Context Server (CXS) PROPOSAL

Version 0.7

Written by Serge Huber (Jahia) and Thomas Sigdestad (Enonic)
Table of Contents

Context Server (CXS) PROPOSAL
Table of Contents
Introduction
  Seeing things in Context
Use Cases
  WCM targeting
  Profile list
  Newsletter
  Deal closing
  Content Recommendations
  A/B Testing
  Social
  Reporting and analytics
The Context Server
Repositories
  Use case R1 - Profile query
Views
  Use case V1 - Existing Segment
  Use case V2 - New Custom Segment
Profiles - the aggregated visitor properties
Context - the aggregated visitor context
Sessions
Events - representing interaction between profiles and items
  Standard Properties
  Standard event types
Rules
Conditions
Actions
  Sample action that increments a property value
Segments - Rules without actions
Algorithms - Data-driven personalization
(Optional) Profile Merges
Context server requirements
  Providing context in real time
Technical solution
REST API
  Accessing API Endpoints
  Condition Types API
    Get all condition types
Get specific condition types
Get all condition types by tag
Sample condition type
Standard condition types

Actions Types API
Get all action types
Get specific action definition
Get all action types by tag
Sample action definition

Property Types API
Get all property types
Get specific property type
Get all property types by tag
Create property type
Update property type
Delete property type
Sample property type

Value Types API
Get all value types
Get specific value type
Get all value types by tag

Profile API
Get all profiles
Get specific profile
Find profiles
Create Profile
Update Profile
Delete Profile
Get all profiles that match a segment

Event API
Collect Event
Example
Get Event
Find Events
Delete Event

Context API
Event Collector Script

Segments API
Get segments
Get segment
Find segments
Create segment
Update Segment
Delete Segment

Rules API
- Get Rules
- Get Rule
- Create Rule
- Update Rule
- Delete Rule

Algorithms API
- Sample Algorithm
- Example
  - Retrieving recommended items for item x

Building queries using condition trees
Query
- Sample condition usage that checks the gender of a user
- Sample of condition query used inside a segment definition

Aggregations
Paging
Sorting

Other ideas
- Items - generic object representing any physical or virtual entity
  - Standard item types

Item API
- Get Items
- Get Item
- Create item
- Update item
- Delete item
Introduction

“To create a perfect experience for you - we must understand why you are here. We achieve this by understanding who you are and your current context…”

Creating, managing and delivering personalized web and digital experiences are considered one of the hottest - and most complex topics right now.

The modern digital user expect businesses to fully understand them and their personal needs. Sadly, in real life - this seems further from the truth than just about anything else. Only a very few businesses are able to deliver on these expectations today.

*The user expects businesses to recognize him across systems and channels*
Businesses face the “Context Management eXperience”

The Context Server - a hub for collecting and distributing user context across systems
Businesses are already collecting and storing massive amounts of information - almost always in silos. If combined and accessible, that information could be actively used to improve the business' digital (and even analog) services.

Systems in areas of Analytics, Social, Marketing Automation, CRM, CMS, E-commerce etc all perform do this in some extent. With the increased use of Cloud tools - the number and variety of systems is growing rapidly. Even if these systems provide API's they typically only provide peer-to-peer integrations. As such they are hard and expensive to integrate.

By enabling these systems to work better together - businesses will be able to build, integrate and deliver consistent and tailored web and digital experiences faster and better across channels.

**Seeing things in Context**

We need a common standard to store and deliver consistent user context across different systems - this will enable all systems to participate in creating _the big context_ - and share it among each other.

CXS defines a standardized and extensible model for collecting and sharing visitor context - in order to deliver personalized and better user experiences across systems, while improving user privacy at the same time.

**Use Cases**

Below are suggested use-cases. At the moment these are used to identify the scope of the standard..

**WCM targeting**

CMS managers needs to target “specific ad” to existing customers they want to sell a specific product to. Implies that the context server knows about historical events of the visitor.

**Profile list**

Marketing wants to obtain a list of “profiles” based on age, behaviour and demographic conditions. Implies that the system is able to “query” existing profile information and export the results.

**Newsletter**

Newsletter application must avoid sending “the wrong” content to it’s recipients. For example avoid sending a reduced price to a customer who just bought the product at the old price.

**Deal closing**

Convince a hesitating user to buy a product by offering him free shipping or the ability to talk to customer services. This use case implies that the system identifies the “hesitation” of the user and may send the offer
directly on the site or via email. Hesitation could also be used to detect users that are looking for information but can’t find it, and offering to contact them (live chat, email).

### Content Recommendations

Present visitor with relevant content based on the behaviour of other visitors. A typical use-case for e-commerce in “users who viewed this also viewed” or “users who bought this also bought”.

### A/B Testing

Optimization team wants to test the efficiency of different experiences with randomized target groups. A typical requirement for any A/B testing strategy - regardless of tool. A plugin to the context server could set a specific “target group” on a visitor.

### Social

Commerce site wants to give special offers to people who “like” their Facebook page. Implies the integration with social services such as Facebook, Twitter, …

### Reporting and analytics

Content author wants to know about the historical and real-time traffic on of his latest article. This could be achieved through querying and retrieving “faceted” data from the context server (and limited in time for real-time queries)

### The Context Server

The context server represents the core solution to the problems and requirements described above. It is responsible for collecting, aggregating and delivering the context information to any client application. This way the context server can act as the hub between the different applications and in turn enable better experiences.

### Repositories

A context server may support multiple repositories. Repositories provide multi-tenancy and grouping of contextual data into different silos. Each repository contains it’s own set of events, sessions, profiles and items.

Also, rules - with their consequences and actions are configured on the repository level to manage how data is processed.

A repository should have the following properties.

- Identifier (UUID or a unique string name)
- Display Name (For instance company name)
• Description (optional text)

NB! For implementations with a single repository, identifier can be discarded during tracking

Use case R1 - Profile query

At any time, an authorized user may perform “queries” against the repository. Examples of such queries are:
• Profiles matching specific conditions (re-using same logic as for segments)
• Events matching specific conditions
• Items matching specific conditions??

TODO

Views

Once data is collected, it can be analysed and processed. Views provide a flexible way to organize management and reporting on the data in the repository. I.e. a view can be used for different locations, websites, mobile applications, or business divisions in the organization.

Within a view - segments, and algorithm configurations are stored. Clients operate against the entire repository, or a specified view.

A view must have the following properties.

• Identifier (UUID or a unique string name)
• Display Name (For instance company name)
• Description (optional text)

Use case V1 - Existing Segment

Large Enterprise logs data from all systems into Context server. Content Manager on Norwegian web site wants to target an ad to a defined segment (men above 50, defined in the “International” view). The user will end up with a id representing the segment, and tag that ad with the segment ID. At a later point when an ad is supposed to be displayed, the AD system can ask the context server for the current profile, search his ads and serve the best match - based on the tagging.

Use case V2 - New Custom Segment

Large Enterprise logs data from all systems into Context server. Content Manager on Norwegian web site needs to create a custom segment - “norwegian real estate owner”. The user will now create the new segment in the view he has permissions to write into i.e. “Norway”, preferably using his local interface and the CXS Rest API. During creation and adding conditions, the user will see the number of profiles matching in real time (both segment and condition???). User then continues as above.
Profiles - the aggregated visitor properties

Profiles are built from the information recorded by the context server or directly populated by connected systems.

A Profile represents an individual - and can be everything from an “anonymous user” (first time visitor) all the way to a rich object representing a known individual with a long history of events. As more and more information is collected, the profile will gradually evolve or even be merged with other profiles (for example if we discover that an anonymous user is actually related to an existing user profile). The profiles properties can be aggregated and built dynamically - or even set manually.

Based on defined conditions, a context server should also be able to return matching profiles. Profile lists can effectively be used by 3rd party applications for any given purpose. For instance, a newsletter could be sent to a specific set of people.

Profiles may also be made accessible for updating profile properties or even for managing privacy (by preventing data recording or deleting existing data).

Context - the aggregated visitor context

The context object is created by or delivered to clients on demand. The context object contains all relevant data such as visitor profile, visitor segments, visitor location etc. Also the context server endpoint(s) for any API requests.

Sessions

When actively using the context server from any given client to track behaviour - and providing contextual responses - an identifier is used to track the actions of the visitor for the duration of a given session. The client can optionally provide this identifier - if no identifier is provided the server will provide one - which must be held by the client for the duration of the session. This identifier must be unique and non-predictable.

A session holds the following properties
  - Id (automatically generated id)
  - Source Id
  - Profile reference (or reverse?)
  - Session Start time
  - Session Duration (time of last event minus start time)

Session identifiers may then be used to collect events and return matching responses.

Events - representing interaction between profiles and items

Events are of a specific kind and describe an interaction between a profile and an item. Event kinds are typically verbs such as: “view”, “hover”, “purchase”, “rent”, “like” etc.
**Standard Properties**

Events have four mandatory properties:
- **Id** (automatically created by system)
- **Session** (the session the event occurred in, which includes a reference to the)
- **Target** - i.e. “SKU123234” or “https://my.store.com/” (also referred to as item)
- **Source** (event source name, i.e. mysite.com)
- **Time of occurrence**

Additionally optional properties
- **Location** (geo point)
- **Remote Id** (to uniquely identify this event when logged from external system)

Finally, events may also have a set of custom properties as defined by its type.

Events are immutable

**Standard event types**

To simplify use and compatibility - some standard event types should always be present in any CXS compliant context server. Below is a list of suggested Event Types:
- **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**
- **ProfileMerge** (“when two profiles are merged”)
- **Login**
- **Logout**
- **RequestFriendship**
- **AcceptFriendship**
- **DenyFriendship**
- **Click**
- **View**
- **Contribute** (comment, blog etc?)
- **Conversion** (purchase, download, signs up for a service)

**Rules**

Rules are fundamental to make use of the data collected, and to effectively manage a context server. Rules are triggered by logging of events, background jobs or active triggering - and potentially execute actions based on a defined set of conditions.
A rule may be described as such:

```
rule "<name>"
  when
    <condition>*
  then
    <action>*
end
```

There are essentially a few different types of rules:
- Event Rules - applied when an event occurs - the event is input to the rule.
- Profile Rules - applied when “executed” manually, or by background job - All profiles (or optionally a set of them?) are input to rule
- Item Rules (TBD)

**Use case Event Rule: “Score incrementation”**
When a page view event occurs, a user profile score property is set to indicate that the current visitor has reached another marketing goal.

**Use case Event Rule: “Property update”**
When an event is logged (i.e. page view), we want to tag the profile with “language” i.e. “Swiss German (ch_DE)” - which may be derived from the event creation request (i.e. headers)

**Use case Profile Rule: “profile cleanup”**
Remove all profiles that have been inactive for more than 5 years

Here is a JSON format example of a rule that will copy all event properties when a user logs in with Facebook, and then trigger a profile merge using the “email” profile property as a merge key.

```json
{
  "metadata": {
    "id": "facebookLogin",
    "name": "Facebook login",
    "description": "Upon Facebook login event, copy all event properties to profile properties and merge existing profiles based on email"
  },
  "condition": {
    "type": "eventCondition",
    "parameterValues": {
      "eventTypeId": "facebookLogin"
    }
  },
  "actions": [
```
Conditions

Conditions are functions with a defined set of parameters that may be provided as standard by the context server or be provided by plugins. Conditions may be combined by AND, OR or other logic operators into expressions. Here are some sample conditions:

- Visitor age > 40
- Visitor has purchased > 1 item from author “x”
- Has visited “crime” pages the last 24 hours (historical)
- Etc..

NB! Some conditions should be a standard part of the context server core, suggestions are:

- Logical operations (AND, OR etc)
- Profile Conditions (x < y , “x” = “x”, regular expressions, empty etc)
- Event Conditions (matching specific event properties)
- Session Conditions (creation date matching, duration, property testing)
- Match All (returns true always)

Actions

Actions are simply statements that will be executed sequentially whenever a rule matches for a particular event. Actions should be pluggable. Example actions are:

- Set profile property (property name, property value)
- Copy an event property to a profile property (for example copying form values to profile properties)
- Log a new event
- Trigger a job
- Delete something
- Update a remote system
- Notify XXX
Sample action that increments a property value

"actions" : [
   {
      "type": "incrementProfilePropertyValueAction",
      "parameterValues": {
         "propertyName": "pageViews",
      }
   }
]

Segments - Rules without actions

To target visitors with content from 3rd party systems - like CMS’s, Web-applications, E-commerce etc. we need a simple way to interact with the context server.

Segments provide a sound solution to this problem - the primary idea is to easily determine if a visitor/user is actually part of a segment. A CMS could easily tag it’s content with segments, use the context server to gain knowledge of what segments the current visitor is in, and then display relevant content for that visitor. Other systems could reuse the same segments for similar purposes.

Here is an example of a possible segment definition:

Name: “Young rich US segment”
condition: Visitor age is < 30 (demographic)
condition: Income is more than USD 100’000/year (demographic)
condition: Is browsing from a location in the “US” (real-time)

Segments could use other segments in conditions, to build new segments based on existing ones, promoting segment re-use.

Here is an example of a segment expressed in JSON format :

{
   "metadata": {
      "id": "youngRich",
      "name": "Young rich",
      "description": "This segment includes all young rich"
   },
   "condition": {


Algorithms - Data-driven personalization

Algorithms provide a highly automated way of personalizing the user experience - it is able to deliver the user with relevant content based on the behaviour of other users and collected information. The more content and the more visitors a site or application has - the more relevant algorithms may be to deliver the best experience. For example e-commerce site visitors might receive product recommendations based on the purchase history of other users. Another example is listing videos that are similar to the one that was just watched by a visitor.

(Optional) Profile Merges

During event collecting at specific moments the context server will be able to discover if the current visitor (typically anonymous) has multiple profiles (for example because of browsing with different devices) - this might for instance be discovered when the visitor logs in.

At this time the context server could perform a profile merge operation to ensure that all the data collected in both profiles are available in the context.

When two profiles are merged, the event “ProfileMerge” will be logged between the two profiles. It is up to the context server implementation for how and if an actual merge is handled. The only requirement is that
the context object is populated with a unified profile (for example a context server could choose to perform “merges” in near real time when delivering the context object to the client).

**Context server requirements**

- MUST be able to provide clients with a context object for the “current profile” (or create one if there isn’t one yet)
- MUST be able to log events produced by the “current profile”
- MUST be able to store profiles created from anonymous visitors
- MUST be able to track visitors across sessions and match them with profile
- MUST be able to determine matching segments for a profile
- SHOULD scale to accommodate any expected traffic on the accessed systems
- SHOULD provide a way to query the data in the object model
- SHOULD be able to import data (i.e. profile, event, segments, items, …)

**Providing context in real time**

An important factor that needs to be considered is the fact that a context server not only must be capable of storing and processing large amounts of information - it must also be able to provide near real time responses in parallel.

**Technical solution**

The objects that are stored in the context server are defined in the object model (see below) and the API defines how these objects are accessed. URL API endpoints are defined on the server so that the clients may connect to them.

**REST API**

The API defines how to interact with the content in the context server using a simple REST approach. The API is composed of different service layers accessible through different endpoints:

**Accessing API Endpoints**

We can separate the API endpoints into two categories:

- Public APIs that are designed to be accessible by anyone, to integrate directly with clients such as browsers, and that are usually designed to be minimal and fast
- Private APIs that contain management, reporting and other functions that will usually be protected by authentication.
Condition Types API

The purpose of this API is to access all condition types installed on the context server. Condition types are used to determine the capabilities of the server and for dynamically building queries in external user interfaces.

Get all condition types
GET: <baseurl>/types/conditions
Response: {}

Get specific condition types
GET: <baseurl>/types/conditions/<conditionID>
Response: {}

Get all condition types by tag
GET: <baseurl>/types/conditions/_tags/<tag>
Response: {}

Sample condition type
The condition definition can be used by user interfaces to build queries

{
    "id": "userPropertyCondition",
    "tags": [ "demographic", "userCondition" ],
    "parameters": [
        {
            "id": "propertyName",
            "type": "string",
            "multivalued": false,
        },
        {
            "id": "operator",
            "type": "comparisonOperator",
            "multivalued": false,
        },
        {
            "id": "propertyValue",
            "type": "string",
            "multivalued": false,
        }
    ]
}
Standard condition types
- AND
- OR
- User property matching
- Event (login, page view) matching
- Geolocation matching

Actions Types API
Action types are used when building rules. Action types are used to determine the capabilities of the server and for dynamically building rules with actions in external user interfaces.

Get all action types
GET: <baseurl>/types/actions
Response: {}

Get specific action definition
GET: <baseurl>/types/actions/<actionID>
Response: {}

Get all action types by tag
GET: <baseurl>/types/actions/_tags/<tag>
Response: {}

Sample action definition

```json
{
  "id": "incrementProfilePropertyValueAction",
  "tags": ["demographic", "userActions"],
  "parameters": [
    {
      "id": "propertyName",
      "type": "string",
      "multivalued": false,
    }
  ]
}
```
Property Types API

Property types are used to define an available set of properties that can be used within profiles and sessions - and optionally in other objects. The property types enables you to dynamically define profiles through external user interfaces.

Get all property types
GET <baseurl>/types/properties
Response: {}

Get specific property type
GET <baseurl>/types/properties/<propertyID>
Response: {}

Get all property types by tag
GET <baseurl>/types/properties/_tags/<tag>
Response: {}

Create property type
POST: <baseurl>/types/properties/<propertyID>
Response: {}

Update property type
PUT <baseurl>/types/properties/<propertyID>
Response: {}

Delete property type
DELETE <baseurl>/types/properties/<propertyID>
Response: {}

Sample property type

Example of a JSON property type:
{
   "id": "age",
   "valueType": "integer",
   "tags": ["demographicUserProperties"],
   "defaultValue": "",
}

NB! We can possibly map properties to schema.org
Value Types API

Value types are provided from the context server list of available value-types. These may include simple types ranging from “string” and “email” to complex types such as “property set”.

Get all value types

GET <baseurl>/types/values
Response: {}

Get specific value type

GET <baseurl>/types/values/<valueID>
Response: {}

Get all value types by tag

GET <baseurl>/types/values/tags/<tag>
Response: {}

Sample value type

Example of a JSON value type:

```
{
    "id": "email"
}
```

Standard value types are:

- String
- Floating point
- Integer
- Email
- Date and time
- Geopoint

Optional value types may include:

- PropertySet
- Binary

Profile API

The purpose of this API is to manage using CRUD operations the profiles present in a context server.

Get all profiles

GET <baseurl>/profiles
Response: {}

Get specific profile
GET <baseurl>/profiles/<profileID>
Response: {}

Find profiles
GET?/POST <baseurl>/profiles/_find?<query>&<aggregations>&<start>&<count>&<sort>
NB! Result always include “total”
Response: {}

Create Profile
POST <baseurl>/profiles/<profileID>
Response: {}

Update Profile
PUT <baseurl>/profiles/<profileID>
Response: {}

Delete Profile
DELETE <baseurl>/profiles/<profileID>
Response: {}

Get all profiles that match a segment
GET <baseurl>/profiles/_match/segment?start=&count
Response: {}

Event API
This API should be protected by authentication and is not suitable as a public API.

The purpose of this API is to manage using CRUD operations the different events present in a context server.

The collection of events is NOT part of this API, as we are dedicating a separate REST API for live event collections, as the requirements of that API are quite different in both terms of authentication and performance.

Also, usually events are collected into the context server and then managed by the server using its own event lifecycle, so regular management operations such as creating, updating or deleting events might not always be available in the API of a context server.
Collect Event
In order to collect events from live web traffic, a dedicated optimized interface is provided that should aware of the context in which transitions occur between events and items types. For example this API could already be tracking the visitor, making it redundant to specify a visitor ID somewhere in the request.

This API has a specific (and public) endpoint that is “write-only”, and will not allow other operations such as reading or querying.

**GET/POST:** `<baseUrl>/event/collect/<eventtype>?parameters`  
(parameters are specific to event type, and configured in the context server)  
**Response:** none (this is to optimize the performance of the event collection)

**Example**
A visitor has viewed item - identified by “x” and this shall now be logged  
GET: `<host>/event/collect/view?itemid=x>  
Response: none or updated profile???

Get Event
**GET/POST:** `<baseUrl>/events/<event-key>`  
**Response:** `{ _id : "<event-key>", _kind : "rating", _fromItem: "a", _toltem : "b", _timestamp : "2007-04-05T12:30-02:00", _location: "<geopoint>", score: "0.95"}`  
The “_” character indicates that the id is auto generated by the server and that it is usually not mutable.

Find Events
**GET/POST:** `<baseUrl>/events/_find/`  

Delete Event
**DELETE:** `<host>/events/<event-key>`  
**Response:** `{ _id : "<event-key>", _kind : "rating", _fromItem: "a", _toltem : "b", _timestamp : "2007-04-05T12:30-02:00", _location: "<geopoint>", score: "0.95", a: "b"}`

Context API
In order to obtain context information from the context server, the Context API can be used.

**GET:** `<baseUrl>/context.js`  
**Response:**  
window.digitalData = window.digitalData || {};
var cxsDigitalData = {
    "loaded": true,
Event Collector Script
The context server is capable of serving a specific script (Javascript) that will be used for event collection. The script must be loaded on the specific web site that wants to log events. The event collector script uses the event collector API.

We will have to be careful to specify how this will work with CORS security:

View API
Todo

Segments API
The API defines access points for segments, conditions and profiles.
**NB! Must remember that segments are part of views.**

Get segments
GET/POST /segments/
Response: {}

Get segment
GET/POST /segments/<segmentId>/
Response: {}/
Find segments
GET/POST /segments/_find
Response: {}

Create segment
POST /segments/<segmentID>
Response: {}

Update Segment
PUT /segments/<segmentID>
Response: {}

Delete Segment
DELETE /segments/<segmentID>
Response: {}

Rules API
The API defines access points for rules

Get Rules
GET /rules
Response: {}

Get Rule
GET /rules/<ruleID>
Response: {}

Create Rule
POST /rules/<ruleID>
Response: {}

Update Rule
PUT /rules/<ruleID>
Response: {}

Delete Rule
DELETE /rules/<ruleID>
Response: {}

Algorithms API

This API will allow to use the context server to execute algorithms that will produce results based on the current context and previously collected data.

Sample Algorithm

**GET/POST:** `<baseUrl>/algorithms/<algorithm>?parameters`

**Response:** JSON output specific to the algorithm used.

The parameters format is specific to the algorithm used, only their transport as URL parameter is specified here.

Example

*Retrieving recommended items for item x*

GET `<baseUrl>/algorithms/recommendations?itemId=x&eventFilter=view&Itemfilter=<some-query limiting item types returned>`

Response:

```
[ { _id: "item1", _type: "profile", a: "b"},
  { _id: "item2", _type: "product", a: "b"}]
```

Building queries using condition trees

The different parts of the API exposes find methods that are basically used to retrieve a result.

The find methods are built from the following parts: Query, Aggregations, Paging and Sorting. These parts are described below.

**Query**

The query language is based on the conditions system. This allows the query language to be "pluggable" since new conditions could be deployed at any time to perform any advanced queries.

Important factors of the conditions is that it is designed for dynamic UI creation - so external user interfaces can then actually build queries.

**Sample condition usage that checks the gender of a user**

A query will always have a single root condition. This query has only a single condition.

```
{
  "type": "userPropertyCondition",
```
Sample of condition query used inside a segment definition

```json
{
    "metadata": {
        "id": "youngRich",
        "name": "Young rich",
        "description": "This segment includes all young rich"
    },
    "condition": {
        "parameterValues": {
            "subConditions": [
                {
                    "parameterValues": {
                        "propertyName": "properties.age",
                        "propertyValue": 25,
                        "comparisonOperator": "lessThan"
                    },
                    "type": "userPropertyCondition"
                },
                {
                    "parameterValues": {
                        "propertyName": "properties.income",
                        "propertyValue": 10000000,
                        "comparisonOperator": "greaterThan"
                    },
                    "type": "userPropertyCondition"
                }
            ]
        },
        "type": "andCondition"
    }
}
```
Other ideas

This chapter contains other thoughts that have been extracted from the document, and are included as general ideas.

Items - generic object representing any physical or virtual entity

NOTE: Items have been taken out as we currently think context server will work without them. Items can be useful if we were going further into machine learning and algorithms approach.

Items are of a specific type and represent any physical or virtual entities. An item must have a unique identifier, optionally specified by the client.

An item may optionally have a set of custom properties as defined by its type. (Open question: should we support document style properties (with sub-nodes) or flat style?)

Examples of item types are: pages, products, locations etc. Actual items could then be: “my/path/to/home” (page), product “sku#123”, location “Geneva”.

Standard item types

Item types that should be always present in any CXS compliant context server.

- Physical Product
- Virtual Product
- Page (URL)
- Digital Asset (URL)

Item API

The purpose of this API is to manage using CRUD operations the different items present in a context server.
Get Items
GET: <baseurl>/items

Get Item
GET: <baseurl>/items/<myitemid>
Response: { _id : “myitemid”, _type: “user”, a: “b” }

Create item
POST: <baseUrl>/items/?_id=myitemid&a=b

Update item
PUT: <baseUrl>/items/myitemid?a=b&c=d

Delete item
DELETE: <baseUrl>/items/myitemid