client may be used to track Events from a number of different websites, where each website can be tagged with a source. Sources provide a way to identify the exact origin of the events beyond the client. As such, sources are comparable to siteID's in Google Analytics.

4.8.1. EventTypes

Events must always be of a specific type. CDP implementations must implement a set of standard EventTypes, any other EventTypes are implementation specific.

For flexibility reasons, implementers are encouraged to make EventTypes pluggable. Implementation specific, or pluggable EventTypes SHOULD be registered with a prefix, to avoid naming conflicts. All standard EventTypes will be prefixed with CDP.

In the CDP, every EventType will need both an regular GraphQL type, and a GraphQL input.

NOTE	EventType fields MUST match the CDP propertyType format, and its underlying valueTypes

When custom EventTypes are registered in a server, new corresponding fields will be added to the in the CDP_EventInput type and the convention is that the field name is the same as the type name but starting with a lowercase character instead of an uppercase one.

Custom output event types must also inherit from the CDP_EventInterface interface.

Below are examples of what custom EventTypes might look like, we used VENDOR as prefix, since it is recommended to avoid conflicts with other types:

Sample EventType for Page Views

```
input VENDOR_PageViewEventInput {
  pageID : String,
  language : String,
  pageUrl : String,
  referrer : String,
  userAgent : String
}

type VENDOR_PageViewEvent implements CDP_EventInterface {
  # The following fields come from the EventInterface
  id: ID!
  cdp_source : CDP_Source
  cdp_client : CDP_Client
  cdp_profileID: CDP_ProfileID!
  cdp_profile : CDP_Profile!
  cdp_object: CDP_Object!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [CDP_Topic]
  # The following fields are specific to this event type
  pageID : String,
  language : String,
  pageUrl : String,
  referrer : String,
  userAgent : String
}
```

The CDP_EventInput type will therefore modified to add the new event type-specific field.

Sample EventType for Page Views

```
input CDP_EventInput {
  id: ID
  cdp_sourceID : String
  cdp_profileID: CDP_ProfileIDInput!
  cdp_objectID: ID!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [ID]
  cdp_profileUpdateEvent : CDP_ProfileUpdateEventInput
  cdp_consentUpdateEvent : CDP_ConsentUpdateEventInput
  cdp_listsUpdateEvent : CDP_ListsUpdateEventInput
  cdp_sessionEvent : CDP_SessionEventInput

  vENDORPageViewEvent : VENDOR_PageViewEventInput # note the lowercase v as a
convention as well as the absence of the "Input" suffix.
}
```

Once the event types are defined (in an implementation-specific manner), they can be sent using the `processEvents` mutation field that uses the CDP_EventInput type and queried using the `findEvents` query field

Sample EventType for CRM updates

```
input VENDOR_CrmLeadEventInput {
  leadStatus : String,
  leadID : String,
  firstName : String,
  lastName : String,
  email : String
}

type VENDOR_CrmLeadEvent implements CDP_EventInterface {
  # The following fields come from the EventInterface
  id: ID!
  cdp_source : CDP_Source
  cdp_client : CDP_Client
  cdp_profileID: CDP_ProfileID!
  cdp_profile : CDP_Profile!
  cdp_object: CDP_Object!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [CDP_Topic]
  # The following fields are specific to this event type
  leadStatus : String,
  leadID : String,
  firstName : String,
  lastName : String,
  email : String
}
```

The CDP_EventInput type will also have a new field called `vENDOR_CrmLeadEvent : VENDOR_CrmLeadEventInput` as in the previous example.

Standard event types:

- Updating profile properties, needs to match the profile properties definitions (built-in)
- Session start/paused/resumed/stopped (built-in)
- Updating consent (see http://ec.europa.eu/ipg/basics/legal/cookies/index_en.htm) (built-in)
- Opt-in / opt-out of a list (built-in)

Suggested event types:

- Transaction (generic)
- Like (“user likes a product”)
- Dislike (“visitor dislikes a comment”)

- Abuse, “user reports abuse on a page”
- Rate (score in percent) “user rates product 4 out of 5 stars”
- Vote
- Download (“user downloaded a digital product”)
- Register/Submission
- Login
- Logout
- RequestFriendship
- AcceptFriendship
- DenyFriendship
- Click
- View
- Contribute (comment, blog etc?)
- Conversion (purchase, download, signs up for a service

4.8.2. CDP_EventInterface

Events make use of type inheritance. To avoid name space conflicts, all standard event fields are prefixed with ´´_´´.

```
interface CDP_EventInterface {
  id: ID!
  cdp_source : CDP_Source
  cdp_client : CDP_Client
  cdp_profileID: CDP_ProfileID!
  cdp_profile : CDP_Profile!
  cdp_object: CDP_Object!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [CDP_Topic]
}
```

4.8.3. CDP_EventInput

```
input CDP_EventInput {
  id: ID
  cdp_sourceID : String
  cdp_profileID: CDP_ProfileIDInput!
  cdp_objectID: ID!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [ID]
  cdp_profileUpdateEvent : CDP_ProfileUpdateEventInput
  cdp_consentUpdateEvent : CDP_ConsentUpdateEventInput
  cdp_listsUpdateEvent : CDP_ListsUpdateEventInput
  cdp_sessionEvent : CDP_SessionEventInput
  # Sample custom EventTypes below:
  # my_pageView : MY_PageViewEventInput
  # my_addedToCart : MY_addedToCartEventInput,
  # other_crmUpdate : OTHER_crmUpdateEventInput
}
```

4.8.4. CDP_Query

Event queries

```
  getEvent(id : String!) : CDP_EventInterface
  findEvents(filter : CDP_EventFilterInput, orderBy : [CDP_OrderByInput], first: Int,
after: String, last: Int, before: String) : CDP_EventConnection
```

4.8.5. CDP_Mutation

Event mutations

```
  processEvents(events: [CDP_EventInput]!) : Int
```

4.8.6. CDP_Subscriptions

Event subscriptions

```
  eventListener(filter: CDP_EventFilterInput) : CDP_EventInterface!
```

4.8.7. Event processing sample

Mutation

```
mutation profileUpdateExample($events: [CDP_EventInput]!) {
  cdp {
    processEvents(events: $events)
  }
}
```

Mutation variables

```
{"events": [
  {
    "_profileID": {
      "id": "1234567890",
      "clientID": "web-tracker"
    },
    "_objectID": "http://acme.org/aboutUs",
    "pageViewEvent": {
      "language" : "en"
    }
  }
]}
```

4.9. EventFilters

EventFilters are a specific version of filters for querying events.

Example: Filter for identifying events of type `_profileUpdate` with a first name starting with `T` and a last name ending with `d` within the last 30 days.

Operation

```
query findEvents($filter: CDP_EventFilterInput) {
  cdp {
    findEvents(filter: $filter) {
      edges {
        node {
          __typename
          cdp_timestamp
          cdp_object {
            uri
          }
        }
      }
    }
  }
}
```

Variables

```
{
  "filter" : {
    "_timestamp_between" : {
      "after" : "NOW-30DAYS",
      "before" : "NOW",
      "includeBefore": false,
      "includeAfter": false
    },
      "_profileUpdateEvent" : {
      "firstName_startsWith" : "T",
      "lastName_endsWith" : "d"
    }
  }
}
```

4.9.1. CDP_EventFilter

```
type CDP_EventFilter {
  and : [CDP_EventFilter]
  or : [CDP_EventFilter]
  id_equals : String
  cdp_clientID_equals: String
  cdp_sourceID_equals : String
  cdp_profileID_equals : String
  cdp_objectID_equals : String
  cdp_location_distance : CDP_GeoDistanceFilter
  cdp_timestamp_equals : DateTime
  cdp_timestamp_lt : DateTime
  cdp_timestamp_lte : DateTime
  cdp_timestamp_gt : DateTime
  cdp_timestamp_gte : DateTime
  cdp_topics_equals : String
  cdp_profileUpdateEvent : CDP_ProfileUpdateEventFilter
  cdp_consentUpdateEvent : CDP_ConsentUpdateEventFilter
  cdp_listsUpdateEvent : CDP_ListsUpdateEventFilter
  cdp_sessionEvent : CDP_SessionEventFilter
  # generated event types will be listed here
}
```

4.9.2. CDP_EventFilterInput

```
input CDP_EventFilterInput {
  and : [CDP_EventFilterInput]
  or : [CDP_EventFilterInput]
  id_equals : String
  cdp_clientID_equals: String
  cdp_sourceID_equals : String
  cdp_profileID_equals : String
  cdp_objectID_equals : String
  cdp_location_distance : CDP_GeoDistanceFilterInput
  cdp_timestamp_equals : DateTime
  cdp_timestamp_lt : DateTime
  cdp_timestamp_lte : DateTime
  cdp_timestamp_gt : DateTime
  cdp_timestamp_gte : DateTime
  cdp_profileUpdateEvent : CDP_ProfileUpdateEventFilterInput
  cdp_consentUpdateEvent : CDP_ConsentUpdateEventFilterInput
  cdp_listsUpdateEvent : CDP_ListsUpdateEventFilterInput
  cdp_sessionEvent : CDP_SessionEventFilterInput
  # generated event types will be listed here
}
```

4.10. Profiles

Profiles are in many ways the holy grail of CDP. The Customer Data Platform dynamically creates and build profiles from events that occur over time.

A Profile can be created from an anonymous visitor on a webpage, populated from an identity system, a CRM, or the combination of all of them.

Different Clients like a website tracking script, CRM or identity system can be configured to feed Events to the Customer Data Platform.

The Customer Data Platform is responsible for building profiles based on the provided identifiers and the stream of events coming from each Client.

4.10.1. Profile properties

Each deployment of CDP will be unique in how data are collected, and what data is stored per profile. Profile properties enable us to define custom properties required by an organization.

Administrators and developers may define and maintain a consistent data model for profiles across different Clients. Any data to be recorded in a profile must be mapped to a corresponding profile property.

The specification does not define a set of standard profile properties. However, implementors SHOULD include the following standard properties :

- fullName : string

- email : identifier
- phoneNumber : identifier
- birthday : datetime # this can be very important for managing consents, or managing any approval for children, minors, etc...

Profiles are updated through events. The history of external or internal profile modifications is accessible through the profile update events. CDP implementations SHOULD also support subscriptions on profile modifications so that external systems can retrieve the profile modifications in real-time.

Properties can be dynamically defined for profiles using the `createOrUpdateProfileProperties` and `deleteProfileProperties` mutations. Once a property is associated with a profile, it will become available in the CDP_Profile and CDP_Persona types.

NOTE	It is the responsability of clients accessing the GraphQL API to handle the lifecycle of properties properly, as new properties may be defined at any time, or more importantly, properties may be also deleted, potentially breaking a client's use of the API.

As an example, let's assume we have a starting CDP_Profile type that looks like this:

Profile before

```
type CDP_Profile implements CDP_ProfileInterface {
  cdp_profileIDs : [CDP_ProfileID]
  cdp_events(filter : CDP_EventFilterInput, first : Int, last: Int, after : String,
before: String) : CDP_EventConnection
  cdp_lastEvents(count : Int, profileID : CDP_ProfileIDInput) : CDP_EventConnection
  cdp_segments(views : [ID]) : [CDP_Segment]
  cdp_interests(views : [ID]) : [CDP_Interest]
  cdp_consents : [CDP_Consent]
  cdp_lists(views : [ID]) : [CDP_List]
  cdp_matches(namedFilters : [CDP_NamedFilterInput]) : [CDP_FilterMatch]
  cdp_optimize(parameters : [CDP_OptimizationInput]) : [CDP_OptimizationResult]
  cdp_recommend(parameters : [CDP_RecommendationInput]) : [CDP_RecommendationResult]
  # fields will be added here according to registered profile properties
}
```

Now let's use the mutation to create a new property.

Operation

```
mutation addProperties($properties: [CDP_PropertyInput]) {
  cdp {
    createOrUpdateProfileProperties(properties: $properties)
  }
}
```

Variables

```
{
  "properties": [
    {
      "string": {
        "name": "firstName"
      }
    },
    {
      "set": {
        "name": "sample_Address",
        "properties": {
          "string": {"name": "streetName"},
          "string": {"name": "postalCode"}
        }
      }
    }
  ]
}
```

This will resulting in the following modifications to the CDP_Profile type:

Profile after

```
type CDP_Profile implements CDP_ProfileInterface {
  cdp_profileIDs : [CDP_ProfileID]
  cdp_events(filter : CDP_EventFilterInput, first : Int, last: Int, after : String,
before: String) : CDP_EventConnection
  cdp_lastEvents(count : Int, profileID : CDP_ProfileIDInput) : CDP_EventConnection
  cdp_segments(views : [ID]) : [CDP_Segment]
  cdp_interests(views : [ID]) : [CDP_Interest]
  cdp_consents : [CDP_Consent]
  cdp_lists(views : [ID]) : [CDP_List]
  cdp_matches(namedFilters : [CDP_NamedFilterInput]) : [CDP_FilterMatch]
  cdp_optimize(parameters : [CDP_OptimizationInput]) : [CDP_OptimizationResult]
  cdp_recommend(parameters : [CDP_RecommendationInput]) : [CDP_RecommendationResult]
  # fields will be added here according to registered profile properties
  firstName : String
  sample_Address : Sample_Address
}
```

The following type is generated from the property definition. The name of the type starts with an uppercased character from the property name.

Generated type

```
type Sample_Address {
  streetName : String,
  postalCode : String
}
```

This will also generate new filter fields in the CDP_ProfilePropertiesFilterInput type:

Updated filters

```
type CDP_ProfilePropertiesFilter {
  and : [CDP_ProfilePropertiesFilter]
  or : [CDP_ProfilePropertiesFilter]
  # generated profile properties filters will be listed below
  firstName_equals : String,
  firstName_contains : String
  sample_Address : Sample_AddressFilter
}

type Sample_AddressFilter {
  streetName_equals : String,
  streetName_contains : String,
  postalCode_equals : String,
  streetName_contains : String
}

input CDP_ProfilePropertiesFilterInput {
  and : [CDP_ProfilePropertiesFilterInput]
  or : CDP_ProfilePropertiesFilterInput
  # generated profile properties filters will be listed below
  firstName_equals : String,
  firstName_contains : String
  sample_Address : Sample_AddressFilterInput
}

input Sample_AddressFilterInput {
  streetName_equals : String,
  streetName_contains : String,
  postalCode_equals : String,
  streetName_contains : String
}
```

As you can see the generation system also creates filter types (input and output) and adds the "Filter" and "FilterInput" suffix to them. This will always happen and implementations MUST do this.

Also not illustrated here, the same generation system will also add fields to the following types :

- CDP_ProfileUpdateEvent

- CDP_ProfileUpdateEventInput
- CDP_ProfileUpdateEventFilter
- CDP_ProfileUpdateEventFilterInput
- CDP_Persona
- CDP_PersonaInput

The naming and generation conventions are exactly the same as for the profiles properties.

4.10.2. Profile merges

Customer Data Platforms implementations MUST support profile merges.

As profiles evolve over time, the Customer Data Platform may discover that two profiles actually represent the same individual. I.e. if the same e-mail address is registered in both two different profiles.

This may then result in a profile merge. During a profile merge, the Customer Data Platform will link two (or more) separate profiles together. In order to keep event history and avoid re-processing of data, the merge process must not affect the existing and unique profileIDs. This is why profiles are defined to have multiple profileIDs.

Example: As such, when visitors on a website are tracked through a cookie (defining the profileID), the cookie will remain the same even if the profile is merged.

Profile merges may for instance be supported by using identifying profile properties (such as email and/or social security number). The resulting merged profile MUST contain all the profile IDs of the merged profiles, as well as the merged profile data. The original profiles that were merged may be flagged or deleted, this is implementation specific.

4.10.3. Deleting profile personal data (aka profile anonymizing)

The API provides a way to delete personal data associated with a profile. The effect of this operation is not specified in details but it should respect existing privacy laws such as GDPR. For example, it could remove all properties flagged as containing personal data and/or it could even process events in ways to anonymize data.

4.10.4. CDP_ProfileID

Profiles are created from a client. As such, each profile has a composite key based on a unique ID within that client, and the client.

```
type CDP_ProfileID {
    client : CDP_Client!
    id : ID!
    uri : ID # "cdp_profile:source/id"
}
```

4.10.5. CDP_ProfileIDInput

```
input CDP_ProfileIDInput {
    clientID : ID!
    id : ID!
}
```

- **id** ID must be unique within the client

4.10.6. CDP_ProfileInterface

Common interface for Profiles and Personas

```
interface CDP_ProfileInterface {
  cdp_profileIDs : [CDP_ProfileID]
  cdp_segments(views : [ID]) : [CDP_Segment]
  cdp_interests(views : [ID]) : [CDP_Interest]
  cdp_consents : [CDP_Consent]
  cdp_lists(views : [ID]) : [CDP_List]
}
```

profileIDs

A single profile may consist of multiple id's as profiles are being merged. The CDP may also generate a system profile ID and expose it here

4.10.7. CDP_Profile

```
type CDP_Profile implements CDP_ProfileInterface {
  cdp_profileIDs : [CDP_ProfileID]
  cdp_events(filter : CDP_EventFilterInput, first : Int, last: Int, after : String,
before: String) : CDP_EventConnection
  cdp_lastEvents(count : Int, profileID : CDP_ProfileIDInput) : CDP_EventConnection
  cdp_segments(views : [ID]) : [CDP_Segment]
  cdp_interests(views : [ID]) : [CDP_Interest]
  cdp_consents : [CDP_Consent]
  cdp_lists(views : [ID]) : [CDP_List]
  cdp_matches(namedFilters : [CDP_NamedFilterInput]) : [CDP_FilterMatch]
  cdp_optimize(parameters : [CDP_OptimizationInput]) : [CDP_OptimizationResult]
  cdp_recommend(parameters : [CDP_RecommendationInput]) : [CDP_RecommendationResult]
  # fields will be added here according to registered profile properties
}
```

4.10.8. CDP_ProfileUpdateEvent

Profiles are crated and updated through this event type. This event is part of the standard and MUST be available for any implementation of the specification.

```
type CDP_ProfileUpdateEvent implements CDP_EventInterface {
  id: ID!
  cdp_source : CDP_Source
  cdp_client : CDP_Client
  cdp_profileID: CDP_ProfileID!
  cdp_profile : CDP_Profile!
  cdp_object: CDP_Object!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [CDP_Topic]
  # fields will be added here according to registered profile properties. To remove a
property value pass a null value
}
```

4.10.9. CDP_ProfileUpdateEventInput

This is the input equivalent, notice because of missing input type inheritance in GraphQL, it only contains the actual properties to update.

Operation

```
mutation updateProfile($events: [CDP_EventInput]!) {
  cdp {
    processEvents(events: $events)
  }
}
```

Variables

```
{"events": [
  {
    "_object" : "cdp_profile:crm/crm-profile-id",
    "_profileID": {
      "clientID": "crm",
      "id": "crm-profile-id"
    },
    "_profileUpdateEvent": {
      "firstName" : "Serge",
      "sample_Address" : {
        "streetName" : "My street name",
        "postalCode" : "12345"
      }
    }
  }
]}
```

4.10.10. CDP_ProfileUpdateEventFilter

Sample ProfileUpdateEventFilter

```
input CDP_ProfileUpdateEventInput {
  firstname : String
  dateofbirth : Date
  # more fields will be available based on defined profile properties
}
```

4.10.11. CDP_ProfileUpdateEventFilterInput

Sample ProfileUpdateEventFilterInput

```
input CDP_ProfileUpdateEventInput {
  firstname : String
  dateofbirth : Date
  # more fields will be available based on defined profile properties
}
```

4.10.12. CDP_Query

Profile queries

```
  getProfile(profileID : CDP_ProfileIDInput, createIfMissing: Boolean) : CDP_Profile
  findProfiles(filter: CDP_ProfileFilterInput, orderBy: [CDP_OrderByInput], first:
Int, after: String, last: Int, before: String) : CDP_ProfileConnection
  getProfileProperties : CDP_PropertyConnection
```

4.10.13. CDP_Mutation

The profile property mutation fields (createOrUpdateProfileProperties, deleteProfileProperties) are OPTIONAL (see Conformance section).

Profile mutations

```
  createOrUpdateProfileProperties(properties : [CDP_PropertyInput]) : Boolean
  deleteProfileProperties(propertyNames : [ID]!) : Boolean
  deleteProfile(profileID : CDP_ProfileIDInput) : CDP_Profile
  deleteAllPersonalData(profileID : CDP_ProfileIDInput) : Boolean
```

4.10.14. CDP_Subscription

Profile subscriptions

```
extend type CDP_Subscription {
```

4.11. ProfileFilters

Profile Filters are slightly more complex than EventFilters. As profileFilter are composed from both searching profile properties, and events related to the profile.

Here is an example of a GraphQL query (with variables) that will retrieve profiles that "have joined the list with the id NEWSLETTER-LIST-ID since June 28th, 2018 at 5:25"

Operation

```
query profileFilterExample(
  $profileFilter: CDP_ProfileFilterInput
  $orderBy: [CDP_OrderByInput]
) {
  cdp {
    findProfiles(filter: $profileFilter, orderBy: $orderBy, first : 10) {
      totalCount
      edges {
        node {
          cdp_profileIDs {
            client {
              id
            }
            id
          }
          cdp_segments {
            name
          }
        }
      }
    }
  }
}
```

Variables

```
{
  "profileFilter": {
    "lists_contains" : [ "NEWSLETTER-LIST-ID" ],
    "properties": {},
    "events": {
      "minimalCount": 1,
      "eventFilter": {
        "_timestamp_gt": "2018-06-28T05:25:28+00:00",
        "_listsUpdateEvent": {
          "joinLists_contains" : ["NEWSLETTER-LIST-ID"]
        }
      }
    }
  },
  "orderBy": [
    {"fieldName": "properties.firstName", "order": "ASC"}
  ]
}
```

4.11.1. CDP_ProfileFilter

```
type CDP_ProfileFilter {
  profileIDs : [String]
  properties : CDP_ProfilePropertiesFilter
  segments_contains : [ID]
  consents_contains : [ID]
  lists_contains : [ID]
  interests : CDP_InterestFilter
  events : CDP_ProfileEventsFilter
}
```

4.11.2. CDP_ProfileFilterInput

```
input CDP_ProfileFilterInput {
  profileIDs_contains : [String]
  properties : CDP_ProfilePropertiesFilterInput
  segments_contains : [ID]
  consents_contains : [ID]
  lists_contains: [ID]
  interests : CDP_InterestFilterInput
  events : CDP_ProfileEventsFilterInput
}
```

4.11.3. CDP_ProfilePropertiesFilter

```
type CDP_ProfilePropertiesFilter {
  and : [CDP_ProfilePropertiesFilter]
  or : [CDP_ProfilePropertiesFilter]
  # generated profile properties filters will be listed below
}
```

4.11.4. CDP_ProfilePropertiesFilterInput

```
input CDP_ProfilePropertiesFilterInput {
  and : [CDP_ProfilePropertiesFilterInput]
  or : CDP_ProfilePropertiesFilterInput
  # generated profile properties filters will be listed below
}
```

4.11.5. CDP_ProfileEventsFilter

```
type CDP_ProfileEventsFilter {
  and : [CDP_ProfileEventsFilter]
  or : [CDP_ProfileEventsFilter]
  not : CDP_ProfileEventsFilter
  minimalCount : Int,
  maximalCount : Int,
  eventFilter : CDP_EventFilter
}
```

4.11.6. CDP_ProfileEventsFilterInput

```
input CDP_ProfileEventsFilterInput {
  and : [CDP_ProfileEventsFilterInput]
  or : [CDP_ProfileEventsFilterInput]
  not : CDP_ProfileEventsFilterInput
  minimalCount : Int,
  maximalCount : Int,
  eventFilter : CDP_EventFilterInput
}
```

4.12. Sessions

When individuals interact, clients may enrich the data associated with interaction by specifying sessions. For instance, a session may start when a user loads a specific app, and end when he closes it.

The CDP_SessionEventInput is used to signify the beginning, pause, resume or end of a session.

4.12.1. CDP_SessionState

```
enum CDP_SessionState {
  START,
  STOP,
  PAUSE,
  RESUME
}
```

4.12.2. CDP_SessionEvent

```
type CDP_SessionEvent implements CDP_EventInterface {
  id: ID!
  cdp_source : CDP_Source
  cdp_client : CDP_Client
  cdp_profileID: CDP_ProfileID!
  cdp_profile : CDP_Profile!
  cdp_object: CDP_Object!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [CDP_Topic]
  state : CDP_SessionState
}
```

4.12.3. CDP_SessionEventInput

```
input CDP_SessionEventInput {
  state : CDP_SessionState
}
```

Example of how to update a session's state

Operation

```
mutation updateSessions($events: [CDP_EventInput]!) {
  cdp {
    processEvents(events: $events)
  }
}
```

Variables

```
{
  "events": [
    {
      "_profileID": {
        "clientID": "crm",
        "id" : "crm-profile-id"
      },
      "_object": "cdp_profile:crm/crm-profile-id",
      "_sessionEvent": {
        "state": "PAUSE"
      }
    }
  ]
}
```

4.12.4. CDP_SessionEventFilter

This type is used in EventFilters to filter session events

```
type CDP_SessionEventFilter {
  state_equals : CDP_SessionState
}
```

4.12.5. CDP_SessionEventFilterInput

This type is used in EventFilters to filter session events

```
input CDP_SessionEventFilterInput {
  state_equals : CDP_SessionState
}
```

4.13. Consents

New legislation and stricter rules for use of personal data is already here (i.e. GDPR). As such, consents are inherently more important to ensure you are using and storing data in compliance with policies.

Consents hold an identifier that uniquely identifies the consent across your systems.

Consents are given and revoked through events. This means that the CDP specification defines reserved property types for granting and revoking consents.

Sample GRANTED consent

```
    {
      "_sourceID" : "example.com",
      "_profileID": {
        "clientID": "crm",
        "id" : "crm-profile-id"
      },
      "_object": "cdp_profile:crm/crm-profile-id",
      "_consentUpdateEvent": {
        "type": "send-to-salesforce",
        "status": "GRANTED",
        "lastUpdate": "NOW",
        # no revoke date means it will not expire or defaults to system or legal
standard (GDPR)
      }
    }
```

Sample DENY consent

```
    {
      "_sourceID" : "example.com",
      "_profileID": {
        "clientID": "crm",
        "id" : "crm-profile-id"
      },
      "_object": "cdp_profile:crm/crm-profile-id",
      "_consentUpdateEvent": {
        "type": "newsletter-subscription-latestNews",
        "status": "DENY",
        "lastUpdate": "NOW",
        # no revoke date means it will not expire or defaults to system or legal
standard (GDPR)
      }
    }
```

Consent Types may include: - tracking - list membership - newsletter membership - access to camera - access to friends / contacts data - access to medical records - send sms - call you - send personal data to third parties - send anonymous data to third parties

Consent types are not defined in the specification, only the format of the type identifier should use a URI convention. Some URIs could actually be URLs and point to real resource that would give the semantics of the consent type. Types are not globally unique, a combination of view and types are globally unique and context server implementations may use "global" or "system" views to share types.

It is not in the scope of this specification to define how authentication and consents interact but it is expect that CDP implementations secure consent modifications. Also, tracking consents processing is not specified but it is highly recommended that implementations provide some mechanism to

ease the pain of implementing tracking management with minimal end-user disturbance.

4.13.1. CDP_ConsentStatus

Uniquely specifies the status of any given Consent

```
enum CDP_ConsentStatus {
    GRANTED,
    DENIED,
    REVOKED
}
```

4.13.2. CDP_Consent

CDP_Consent represents a persisted Consent, always attached to a specific profile.

```
type CDP_Consent {
  token : ID!
  source : CDP_Source
  client : CDP_Client
  type : String!
  status : CDP_ConsentStatus!
  lastUpdate : DateTime
  expiration : DateTime
  profile : CDP_ProfileInterface
  events : CDP_EventConnection
}
```

Token

Similar to OAuth 2 authorization tokens to access the consent without the profile, also useful to delete the consent

Type

Should be a Url or other meaningful identifier like `//mycompany.com/consents/newsletters/weekly`, `//crmcompany.com/consents/push-to-crm` or `//oasis_open.org/cxs/consents/send-to-third-parties`

4.13.3. CDP_ConsentUpdateEvent

Standard EventType to create or update Consents.

```
type CDP_ConsentUpdateEvent implements CDP_EventInterface {
  id: ID!
  cdp_source : CDP_Source
  cdp_client : CDP_Client
  cdp_profileID: CDP_ProfileID!
  cdp_profile : CDP_Profile!
  cdp_object: CDP_Object!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [CDP_Topic]
  type : String!
  status : String,
  lastUpdate : DateTime,
  expiration : DateTime
}
```

4.13.4. CDP_ConsentUpdateEventInput

Input type for ConsentUpdateEvent

```
input CDP_ConsentUpdateEventInput {
  type : String!
  status : String,
  lastUpdate : DateTime,
  expiration : DateTime
}
```

Example of how to update a consent for a profile :

Operation

```
mutation updateConsent($events: [CDP_EventInput]!) {
  cdp {
    processEvents(events: $events)
  }
}
```

Variables

```
{
  "events": [
    {
      "_profileID": {
        "clientID": "crm",
        "id" : "crm-profile-id"
      },
      "_object": "cdp_profile:crm/crm-profile-id",
      "_consentUpdateEvent": {
        "type": "newsletter",
        "status": "GRANTED",
        "lastUpdate": "now",
        "expiration": "now+365d"
      }
    }
  ]
}
```

4.13.5. CDP_ConsentUpdateEventFilter

Filter for ConsentUpdateEvents

```
type CDP_ConsentUpdateEventFilter {
  type_equals : String,
  status_equals : String,
  lastUpdate_equals : DateTime,
  lastUpdate_lt : DateTime,
  lastUpdate_lte : DateTime,
  lastUpdate_gt : DateTime,
  lastUpdate_gte : DateTime,
  expiration_equals : DateTime,
  expiration_lt : DateTime,
  expiration_lte : DateTime,
  expiration_gt : DateTime,
  expiration_gte : DateTime
}
```

4.13.6. CDP_ConsentUpdateEventFilterInput

Input type for of ConsentUpdateEventsFilter

```
input CDP_ConsentUpdateEventFilterInput {
  type_equals : String,
  status_equals : String
  lastUpdate_equals : DateTime,
  lastUpdate_lt : DateTime,
  lastUpdate_lte : DateTime,
  lastUpdate_gt : DateTime,
  lastUpdate_gte : DateTime,
  expiration_equals : DateTime,
  expiration_lt : DateTime,
  expiration_lte : DateTime,
  expiration_gt : DateTime,
  expiration_gte : DateTime
}
```

4.14. Views

Views provide a way of grouping administrative objects in the Customer Data Platform. Profiles, Events and Consents are all collected and stored globally, but other items are typically handled by administrators or marketeers, and benefit from being grouped into different views to simplify handling.

Lists, Segments, Topics and Personas are all tagged with Views.

4.14.1. CDP_View

```
type CDP_View {
  name: ID!
}
```

4.14.2. CDP_ViewInput

```
input CDP_ViewInput {
  name: ID!
}
```

4.14.3. CDP_Query

View queries

```
  getViews : [CDP_View]
```

4.14.4. CDP_Mutation

View mutations

```
  createOrUpdateView(view: CDP_ViewInput) : CDP_View
  deleteView(viewID : ID!) : Boolean
```

4.15. Topics

Topics represent the core entities of the business that is using the Customer Data Platform. The Customer Data Platform aims to find correlation between profiles and the topics. When such correlations are identified, it is called Interests.

CDP Administrators need to maintain a list of topics in order to obtain profile interests. Profile interests is typically a core objective of Marketing activities, and targeting users with better content.

Example Topics for a car manufacturer might for instance be:

- "Model S"
- "Model 3"
- "Model X"

Topics are associated with Objects and Profiles through Events. An example of how this might work in real life: A website promoting a specific Product, for instance "Car type X", should also contain meta-data for the associated topic i.e. "model X". The web tracking script can then feed this information back to the CDP, including both the object (web page in this case), and the specific topic. This way, the CDP will be able to build a model of association.

4.15.1. CDP_Topic

```
type CDP_Topic {
  id : ID!
  view : CDP_View!
  name: String!
}
```

4.15.2. CDP_TopicInput

```
input CDP_TopicInput {
  id : ID
  view : ID!
  name: String!
}
```

4.15.3. CDP_TopicFilterInput

```
input CDP_TopicFilterInput {
  and : [CDP_TopicFilterInput]
  or : [CDP_TopicFilterInput]
  view_equals : ID
  id_equals : String
  name_equals : String
}
```

4.15.4. CDP_Query

Topic queries

```
  getTopic(topicID : ID) : CDP_Topic
  findTopics(filter: CDP_TopicFilterInput, orderBy: [CDP_OrderByInput], first: Int,
after: String, last: Int, before: String) : CDP_TopicConnection
```

4.15.5. CDP_Mutation

Topic mutations

```
  createOrUpdateTopic(topic : CDP_TopicInput) : CDP_Topic
  deleteTopic(topicID : String) : CDP_Topic
```

4.16. Interests

An important use-case for the Customer Data Platform is to determine a profile's "Interests". Whenever the Customer Data Platform registers an events that are associated with one or more Topics, this will affect the profile's interest for the specific Topic. A profile's interest for a specific topic is measured between 0-1, where 1 is maximum. As such 0,5 would indicate a higher interest than 0,35.

The algorithm for how a Customer Data Platform scores and interest is implementation specific - but implementations should also take care of automatically decreasing interest over time, unless new and relevant events occur.

Example interests for products from a car manufacturer might be:

- Model S = 0.1
- Model 3 = 0.3
- Model X = 0.9
- Model Y = 1.0

4.16.1. CDP_Interest

Interests are calculated automatically based on implementation specific algorithm

```
type CDP_Interest {
  topic: CDP_Topic!
  score : Float
}
```

score

will be between 0.0 to 1.0

4.16.2. CDP_InterestInput

Specifying interest is only relevant for Personas

```
input CDP_InterestInput {
  topic : ID!
  score : Float
}
```

4.16.3. CDP_InterestFilter

Used to filter interests, mostly for administration purposes

```
type CDP_InterestFilter {
  and : [CDP_InterestFilter]
  or : [CDP_InterestFilter]
  topic_equals : ID
  score_equals : Float
  score_lt : Float
  score_lte : Float
  score_gt : Float
  score_gte : Float
}
```

4.16.4. CDP_InterestFilterInput

```
input CDP_InterestFilterInput {
  and : [CDP_InterestFilterInput]
  or : [CDP_InterestFilterInput]
  topic_equals : ID
  score_equals : Float
  score_lt : Float
  score_lte : Float
  score_gt : Float
  score_gte : Float
}
```

4.17. Personas

A persona is a concept used to personify your audience. This may for instance be used to test personalization and targeting of content in a 3rd party system.

In CDP, personas are essentially "dummy" profiles with the primary purpose of testing or emulating a real profile. A common use-case would be testing personalized content in a CMS or a newsletter.

Personas and their fields can be explicitly created, where real profiles are built from a stream of events.

Here’s an example of creating a persona :

Operation

```
mutation updatePersona($persona: CDP_PersonaInput) {
  cdp {
    createOrUpdatePersona(persona: $persona) {
      id
    }
  }
}
```

Variables

```
{
  "persona": {
    "_name": "mikeMarketing",
    "_view": "acme",
    "_segments": ["segment1", "segment2"],
    "_consents": [ {
      "type": "newsletter",
      "status": "GRANTED",
      "lastUpdate": "NOW",
      "expiration" : "NOW+30DAYS"
    } ],
    "_interests": [{"topic": "topic1", "score": 10}]
    "firstName" : "Mike",
    "lastName" : "Marketing"
  }
}
```

4.17.1. CDP_Persona

```
type CDP_Persona implements CDP_ProfileInterface {
  id : ID!
  cdp_name : String!
  cdp_view : CDP_View!
  cdp_profileIDs : [CDP_ProfileID]
  cdp_segments(views : [ID]) : [CDP_Segment]
  cdp_interests(views : [ID]) : [CDP_Interest]
  cdp_consents : [CDP_Consent]
  cdp_lists(views : [ID]) : [CDP_List]
  # fields will be added here according to registered profile properties
}
```

4.17.2. CDP_PersonaInput

```
input CDP_PersonaInput {
  id : ID
  cdp_name : String!
  cdp_view : ID!
  cdp_profileIDs : [CDP_ProfileIDInput]
  cdp_segments : [ID]
  cdp_interests : [CDP_InterestInput]
  cdp_consents : [CDP_PersonaConsentInput]
  # fields will be added here according to registered profile properties
}
```

4.17.3. CDP_PersonaConsentInput

Special type to set PersonaConsent without the use of events

```
input CDP_PersonaConsentInput {
  type : String!
  status : String,
  lastUpdate : DateTime,
  expiration : DateTime
}
```

4.17.4. CDP_Query

Persona queries

```
  getPersona(personaID : String) : CDP_Persona
  findPersonas(filter: CDP_ProfileFilterInput, orderBy: [CDP_OrderByInput], first:
Int, after: String, last: Int, before: String) : CDP_ProfileConnection
```

4.17.5. CDP_Mutation

Persona mutations

```
  createOrUpdatePersona(persona : CDP_PersonaInput) : CDP_Persona
  deletePersona(personaID : String) : CDP_Persona
```

4.18. Lists

Lists are explicitly created and named in the Customer Data Platform. Profiles may then be added to a list, and later opt out if desired. Whenever a profile opts out of a list, that information will also be stored. This prevents the profile from accidentally being added back to the list at a later point.

A common use-case for lists is creating a list for a campaign, and add the target profiles to the list as the campaign starts.

4.18.1. CDP_List

```
type CDP_List {
  id : ID!
  view: CDP_View!
  name : String!
  active(first: Int, after: String, last: Int, before: String) : CDP_ProfileConnection
  inactive(first: Int, after: String, last: Int, before: String) :
CDP_ProfileConnection
}
```

id

Cannot change and is usually server generated

4.18.2. CDP_ListInput

```
input CDP_ListInput {
  id : ID
  view: ID!
  name : String!
}
```

4.18.3. CDP_ListsUpdateEvent

Standard Event to update profile membership for specified lists

```
type CDP_ListsUpdateEvent implements CDP_EventInterface {
  id: ID!
  cdp_source : CDP_Source
  cdp_client : CDP_Client
  cdp_profileID: CDP_ProfileID!
  cdp_profile : CDP_Profile!
  cdp_object: CDP_Object!
  cdp_location: GeoPoint
  cdp_timestamp: DateTime
  cdp_topics : [CDP_Topic]
  joinLists : [CDP_List]
  leaveLists : [CDP_List]
}
```

4.18.4. CDP_ListsUpdateEventInput

```
input CDP_ListsUpdateEventInput {
  joinLists : [ID]
  leaveLists : [ID]
}
```

Example of how to update lists for a profile :

Operation

```
mutation updateLists($events: [CDP_EventInput]!) {
  cdp {
    processEvents(events: $events)
  }
}
```

Variables

```
{
  "events": [
    {
      "_profileID": {
        "clientID": "crm",
        "id" : "crm-profile-id"
      },
      "_object": "cdp_profile:crm/crm-profile-id",
      "_listsUpdateEvent": {
        "joinLists": ["list1", "list2"],
        "leaveLists": ["list3", "list4"]
      }
    }
  ]
}
```

4.18.5. CDP_ListsUpdateEventFilter

```
type CDP_ListsUpdateEventFilter {
  joinLists_contains : [ID]
  leaveLists_contains : [ID]
}
```

4.18.6. CDP_ListsUpdateEventFilterInput

Used to filter list update events when querying events

```
input CDP_ListsUpdateEventFilterInput {
  joinLists_contains : [ID]
  leaveLists_contains : [ID]
}
```

4.18.7. CDP_ListFilterInput

Used to filter lists in for management purposes

```
input CDP_ListFilterInput {
  and : [CDP_ListFilterInput]
  or : [CDP_ListFilterInput]
  view_equals : ID
  name_equals : String
}
```

4.18.8. CDP_Query

List queries

```
  getList(listID : ID) : CDP_List
  findLists(filter: CDP_ListFilterInput, orderBy: [CDP_OrderByInput], first: Int,
after: String, last: Int, before: String) : CDP_ListConnection
```

4.18.9. CDP_Mutation

List mutations

```
  createOrUpdateList(list : CDP_ListInput) : CDP_List
  addProfileToList(listID : ID, profileID : CDP_ProfileIDInput, active : Boolean) :
CDP_List
  removeProfileFromList(listID : ID, profileID : CDP_ProfileIDInput) : CDP_List
  deleteList(listID : ID) : CDP_List
```

4.19. Segments

Segments are similar to lists in that profiles may be in the segment, or not. However, where profiles are explicitly added to lists, they are dynamically resolved to segments based on the filter defined in the segment.

Administrative users define segments through Filters.

Example segments:

- **Rich europeans**: Profiles in Europe with income above €100k
- **Frequent buyer**: Profiles that have completed more than 5 transactions in the last 3 months

Here's an example operation to create a "male" segment (it assumes a "gender" profile property has been defined).

operation

```
mutation createSegment($segment: CDP_SegmentInput) {
  cdp {
    createOrUpdateSegment(segment: $segment) {
      name
    }
  }
}
```

variables

```
{
  "segment": {
    "name": "males",
    "view": "acme",
    "profiles": {
      "properties": {
        "gender_equals" : "male"
      }
    }
  }
}
```

4.19.1. CDP_Segment

```
type CDP_Segment {
  id : ID!
  view: CDP_View!
  name : String!
  profiles : CDP_ProfileFilter
}
```

4.19.2. CDP_SegmentInput

```
input CDP_SegmentInput {
  id : ID
  view : ID!
  name : String
  profiles : CDP_ProfileFilterInput
}
```

4.19.3. CDP_SegmentFilterInput

```
input CDP_SegmentFilterInput {
  and : [CDP_SegmentFilterInput]
  or : [CDP_SegmentFilterInput]
  view_equals : ID
  name_equals : String
}
```

4.19.4. CDP_Query

Segment queries

```
  getSegment(segmentID : ID) : CDP_Segment
  findSegments(filter: CDP_SegmentFilterInput, orderBy: [CDP_OrderByInput], first:
Int, after: String, last: Int, before: String) : CDP_SegmentConnection
```

4.19.5. CDP_Mutation

Segment mutations

```
  createOrUpdateSegment(segment : CDP_SegmentInput) : CDP_Segment
  deleteSegment(segmentID : String) : CDP_Segment
```

4.20. Profile matching

Clients may want to identify in real time if a given profile matches a specific segment, or filter. This can effectively used in order to produce personalized responses or messages.

4.20.1. CDP_NamedFilterInput

Named filters are used to evaluate filters against a profile - useful for building personalized experiences

```
input CDP_NamedFilterInput {
  name : String!
  filter: CDP_ProfileFilterInput
}
```

4.20.2. CDP_FilterMatch

The result of a named filter match request

```
type CDP_FilterMatch {
  name : String
  matched : Boolean
  executionTimeMillis : Int
}
```

Below is an example of matching a profile with a filter in real-time:

Operation

```
query profileMatching(
  $profileID: CDP_ProfileIDInput
  $namedFilters: [CDP_NamedFilterInput]
) {
  cdp {
    getProfile(profileID: $profileID) {
      cdp_matches(namedFilters: namedFilters) {
        name
        matched
      }
    }
  }
}
```

Variables

```
{
  "profileID": {
    "clientID": "crm",
    "id": "crm-profile-id"
  },
  "namedFilters": [
    {
      "name": "continentMatch",
      "filter": {
        "properties": {
          "continent_equals" : "Europe"
        }
      }
    }
  ]
}
```

4.21. Data Intelligence

The collection of structured information in a CDP enables potential beyond simply accessing these data. By applying algorithms or machine learning techniques to the data, a CDP can act as a real-time data source for advanced use cases in other applications.

The collection of structured information in a CDP enables potential beyond simply accessing these data.

4.21.1. CDP_ScoredObject

Objects with a specific scoring

```
type CDP_ScoredObject {
    object : CDP_Object
    score : Float
}
```

4.21.2. CDP_AlgorithmInput

Defining a specific algorithm to apply.

```
input CDP_AlgorithmInput {
    name : String!
    parameters : JSON
}
```

Name

Implementation specific algorithms, examples may be `collaborative-filtering`, `clustering`, `deep`, `trending`, etc

Parameters

JSON object supported by the specified algorithm. Algorithms must validate the object themselves. Parameters can be used to filter the results of the recommendation algorithm or any other custom processing that is supported by the implementation.

4.22. Optimizations

This part of the specification is OPTIONAL

A specific application of data intelligence is smart decision making, or optimizations. In short, an optimization is done by passing a number of objects in, and letting the system rank them according to which is considered optimal. For instance, which product is most relevant for a specific visitor.

4.22.1. CDP_OptimizationResult

The result of an optimization, containing scored objects

```
type CDP_OptimizationResult {
    name : String!
    scoredObjects : [CDP_ScoredObject]
}
```

4.22.2. CDP_OptimizationInput

Definition of the optimization to perform

```
input CDP_OptimizationInput {
    name : String!
    objects : [ID],
    eventOccurenceBoosts : [CDP_EventOccurenceBoostInput]
    strategy : String
    size : Int
}
```

Strategy

Any strategy supported by the algorithm: Unspecified, random, scoring, best first match, worst match, a/b test

4.22.3. CDP_EventOccurenceBoostInput

Used to boost positively/negatively the algorithm based on event type and time span: i.e. return a list of products the profile has viewed in the last year

```
input CDP_EventOccurenceBoostInput {
    eventType : String
    boost : Int
    fromDate : DateTime
    toDate : DateTime
}
```

Boost

Can also be a negative value

Example of an optimization of objects for a given profile :

Operation

```
query profileOptimizations(
  $profileID: CDP_ProfileIDInput
  $optimizationParameters: [CDP_OptimizationInput]
) {
  cdp {
    getProfile(profileID: $profileID) {
      cdp_optimize(parameters: $optimizationParameters) {
        name
        scoredObjects {
          object {
            uri
            scheme
            path
            topics {
              name
              view {
                name
              }
            }
          }
          score
        }
      }
    }
  }
}
```

Variables

```
{
  "profileID": {
    "clientID": "crm",
    "id": "crm-profile-id"
  },
  "optimizationParameters": [
    {
      "name": "carPromotion",
      "objects" : [
        "cars:modelS",
        "cars:modelX",
        "cars:model3"
      ],
      "eventOccurenceBoosts": {
        "eventType": "configuredCar",
        "boost": 3.0,
        "fromDate" : "NOW-1MONTH",
        "toDate" : "NOW"
      },
      "strategy": "scoring",
      "size" : 2
    }
  ]
}
```

4.23. Recommendations

This part of the specification is OPTIONAL

Unlike optimizations that act on a defined list of objects, recommendations take an object as input, only to suggest other objects based on a specific algorithm.

4.23.1. CDP_RecommendationResult

Provides a list of scored object

```
type CDP_RecommendationResult {
    name : String!
    scoredObjects : [CDP_ScoredObject]
}
```

4.23.2. CDP_RecommendationInput

```
input CDP_RecommendationInput {
    name : String!
    objectUri : ID
    topics : [ID]
    size : Int
    algorithm : CDP_AlgorithmInput
}
```

objectUri

Specific object that is the originator of the recommendation

topics

Objects have to be related to these specific topics

size

Maximum number of results to retrieve

Example of how to get a recommendation for a profile :

Operation

```
query profileRecommendations(
  $profileID: CDP_ProfileIDInput
  $recommendationParameters: [CDP_RecommendationInput]
) {
  cdp {
    getProfile(profileID: $profileID) {
      cdp_recommend(parameters: $recommendationParameters) {
        name
        scoredObjects {
          object {
            uri
            scheme
            path
            topics {
              name
              view {
                name
              }
            }
          }
          score
        }
      }
    }
  }
}
```

Variables

```
{
  "profileID": {
    "clientID": "crm",
    "id": "crm-profile-id"
  },
  "recommendationParameters": [
    {
      "name": "similarBooks",
      "objectUri": "books:BOOK-ISBN-CODE",
      "topics": ["murderMysteries"],
      "size": 10,
      "algorithm": {"name": "similar"}
    }
  ]
}
```

5. Security Considerations

The goal of CDP is to aggregate and store personal data. Failure in securing the data may have dramatic consequences, both financially and in direct customer relationship for the involved parties.

5.1. Attack surface

Architecturally, CDP is designed to only be accessible through a single API. This limits the attack surface. Deliberately, the standard does not specify how the API is secured, as this can be handled using traditional web security mechanisms, such as IP filtering and certificates.

5.2. Network communication

All communication going through a network, be it a local or global one, should be encrypted using latest recommended standards in the matter. In the case of the GraphQL API, it is highly recommended to use HTTPS connections to avoid man-in-the-middle attacks and eavesdropping. It is also not recommended that the GraphQL API be publicly and directly available, but only available to known and trusted clients.

5.3. Client tokens

Communicating with the API requires a valid Client. Implementers are strongly encouraged to use additional tokens or similar for securing the client access further.

5.4. Access control

By default, clients get access to all data stored in the CDP.

Implementers are encouraged to implement different levels of access control beyond this. For instance using roles or access control mechanisms, limiting clients to writing events, or allowing access to management objects, etc.

5.5. Authentication

The specification does not set any requirements for authentication However, CDP specifies management objects that are intended to be created and handled by power-users and marketeers.

Implementers are encouraged to support a concepts for users in the implementation and API directly.

By combining the concept of client tokens above with users and/or autorization tokens (i.e using Oauth), implementers may offer granular and controlled access to data through the CDP API.

NOTE	It is always recommended to proxy access to CDP through a gateway client. Direct access from end user devices and other clients poses a higher risk of exposing sensitive data.

5.6. Audit logs

For interaction with management object in particular, it is recommended to implement audit logging.

5.7. Input validation

Thanks to the usage of GraphQL, the API is strongly typed, which implies that input validation is performed on any API request, minimizing the attack surface even more. For more information: https://facebook.github.io/graphql/June2018/#sec-Validation

However, the input validation provided by GraphQL does not free implementations from performing measures against cross-site scripting and other script-injection attacks (eg: SQL injection).

6. Conformance

This section describes requirements for an implementation to claim specification conformance.

6.1. Conformance targets

There are two defined levels of conformance:

- CORE (minimum level conformance)
- FULL (complete implementation, including intelligence capabilities)

6.2. CORE conformance

CORE conformance CDP server implementations:

- MUST implement the specifications of the API section
- MAY implement any parts of the API specification marked as OPTIONAL

6.3. FULL conformance

CORE conformance CDP server implementations:

- MUST meet all requirements of the CORE conformance
- MUST additionally implement all OPTIONAL parts of the API specification

7. Appendix A. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

- Jan Blessenohl, Progress Software
- Kaloyan Nikolov, Progress Software
- Chris Laprun, RedHat

8. Appendix B. Revision History

Revision Date Editor Changes Made