

3D Juump Infinite Integration Manual

[IM_EN] version 3.1

Introduction	7
Changes ●	9
Disclaimer.....	10
1 - Documentation coverage	10
2 - Your system security is your responsibility	11
3 - Documentation liability	11
Overview.....	12
1 - Introduction.....	12
2 - Definitions	13
2.1 - Digital mock-up (DMU).....	13
2.2 - 3D Juump Infinite infrastructure	14
3 - Components.....	16
4 - DMU Flow	18
Integration	21
1 - Introduction.....	21
2 - Project.....	22
2.1 - Create a Project	22
2.2 - Access a Project	22
2.3 - Build a Project.....	22
3 - Connector API.....	22
4 - Project Interface.....	24
4.1 - HTTP API.....	24
4.1.1 - Locate the project interface	24
4.1.2 - Declare a new document	25
4.1.3 - Update an existing document.....	25
4.1.4 - Delete an existing document	26
4.1.5 - Retrieve a document.....	26
4.1.6 - Mandatory fields	26

4.2 - Data documents	27
4.2.1 - Types of data documents	27
4.2.2 - Structure document	27
4.2.3 - Geometry document	30
4.2.4 - Metadata document	31
4.2.5 - Annotation document	32
4.2.6 - Attached document	36
4.2.7 - Raster document	37
4.2.8 - Effectivity document	38
4.2.9 - Configuration document	39
4.3 - Project-related documents	40
4.3.1 - Metadata index mapping document	40
5 - Geometry Access Interface	42
5.1 - Overview	42
5.2 - Build setup ●	42
5.3 - Sourcers setup	45
5.3.1 - File system sourcer	45
5.3.2 - HTTP server sourcer	47
5.3.3 - Supported CAD file formats	48
5.4 - Example	49
6 - Control Interface ●	53
7 - Product structure converter	53
7.1 - Introduction	53
7.2 - Using the PSConverter	55
7.2.1 - Command-line	55
7.2.2 - Input	55
7.2.3 - Output	56
8 - Migration	57
8.1 - Usage	57

8.2 - Migration script	57
8.2.1 - Version	57
8.2.2 - Patch document	58
Customization	59
1 - Introduction	59
2 - Metadata usage customization	59
2.1 - Overview	59
2.2 - Customization document	60
2.3 - Customization script	60
2.3.1 - Version	60
2.3.2 - User information	61
2.3.3 - Configuration information ●	61
2.3.4 - Attribute information	61
2.3.5 - Attribute	62
2.3.6 - Search	62
2.3.7 - Identification card ●	63
2.3.8 - Export	67
2.3.9 - Formatting coordinate	67
2.4 - Task-specific scripts	68
2.4.1 - Annotation task customization	68
2.5 - Utility functions	75
3 - Client default settings	75
3.1 - Backface culling	76
3.2 - Frame orientation	76
3.3 - Field of view	76
3.4 - Dynamic low-def ●	76
3.5 - Profiles	77
3.5.1 - Geometric export profile	77
3.5.2 - Image export profile	78

3.5.3 - Metadata export profile	79
3.5.4 - Datapackage export profile.....	80
3.5.5 - Presentation export profile	81
3.5.6 - Alternative profile declaration	82
3.6 - Task-specific settings	82
3.6.1 - Presentation task settings.....	82
4 - Interoperability.....	82
4.1 - Search URL.....	82
4.2 - Select URL	82
Annexes	84
1 - Range of use.....	84
1.1 - Minimum requirements.....	84
1.2 - Supported input formats	84
1.2.1 - Geometry.....	85
1.2.2 - Metadata.....	85
1.2.3 - Annotation.....	85
1.3 - Supported output formats	86
1.3.1 - Geometry.....	86
1.3.2 - Image	86
1.3.3 - Metadata.....	86
1.3.4 - Presentation	86
1.4 - Limits	87
2 - Third-party software licenses.....	87
3 - Export control classification.....	87

Introduction

This documentation is targeted to your CAD database specialists.

This documentation describes the data set generation procedure to let you build your own data from your information system back-ends.

Changes ●

For returning readers, 3D Juump Infinite 3.1 introduces the following important changes:

- The *control interface* is now part of the *Directory API* and is treated in a dedicated document.
- The *geometry interface* has evolved to introduce a new low-def algorithm and speed up the mirroring process. Several new settings are available.
- Access rights to *builds* are now managed through *tags*, introduced at the *geometry interface* level.
- New customization possibilities are introduced, in particular regarding multi-hierarchy *ID Cards*.

Disclaimer

1 - Documentation coverage

The information outlined in this documentation is intended to be used for the following purposes:

- Creation of a specialized 3D Juump Infinite data sets specific to customer needs;
- Diffusion of 3D Juump Infinite data sets by 3D Juump Infinite back-end;

The developed works define any piece of software you are developing with the 3D Juump Infinite software and data sets which have been generated by 3D Juump Infinite software. REAL FUSIO shall not be liable for any damage or injury arising out of any person or entity about these developed works; and the access or inability to access to the data sets built and served by the 3D Juump Infinite server infrastructure.

The developed works with 3D Juump Infinite software shall conform to design and implementation guidelines and restrictions described in the documentation. The software functions available for development are documented. Undocumented functionality should not be utilized without REAL FUSIO express consent.

2 - Your system security is your responsibility

You are responsible for the security of your system.

The developed works on the 3D Juump Infinite server should neither compromise data integrity; nor security of data and applications. The software accesses should be enforced by the 3D Juump Infinite server software proper configuration, third party software administration and the developed works. Your system administrator should read all documentations provided with this product to fully understand the available features.

3 - Documentation liability

The source code and template examples provided thorough this manual are only intended for educational purposes and are not intended to be a substitute to your developed works to be customized to comply with your security, performance or robustness requirements.

Although reasonable effort is made to ensure that the information in our documentation is complete and accurate at the time of release, REAL FUSIO cannot assume responsibility for any existing errors. Changes and/or corrections to the information contained in such documentation may be incorporated in future versions.

Overview

1 - Introduction

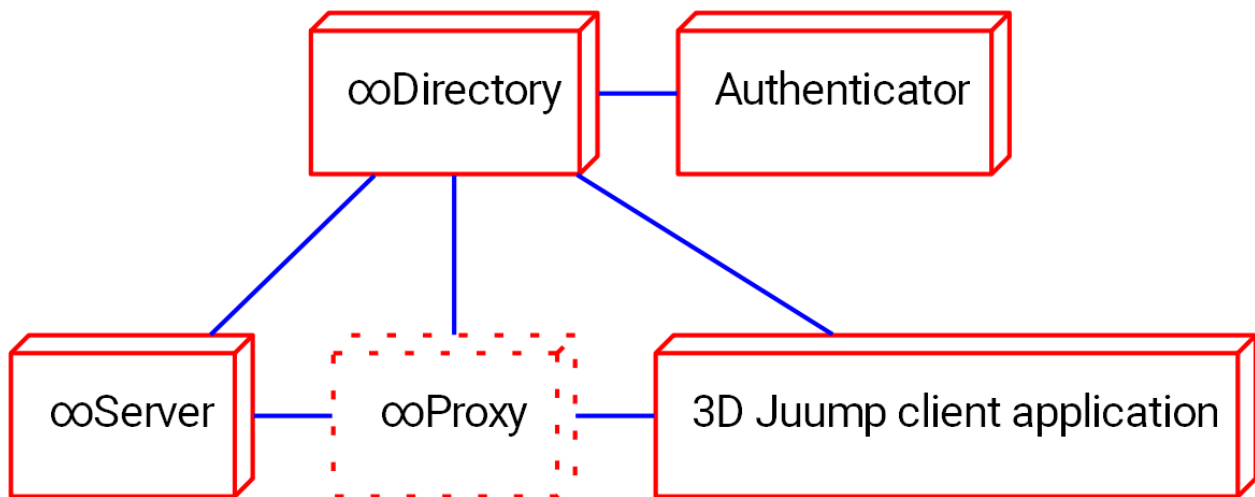
3D Juump Infinite is a software suite allowing the users to browse an entire DMU¹ in 3D on a standard PC².

It relies on a publication server (or **∞Server**), responsible for the preparation and optimization of the CAD source data, a network of relays (or **∞Proxy**) for broadcasting, and a central management server (or **∞Directory**).

On the end user side, the 3D Juump Infinite DMU client application browser is simply called **3D Juump Infinite**.

¹ Digital Mock-Up

² Personal Computer



3D Juump servers and client

2 - Definitions

In order to explain the design of 3D Juump Infinite, it is mandatory to introduce several keywords related to the DMU and CAD product.

2.1 - Digital mock-up (DMU)

The digital mock-up (DMU) is composed of multiple elements:

Digital mock-up

A mock-up is a partial or complete representation of a system in order to preview, test or validate its aspects or its behavior. A digital mock-up is built from computer files storing a tree of three dimension geometries, displayed with a 3D rendering software, in our case, 3D Juump.

Part

A model or template, be it an assembly or a single element. This part is referenced by the digital mock-up but is not present (*instanced*) within it. It is not localized in the digital mock-up. Ex: a wheel.

Part Instance

One instance of a *part*, be it an assembly or a single element. This instance has a 3D representation located in the digital mock-up. Ex: the front-right wheel.

Product Structure

A hierarchy of *part instances*. Ex: a car with four wheels.

Part Number

The unique identifier of a *part*.

Metadata

Any textual or numeric information decorating a *part* or *part instance*, usually presented as a *key=value* pair. Ex: *provider='WheelEx Inc.'*.

Annotation

Any textual or graphical information decorating a *part* or *part instance*, represented in 3D as floating labels. Ex: contextual directions for assembly, tolerancing, etc.

Effectivity

A global parameter or option that changes the way the DMU should be assembled. Ex: What is the car drive-train? Is this car a 4-wheel or a 2-wheel drive vehicle?

Configuration

A set of effectivities. A digital mock-up in a given configuration is also called a configured digital mock-up. Ex: a 4-wheel, metallic paint, 150 hp engine car.

2.2 - 3D Juump Infinite infrastructure

Other keywords are specific to the 3D Juump Infinite infrastructure:

Project

An incremental set of data describing a DMU. It usually corresponds to a product (or product-line) level assembly. For instance, a vehicle manufacturer would probably opt for one project per car model.

Build

An optimized packaged DMU snapshot ready for publication. The DMU processed by 3D Juump Infinite back-end components and served to 3D Juump Infinite front-end client applications.

Connector

A piece of software that feeds a 3D Juump Infinite *project* with data describing the DMU.

DMU data source provider

Any component of your *information system* able to publish DMU data encompassing the part geometry, metadata or effectivity. These components could be a conjunction of your

CMS³, CRM⁴, your ERP⁵, your PDM⁶, your PLM⁷ or simply a file system folder containing all your geometry part files extracted from your CAD⁸ software.

Document

A document is a JSON⁹ exchange file used by the *Connector* and the 3D Juump Infinite back-end to transmit and store data bound to DMU *build* generation. JSON is an open standard human-readable text used to transmit data objects consisting of attribute-value pairs. Thus, this format is also used by 3D Juump Infinite third-party software and the 3D Juump Infinite front-end to exchange data.

DMU flow

A chain of your enterprise components and 3D Juump Infinite back-end components, in charge of DMU extraction from *DMU data source providers*, processing & broadcasting of DMU *builds*.

User

A person that uses 3D Juump Infinite, either through its web administration interface, through a 3D Juump Infinite client application or through one of the offered API. All users must be properly authenticated before being authorized to use 3D Juump Infinite.

Authenticator

A third-party server in charge with the user authentication. 3D Juump Infinite relies on OpenID Connect identity layer to delegate authentication.

Administrator

A user which can logon on the ∞Directory web administration interface. He is able to configure the *DMU flows*, to trigger the generation of DMU *builds* and create *users* and *teams*.

Tag

A keyword used to decorate a *build*, a *user* or a server/proxy component and which defines the access rights to the DMU.

³ Content Management System

⁴ Customer Relationship Management

⁵ Enterprise Resource Planning

⁶ Product Data Management

⁷ Product Lifecycle Management

⁸ Computer Aided Design

⁹ JavaScript Object Notation

Team

A set of users. It mainly acts as a helper concept that applies a common list of tags to its users.

Asset

Any 3D Juump Infinite client application setting (bookmarks, visibility layers, export configurations...) which can be created, manipulated and shared amongst 3D Juump Infinite *users* thanks to 3D Juump Infinite back-end components.

3 - Components

3D Juump Infinite is composed of several software entities.

∞Server

An ∞Server is in charge of the optimization and the packaging of the DMUs (or *projects*) into distributable *builds*. To let *connectors* declare DMUs, the ∞Server publishes a *connector API*.

∞Proxy

An ∞Proxy acts as a proximity relay for DMU broadcasting. It replicates *builds* from ∞Servers or other ∞Proxies. Since an ∞Server is itself an ∞Proxy, the deployment of an ∞Proxy is *not necessary* if your information system and network do not require such DMU broadcasting.

∞Directory

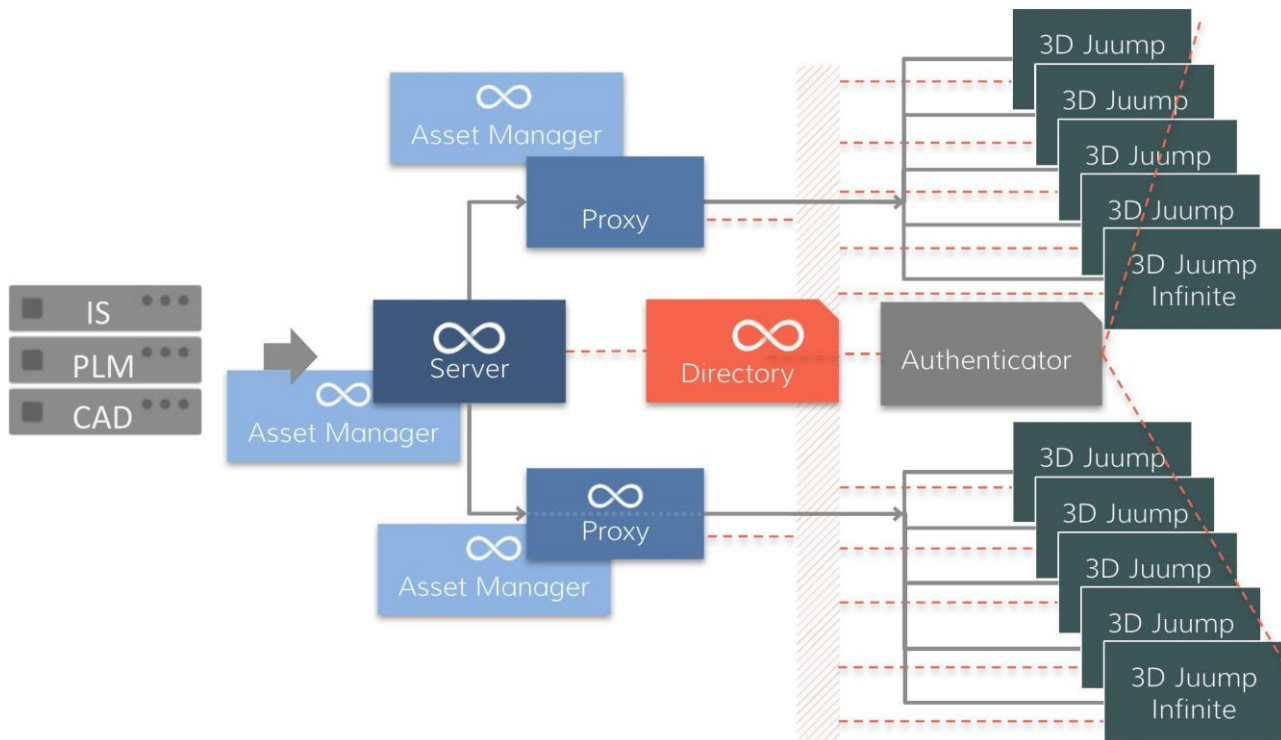
The ∞Directory is in charge of the security management. It monitors and defines the whole 3D Juump Infinite network and is responsible for access-rights management. This management is accessible via a *web administration interface* and programmable via the *directory API*.

3D Juump Infinite client application

The 3D Juump Infinite client application is the DMU browser itself, called *3D Juump Infinite*, when this name is unambiguous.

Web client or Native client

The 3D Juump Infinite client application comes in two flavors. One is the legacy native application (called "Native client") and the other is the new web-based application (called "Web client").



Overview

The previously presented software relies on several third-party components including databases and servers. In particular:

- a [PostgreSQL](#) service,
- a [CouchDB](#) service,
- an [ElasticSearch](#) service,
- an [Apache](#) HTTP service,
- a [LM-X](#) service.

PostgreSQL (or “Postgres”) is an SQL object-relational database management system (ORDBMS). It is used by 3D Juump Infinite ∞Directory, ∞Server and ∞Proxy as underlying database engine.

CouchDB is a document-oriented NoSQL¹⁰ database that uses JSON to store data, JavaScript as its query language using MapReduce, and HTTP for an API¹¹. It is used to store 3D Juump Infinite assets.

Elasticsearch is a distributed, multitenant-capable full-text search server with a RESTful web interface and schema-free JSON documents. Elasticsearch is built on top of on Apache Lucene, developed in Java and is released as open source under the terms of the Apache License. It provides full-text search capabilities to the 3D Juump Infinite client application.

¹⁰ No SQL, i.e. schema-less and non transactional

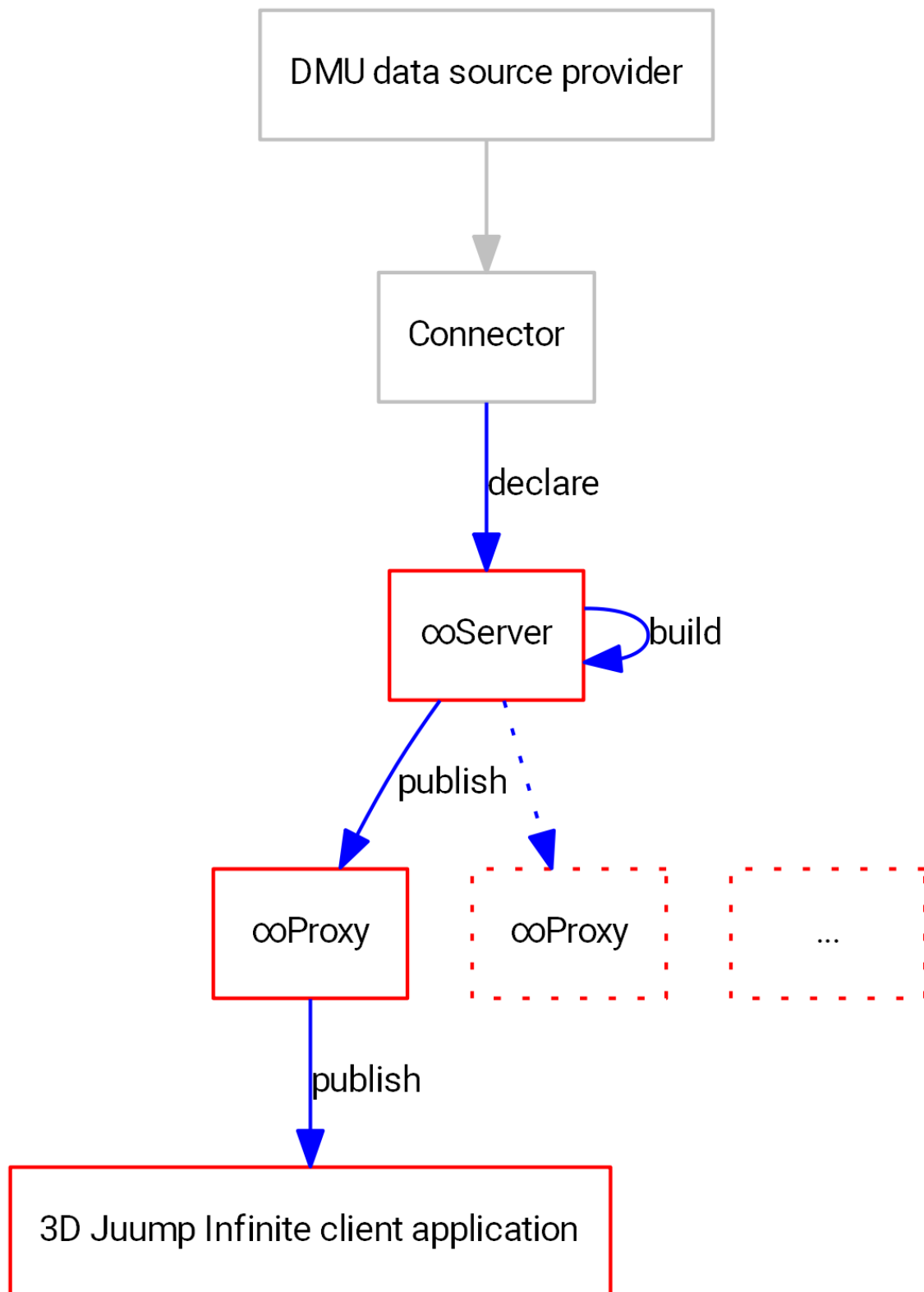
¹¹ Quoted from [CouchDB front page](#)

LM-X is a software licensing solution.

4 - DMU Flow

From the enterprise CAD data source to the 3D Juump Infinite client application, the DMU follows several steps:

- First, your enterprise defined **Connector**, bound to your DMU data source providers, incrementally declares the product structure and the associated metadata, effectivities and configurations to a *project* hosted on an ∞Server.
- Once the declared product structure is coherent, the connector triggers a *build*.
- The ∞**Server** compiles the product structure and the associated data into a hierarchical database where every leaf points to a geometry.
- Then the ∞Server incrementally pulls the geometry files from the connector and builds an optimized package.
- This *build* is then published on the ∞**Proxies** associated to this ∞Server and every connected ∞Proxy then starts to replicate the build locally, in order to obtain a cascade publication.



DMU flow

The Connector must be specified and realized upon your requirements. 3D Juump Infinite only provides interfaces and helpers to feed the ∞Server with such data. In the figure, the enterprise DMU data source providers and your Connector are in gray.

During this process, the ∞**Directory** keeps in touch with every server, be it the ∞Server or the network of relay ∞Proxies. Thus, it is able to handle 3D Juump Infinite client applications connection requests by routing them to the proper ∞Proxy.

For the sake of simplicity, we have described *one* DMU flow. 3D Juump Infinite let you define *several* DMU flows built upon your combination of Connectors bound to your DMU data source providers, attached to several ∞Servers and ∞Proxies.

Integration

1 - Introduction

This chapter describes all the concepts and associated API¹² needed for the development of a **Connector**. It explains:

- what a *project* is and how it is declared,
- what the *Connector API* is and how the *Connector* interacts with it,
- how the *Connector* fleshes a *project* out by feeding the ∞Server with different types of data through an HTTP API (product structure, metadata, effectivities...),
- how the *Connector* can configure and drive the publication of a *project* by the ∞Server using through an HTTP API,
- what are *geometric sourcers*, how the ∞Server relies on them to access the (heavy weight) geometric data and how the *Connector* can configure such sourcers for its particular needs.

Finally, this chapter gives a quick how-to guide on how to develop your own *Connector*.

¹² Application Programming Interface

2 - Project

In 3D Juump Infinite, a *project* is an incremental set of data describing a DMU. It usually corresponds to a product (or product-line) level assembly. For instance, a vehicle manufacturer would probably opt for one *project* per car model so that, in the 3D Juump Infinite client application, the user would be able to configure and browse a set of variations of cars of the same model.

2.1 - Create a Project

To create a *project*, the administrator can log into the ∞Directory web application and process to the [\[PROJECT\]](#) section. The creation of a *project* requires several parameters:

- an existing, up and running ∞Server,
- a human-readable *project* name,
- a unit, either:
 - meter,
 - decimeter,
 - centimeter,
 - millimeter,
 - inch.

A newly created *project* is empty; it does not contain any data describing the DMU. The **Connector** is in charge with the correct feeding of the ∞Server database with DMU information. It does this by using the Connector API.

Note: it is also possible to programmatically create a new *project* through the *control interface*.

2.2 - Access a Project

Once a project has been created, it appears in the project list of the [\[PROJECT\]](#) section. It has also been granted a unique internal identifier by the ∞Server. This *project identifier*, of the form *prj_* followed by a hexadecimal string, is used to access the *project interface*.

2.3 - Build a Project

Once the *Connector* is done with feeding a *project*, it can either programmatically trigger the creation of a new snapshot of the DMU (or *build*) or let the administrator manually trigger it through the dedicated interface in the [\[PROJECT\]](#) section of the ∞Directory.

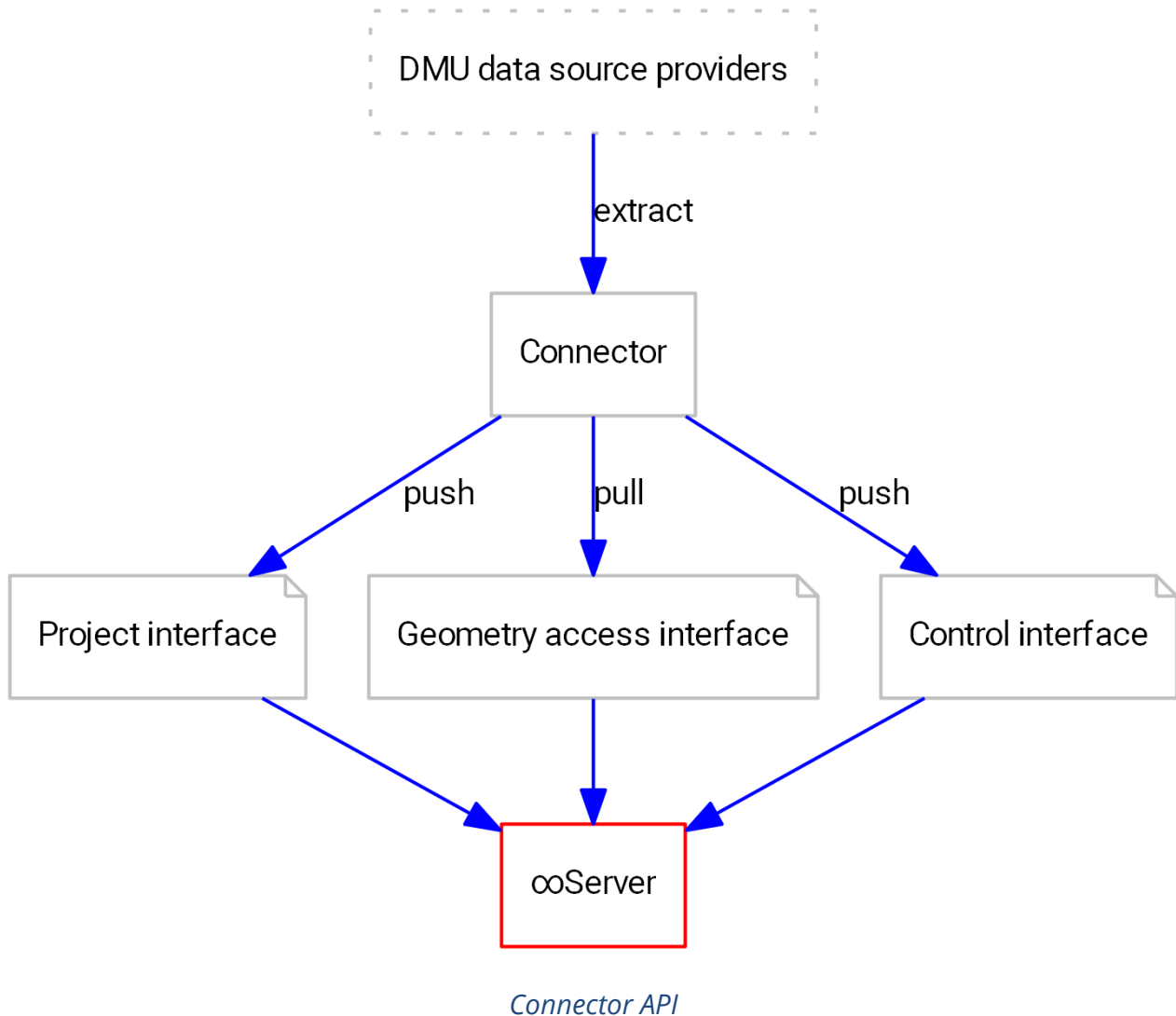
3 - Connector API

The **Connector API** is threefold:

- *Project interface*: this interface enables the connector to declare project-related data, including structure data, metadata and effectivity data but also project-specific parameters and customization scripts. Thanks to this interface, the connector *pushes* data to the ∞Server.

- *Geometry access interface*: through this interface, the ∞Server *pulls* project-related geometry data from the connector. The connector must choose one of the existing implementations.
- *Control interface*: this interface enables the connector to drive the ∞Server programmatically through several HTTP end-points. In particular, it makes it possible to manage the projects (create, delete, build, etc.).

These three interfaces are detailed in the following chapters.



4 - Project Interface

4.1 - HTTP API

For each project, the ∞Server provides a RESTful¹³ API based on the exchange of JSON¹⁴ documents over HTTPS. It relies on [ElasticSearch](#) for this aspect, thus for more information on the API, please refer to the ElasticSearch documentation.

4.1.1 - Locate the project interface

The *project interface* is available as an ElasticSearch index on the ∞Server. It is accessed over HTTPS using the ∞Server's IP address and HTTPS port, the ElasticSearch end-point path and the ∞Server's PostgreSQL authentication (ex: https://postgres:*****@172.16.254.1:443/elastic). It can host several projects, accessible through their unique identifier. To reach the *Project interface* for a given *project*, the following information is needed:

- the *address* of the ∞Server,
- the HTTPS *port* of the ∞Server (default: [443](#)),
- the ElasticSearch end-point *path* (default: [/elastic](#)),
- the *project identifier* suffixed with [_connector](#),
- credentials for a valid PostgreSQL user.

For example, the project "supercar", internal identifier "prj_02ef8964c5d", hosted on a ∞Server listening on the standard HTTPS port on an IPv4 interface at 192.168.1.2 with default postgres credentials and default end-point configuration would have the address: https://postgres:*****@192.168.1.2:443/elastic/prj_02ef8964c5d_connector.



Other indexes may live on the same ElasticSearch instance and should not be messed with. They are indexes used by the ∞Server for internal purpose. In particular, one must pay attention to the proper naming of the *project interface* end-point and not forget the [_connector](#) suffix.

💡 The *project interface* being an ElasticSearch index, many useful ElasticSearch features are available to the connector for fast querying & updating. Feel free to customize the *project interface* index mapping as you see fit. However, the mapping must at least contain the following fields:

```
{
  "mappings": {
    "_doc": {
      "properties": {
        "id": {
          "type": "keyword"
        },
        "type": {
```

¹³ Representational State Transfer

¹⁴ JavaScript Object Notation


```

        "type": "keyword"
      },
      "type": {
        "type": "keyword"
      },
      "ts": {
        "type": "long"
      }
    }
  }
}

```

4.1.2 - Declare a new document

To declare a document to the ∞ Server, the connector simply needs to *PUT* the document to the Elasticsearch index under the `_doc` type. Using [curl](#), the command line would look like:

```
curl -XPUT "https://postgres:*****@192.168.1.2:443/elastic/prj_02ef8964c5d_connector/_doc/supercar" -d '{"id": "supercar", ...}' -H "Content-Type: application/json"
```

Request:

```

PUT /supercar HTTP/1.1
Accept: application/json
Content-Length: ...
Content-Type: application/json
{
  "id": "supercar",
  "type": "structure",
  ...
}

```

Response:

```

HTTP/1.1 201 Created
Content-Length: ...
Content-Type: application/json
{
  "_index": "prj_02ef8964c5d_connector",
  "_type": "_doc",
  "_id": "wheel",
  ...
}

```



The *id* of a document must not begin with an underscore.

4.1.3 - Update an existing document

To partially update a document, the connector can *POST* the partial content to the document URL followed by `/_update`.

```
curl -XPOST "https://postgres:*****@192.168.1.2:443/elastic/prj_02ef8964c5d_connector/_doc/supercar/_update" -d '{"color": "red", ...}' -H "Content-Type: application/json"
```

It is also possible to use *scripted updates* and *upserts* as per [ElasticSearch documentation](#). Update by query is also supported, which enables the fast modification of several matching documents at once.

4.1.4 - Delete an existing document

Deleting a document is very similar to updating it. To delete a document, the connector needs to send a *DELETE* request to the document URL.

```
curl -XDELETE "https://postgres:*****@192.168.1.2:443/elastic/prj_02ef8964c5d_connector/_doc/supercar"
```

Delete by query is also supported.

4.1.5 - Retrieve a document

For debugging purpose, it could be useful to retrieve the content of an existing document by issuing a *GET* request to the document address on the ∞Server.

```
curl -XGET "https://postgres:*****@192.168.1.2:443/elastic/prj_02ef8964c5d_connector/_doc/supercar"
```

Response:

```
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/json
{
  "id": "wheel",
  "type": "structure",
  ...
}
```

4.1.6 - Mandatory fields

4.1.6.1 - ID

Each document must be uniquely identified thanks to its *id* field. The *id* field must not contain any dot (.).

4.1.6.2 - Type

Each document must have a *type* field. The value of this field depends on the type of the document (structure, metadata, etc.).

4.1.6.3 - Timestamp

Until version 3.0, 3D Juump Infinite relied on CouchDB revision mechanism for replication and completion check. From 3.0 and on, the revision mechanism has been replaced with a per-document connector-defined timestamp.

In addition to the mandatory *id* and *type* fields, documents shall also contain a *ts* field. This field must be an integer in the range $[0..2^{63}]$, must evolve with each update of the document, and must define a precedence order - the higher the *ts*, the most recent the document version.

4.1.6.4 - Metadata fields

As detailed in the following chapters, several documents feature a *metadata* object field with customizable content. The field keys of the *metadata* objects must begin with an alphanumeric character ('a-z', 'A-Z' or '0-9') and must not contain dots ('.').

4.2 - Data documents

4.2.1 - Types of data documents

The ∞Server database stores numerous documents that, together, describe the product structure, enriched with metadata and configuration information. There are several types of document that the connector should provide for the ∞Server to properly build a DMU.

The DMU product structure and metadata can be defined with these documents:

- Structure documents: describe how parts are assembled from other parts (product structure tree and transforms) or, for single parts, what representation to use (link to a geometry document)
- Geometry documents: describe how 3D Juump Infinite builds the geometrical representation of each single part
- Metadata documents: list the *metadata* information associated to each part

Other documents are useful to enrich the structure with additional data:

- Annotation documents: describe *annotations* attached to a part and visible in its 3D environment
- Attached documents: allow to embed or reference (by using an URL) external documents and attach them to *structure* or *annotation* documents
- Raster documents: used to define *raster* catalog needed to render raster annotations

Moreover, if you have more than one version/configuration of the DMU, you should fill the effectivity & configuration documents:

- Effectivity documents: list the effectivity information associated to each instantiation link (the link between an assembly-part and its components).
- Configuration documents: configurations of the DMU in the form of a list of active effectivities.

Basically, in the 3D Juump Infinite client application, once the user selects a configuration, the instantiation links that do not match the corresponding effectivities are automatically deactivated. This results in a filtered (or *configured*) product structure.

4.2.2 - Structure document

The documents of type *structure* describe how parts are assembled from other parts (or how they are represented by geometries, for final *Leaf* parts). These documents are mandatory. The granularity is usually that of the source *parts*: for each part of the product structure, there should be exactly one *structure* document.

Here is the typical content of a structure document:

```

{
  "id": <id of the structure document>,
  "type": "structure",
  "ts": <timestamp>,
  "partmdid": <id of the metadata document>,
  "attacheddocuments": [<id of attached document>, ...],
  "annotationdocids" : [<id of annotation document>, ...],
  "children": { // (only for assembly)
    "<linkmdid>": { // <id of the child instantiation link, may po
int to an effectivity document>,
      "hasmetadata" : false,
      "ref": <id of the child structure document>,
      "translation": [<tx>,<ty>,<tz>],
      "rotation": [
        <r00>,<r01>,<r02>,
        <r10>,<r11>,<r12>,
        <r20>,<r21>,<r22>
      ],
      "translationb" : "3 floats or doubles in little-endian Bas
e64 representation",
      "rotationb" : "9 floats or doubles in little-endian Base64
representation",
      "reflection": <true or false>
    },
    ...
  ],
  "geometry": <id of the geometry document> // (only for single part
)
}

```

For structure documents, the *type* field must be set to "structure".

- The *id* field must contain the unique identifier for the part in its current FFF¹⁵. It usually maps to the inner identifier used by the enterprise CAD data management system. The id should not start with a '_'.
- The *partmdid* field identifies the *metadata* document of this part. It is usually set to "partmd_<id of the part>". For example, if the part is identified using the enterprise-defined id 02FFE4546A0093, the structure document would have its *id* set to "02FFE4546A0093" and its *partmdid* set to "partmd_02FFE4546A0093". Note that the same *partmdid* must only correspond to one *structure* in any given product structure.
- The *attacheddocuments* field is an array of *attacheddocument* ids.
- The *annotationdocids* field is an array of *annotation* ids. All those *annotation* documents will be retrieved and positioned using cumulated transform to this *structure* item.

¹⁵ Form, fit and function

- Depending on whether the part is described as an assembly (a node in the product structure) or a single part (a leaf in the product structure), a structure document must contain:
 - either a *children* field
 - or a *geometry* field

4.2.2.1 - Assembly (with children)

The *children* field is a map of objects. Each element corresponds to a sub-part of the assembly. It lists:

- ref*: the id of the structure document corresponding to the sub-part.
- translation*, *rotation* and *reflection*: the 3D transformation of the sub-part relative to its parent. These values are float written as a text string.
- hasmetadata*: if true, tells that the child *instantiation link* key does point to an existing *effectivity* document. This field is optional and defaults to *false*.

For example, the following structure document declares a part "car" which is defined as the assembly of four wheels:

```
{
  "id": "car",
  "type": "structure",
  "ts": 1522236968,
  "partmdid": "partmd_car",
  "children": {
    "front-right-wheel": {
      "ref": "wheel",
      "translation": [-1.5,1,0]
    },
    "front-left-wheel": {
      "ref": "wheel",
      "translation": [-1.5,-1,0]
    },
    "rear-right-wheel": {
      "ref": "wheel",
      "translation": [1.5,1,0]
    },
    "rear-left-wheel": {
      "ref": "wheel",
      "translation": [1.5,-1,0]
    }
  }
}
```

Optionally, in order to mitigate the floating-point error which occurs during text transcription, it is possible to pass the translation and orientation as binary IEEE 754-1985 format floating-point values rather than text value. To this end, use *translationb* and *rotationb* instead of *translation* and *rotation* (mind the final *b*, for *binary*). To be compliant with the JSON format, these binary values must be ordered in little-endian endianness and coded in Base64. Note that the text-based and IEEE-based fields are supposed mutually exclusive. However, should binary

values be declared along with their text counterparts, their priority is higher: any binary value overrides a text one.

4.2.2.2 - Single part (with geometry)

The *geometry* field records the geometry document id used to retrieve the geometry file representing this single part. The geometry document is described in detail in the [corresponding](#) section.

Following on the above example, this document declares the “wheel” single part whose 3D representation is detailed in the “model_wheel” *geometry* document:

```
{
  "id": "wheel",
  "type": "structure",
  "ts": 1522236968,
  "partmdid": "partmd_wheel",
  "geometry": "model_wheel"
}
```

4.2.3 - Geometry document

The documents of type *geometry* describe how the ∞Server builds the 3D representations of final *Leaf* parts. They provide a flexible way to resolve file paths/urls and allow to finely tune the assembly and optimization of the DMU.

Here is the typical content of a geometry document:

```
{
  "id": "model_wheel",
  "type": "geometry",
  "ts": <timestamp>,
  "geometrysettings": {
    "sourcer": "supercar_repository",
    "path": "wheels/my_wheel_model.stp",
    "loginfo" : "optionnal informations that will be logged in case of error"
  }
}
```

For geometry documents, the *type* field must be set to “*geometry*”.

- The *id* field must contain the unique identifier for the 3D representation, as referenced by the [single part](#)’s *structure* document. It usually maps to the inner identifier used by the enterprise CAD data management system. The id should not start with a ‘_’.
- The *geometrysettings* field tells the ∞Server how to access and process the corresponding 3D model. Typically, the *geometrysettings* field combines two pieces of information:
 - It passes any access information needed for the *geometry sourcer* to find the corresponding model file. The *geometry sourcer* concept is detailed in the [corresponding chapter](#).

Basically, a sourcer represents a physical repository where the 3D models are stored. It can either points to a folder on a file-system or to a remote HTTP-server, for instance. Each sourcer has its own way of resolving 3D model references, but the usual way is to rely on a ``path`` field in the ``geometrysettings`` block.

- Optionally, it may override the default settings defined in the *project status* document (see the [corresponding chapter](#)). Each setting in the *geometrysettings* block overrides the default behavior for this very geometry.

Particularly, the ``geometrysettings`` block can specify which sourcer is to be used: the optional ``sourcer`` field can be set to override the default sourcer. For instance, if your DMU is built from two different sources of data (say, one repository for your own design and one for the procurement parts), you could declare two **geometry sourcers** and then use ``geometrysettings.sourcer`` on a part by part basis to tell the *Server* which source it resolves which CAD file.

`![[Updatetoc](icons/updatetoc)` Note that the ``etag`` field can also be overridden. If the **connector** is able to generate the etag for the geometries, it drastically speeds up the mirroring as etags are then compared directly without accessing the geometry files.

Hence, in the above example, the *"model_wheel"* geometry document instructs the *Server* to load the CAD model *wheels/my_wheel_model.stp* from the *geometry sourcer* named *supercar_repository*.

4.2.4 - Metadata document

The documents of type *metadata* list the information associated to each *part*. For every *structure* document describing a part, there *may* exist a corresponding *metadata* document.

Here is the typical content of a metadata document:

```
{
  "id": <id of the metadata document>,
  "type": "partmetadata",
  "ts": <timestamp>,
  "metadata": {...}
}
```

For metadata documents, the *type* field must be set to *"partmetadata"*.

- The *id* field must correspond to the *partmdid* listed in the corresponding structure document of the part. It is usually something of the form *"partmd_<id of the part>"*. The id must not start with an underscore ('_').
- The *metadata* field is an object containing the actual information in the form of *"key": <value>* pairs. Metadata key name must not contain any dot ('.').

For example, this metadata document decorates a “wheel” single part with several pieces of information extracted from your DMU data source providers, for instance your PDM¹⁶:

```
{
  "id": "partmd_wheel",
  "type": "partmetadata",
  "ts": 1522236968,
  "metadata": {
    "width": 185,
    "ratio": 75,
    "radial": true,
    "diameter": 14,
    "load-rating": 82,
    "speed-rating": "S",
    "instruction": "MAX LOAD SINGLE 2000 kg AT 760 kPs COLD"
  }
}
```

4.2.5 - Annotation document

The documents of type *annotation* are used to declare 3D annotations that will be positioned relatively to an instance (see the [corresponding chapter](#)). An *annotation* is a 2D shape display in 3D view. It could contain text or a picture.

4.2.5.1 - Annotation document

Here is the typical content of an annotation document:

```
{
  "id" : <id of the annotation document>,
  "type" : "annotation",
  "ts": <timestamp>,
  "attacheddocuments": [{"33333-33333-33333"}],
  "views" : {
    "MyType_A" : [ ],
    "MyType_B" : [ ]
  }
}
```

For annotation documents, the *type* field must be set to *"annotation"*.

- The *id* field will be referenced in *annotationdocids* field of *structure* documents. The id must not start with an underscore ('_').
- The *attacheddocuments* is identical to the one found in *structure* documents. It allows to attach external documents to all instances that will reference this *annotation* document (useful for extracting title blocks as HTML for instance).

¹⁶ Product Data Management

- The *views* field is a map of *annotationview* arrays indexed by type. Type of *views* will help user to select which *annotation* he wants to see.

4.2.5.2 - Annotation view

An *annotationview* is an entity that define orientation and base location of an annotation group. Here is the typical content of an *annotationview* object.

```
{
  "name" : "Right_view",
  "ver": "6.22",
  "origin" : [0, 0, 0],
  "axis" : [-1, 0, 0],
  "normal" : [0, 1, -0],
  "annotations": [],
  "isscalelocked": false
}
```

- The *name* field allow to specify a name to this view.
- The *origin* field is the origin of the view 3D basis.
- The *axis* field is the first axis of the view 3D basis.
- The *normal* field is the third axis of the view 3D basis (normal to annotation plane).
- The *annotations* field is the list of annotations in this view.
- The *ver* field denotes an internal version and must be set to "6.22",
- The *isscaleLocked* field allow to chose whether the annotations in this particular annotation view will scale depending on the distance to the camera or not.

4.2.5.3 - Annotation

An *annotation* can either be a text or a raster (i.e. an image). The following JSON presents the fields defining an annotation.

```
{
  "name" : "NOA_Bonding2.1",
  "color" : "ffffff",
  "frameshape" : "rectangle",
  "isframelocked" : false,
  "leaders" : [
    {
      "pos": [1., 2., 3.],
      "head": "circle"
    },
    ...
  ],
  "measures" : [
    {
      "A" : [70, -315, -32470],
      "B" : [76, -315, -32470],
      "V" : [0, 31, 0],
      "arrowdir": "outout"
    },
    ...
  ],
}
```

```

    "position" : [-108.603385925293, -214.4671630859375, 4289.80224609
375],
    "children": [<list of annotations>],
    "anchorloc": "abs",
    "type": "raster|text",
    "metadata": {...}
}

```

- The *name* field is the name of this annotation.
- The *color* field defines the color of the shape background.
- The *frameshape* field defines the type of shape. It should have one of the following values: *rectangle*, *square*, *flagright*, *flagleft*, *flagboth* or *parallelogram*.
- The *isframeLocked* field defines the orientation policy hint. If true, *annotation* will be displayed in the *annotationview* plane, if false *annotation* will be billboarded to always face screen. Note: this flag can be overridden by the end-user by tweaking the rendering settings of the 3D Juump Infinite application.
- The *leaders* field defines a list of 3D lines jutting out from the *annotation* frame. Each leader is a JSON object with the following fields:
 - The *pos* field defines the target position of the *leader* (expressed from the *annotationview* center). A 3D line will be displayed between the *annotation* center and the corresponding *leader* position.
 - The *head* field defines whether the *leader* ends with a distinctive shape. Supported values are: *none*, *square*, *circle*, *arrow*, *triangle*, *cross* and *plus*.
- The *measures* field define a list of *measures leaders*. *Measure leaders* are more specific than standard *leaders*, they are designed to show a distance between two 3D points. *Measure leaders* are JSON objects consisting of:
 - Two 3D points *A* and *B*.
 - A *v* vector indicating direction and length of the offset line. A 3D line will be drawn between the closest offset line and the *annotation* center.
 - An orientation for both arrows (*arrowdir*), either: *outout*, *inout*, *outin* or *inin*.
- The *position* field is the offset between *annotationview* center and the *annotation* shape center.
- The *children* field lists the *sub-annotations* attached to this *annotation*. *Sub-annotations* are similar to standard *annotations* except that their positions are relative to their parents'.
- The optional *anchorLoc* field is available for *sub-annotations* and allows to define the relative position of the *sub-annotation* with regards to its parent's shape. It can either be: *abs* (absolute), *bl* (below left), *bc* (below center), *br* (below right), *tl* (top left), *tc* (top center), *tr* (top right), *lb* (left below), *lc* (left center), *lt* (left top), *rb* (right below), *rc* (right center) or *rt* (right top).
- The *type* field can either be *text* (for *text annotations*) or *raster* (for *image annotations*).
- The *metadata* field contains all the metadata associated to this annotation (in particular, it contains the information read from the source CAD file).

Text annotation

Size of font and raster annotation are expressed in *em*. By default render define that 1em = 12pt at 96dpi.

The following JSON presents the specific fields of a *text annotation*.

```
{
  "defaultttextprop" :
  {
    "font" : "Arial",
    "heightem" : 0.8,
    "italic" : false,
    "bold" : false
  },
  "lines" :
  [
    [
      {
        "text" : "First segment of first line",
        "type" : "std"
      }, {
        "text" : "superscript",
        "type" : "sup"
      }, {
        "text" : "subscript",
        "type" : "sub"
      }
    ], [
      {
        "font" : "Courier New",
        "heightem" : 1.2,
        "italic" : true,
        "bold" : true,
        "text" : "Second line with different font",
        "type" : "std"
      }
    ]
  ]
}
```

- The *defaultttextprop* field defines default font properties
 - The *font* field specifies font name.
 - The *heightem* field specifies font height in em.
 - The *italic* field defines if font is italic.
 - The *bold* field defines if font is bold.
- The *lines* field is an array of array. Each sub array contains a list of segments and represents a line. Each segment could override *defaultttextprop* fields.
 - The *text* field contains text of this segment
 - The *type* field defines type of this segment (*std* for standard, *sup* for superscript and *sub* for subscript). Contiguous *sub* and *sup* segments will be display one above the other.

Raster annotation

The following JSON presents specific fields of a *raster annotation*.

```
{
  "rastername" : "raster_NOA_Bonding1_2",
  "sizeem" : [3, 3]
}
```

- The *rastername* field should reference *id* field of a *raster* document [corresponding chapter](#).
- The *sizeem* field defines [width, height] of the *annotation* shape in em.

4.2.6 - Attached document

The documents of type *attacheddocument* define a resource that will be attached to the product structure. Attached documents are available in *idcard*. The resource could be embedded with attached document or be referenced by using an url.

Here is the typical content of an attached document:

```
{
  "id" : <id of the metadata document>,
  "type" : "attacheddocument",
  "ts" : <timestamp>,
  "name" : "mydoc.txt",
  "description" : "a text document",
  "mime-type" : "text/plain",
  "url" : <location of the document data>,
  "datab64" : <document data base 64 encoded>
}
```

For attached documents, the *type* field must be set to *"attacheddocument"*.

- The *id* field will be referenced in *attacheddocuments* field of *structure* and *annotation* documents. The id must not start with an underscore ('_').
- The *name* field will be directly displayed to user in the *idcard*.
- The *description* field may contain additional informations to describe document content.
- The *mime-type* field is used to help handling of document. Some mime-types could be directly displayed into the application (if data is embedded).
- The document must contain either one of:
 - A *url* field that defines location of document data (for external document),
 - or a *datab64* field that contains the base64-encoded document data (for embedded document).

List of data type natively handled

- Plain text (*text/plain*) : data is displayed as utf-8 text.
- Rich text (*text/html*, *application/x-qt-richtext*) : data is display as qt compliant utf-8 rich text.
- Image (*image/jpg*, *image/jpeg*, *image/png*) : data is interpreted as picture data.

This document show how to embed a picture

```
{
  "id" : "attached_picture",
  "type" : "attacheddocument",
  "ts" : 15699846,
  "name" : "png_color_4x4",
  "description" : "a png sample with a resolution of 4x4",
  "mime-type" : "image/png",
  "datab64" : "iVBORw0KGgoAAAANSUgAAAAQAAAAECAIAAAAmkwkAAAAAXNSR0
IArs4c6QAAAARnQU1BAACxjwv8YQUAAAJcEhZcwAACxIAAAASAdLdfvwAAAAYdEVYdFNv
ZnR3YXJlAHBhaw50Lm5ldCA0LjAuNvyMY98AAAAxSURBVhXDcdBAQAgDAMxnBRnN2Udyj
qkQH5Z0kXdItpLDLSp8NODj13xT4Ycp5L9ACpgF836zffeAAAAAE1FTkSuQmCC"
}
```

This document show how to reference an external file

```
{
  "id" : "attached_odt",
  "type" : "attacheddocument",
  "ts" : 15699846,
  "name" : "open office document",
  "description" : "an external open office document",
  "mime-type" : "application/vnd.oasis.opendocument.text",
  "url" : "file:///Z:/folder/document.odt"
}
```

4.2.7 - Raster document

The documents of type *raster* define a picture. Those documents are referenced by raster annotations.

Here is the typical content of a *raster* document:

```
{
  "id" : <id of the raster document>,
  "type" : "raster",
  "ts" : <timestamp>,
  "datab64" : "...",
  "extension" : "png|jpg|jpeg"
}
```

For raster documents, the *type* field must be set to *"raster"*.

- The *id* field will be referenced by the field *rastername* of raster annotations. The id must not start with an underscore ('_').
- The *extension* field indicates format of picture data. It should take one of the following values : png, jpg or jpeg.
- The *datab64* contains picture binary data in base 64 format.

For example, this raster document define a colored picture saved in png:

```
{
  "id" : "raster_png_color_4x4",
  "type" : "raster",
}
```

```

    "ts" : 15699846,
    "datab64" : "iVBORw0KGgoAAAANSUhEUgAAAAQAAAAECAIAAAAmkwkPAAAAAXNSR
    0IARs4c6QAAAAARnQU1BAACxjwv8YQUAAAJcEhZcwAACxIAAAASAdLdfvAAAAAYdEVYdFN
    vZnR3YXJlAHBhaw50Lm5ldCA0LjAuNvyMY98AAAAxSURBVhXDcdBAQAgDAMxnBRnN2Udy
    jqkQH5Z0kXdItpLDLSp8NODj13xT4Ycp5L9ACpgF836zffeAAAAE1FTkSuQmCC",
    "extension": "png"
  }

```

4.2.8 - Effectivity document

The documents of type *effectivity* list the effectivity information associated to each *instantiation link*. That is, for every child link in a *structure* document, a corresponding *effectivity* document *may* exist and describe the effectivity constraints for this link. Later, in the 3D Juump Infinite client application, the user will be able to choose a configuration: then, only the links whose effectivities match the configuration will be assembled.



The effectivity document is not compulsory. If your digital mock-up does not expose any configuration information, you can dispense with creating and sending this document to the ∞Server.

Here is the typical content of an effectivity document:

```

{
  "id": <id of the effectivity document>,
  "type": "linkmetadata",
  "ts": <timestamp>,
  "effectivity": [
    {
      "field1": "value1",
      "field2": "value2",
      ...
    },
    ...
  ],
  "metadata": {...}
}

```

For effectivity documents, the *type* field must be set to *"Linkmetadata"*.

- The *id* field must correspond to the *instanciation link* key listed in the *children* field of the *structure* document. The id must not start with an underscore ('_').
- The *effectivity* field is an array of objects. Each object represents a set of effectivities in the form *"key": "value"* where *key* is a category of the effectivity option and *value* is a valid option value for this category. The valid category and option values are listed by the *effectivity dictionary* (see the next section). In 3D Juump Infinite client application, the corresponding link will be active when at least one of its sets of effectivities matches the selected configuration. Note that links that have no *effectivity* documents are considered not configurable and are thus always active whatever the configuration. Effectivity key name must not contain dots ('.').

- The *metadata* field is an object similar to the one found in *partmetadata* documents. It enables the link to bear information that will be shown in the *id-card* of the instance just like other metadata. Metadata key name must not contain dots ('.').

For example, this effectivity document describes the constraints that must be met for a car to assemble a "fullcarbon" wheel trim:

```
{
  "id": "linkmd_car_wheeltrim+fullcarbon",
  "type": "linkmetadata",
  "effectivity": [
    {
      "model": "GT"
    },
    {
      "model": "ST",
      "trim": "special"
    }
  ],
  "metadata": {
    "catalog": 2017
  }
}
```

In this example, the car will only assemble this *wheeltrim+fullcarbon* wheel trim if the selected configuration is either:

- a GT model
- a ST model with special trims

Written with logical operators, we have : (model=="GT") OR (model=="ST" AND trim=="special")

The categories which are not listed in a set of effectivities are not taken into account for a link configuration activation. Thus, in the previous example, the "fullcarbon" wheel trim would be assembled even if the car configuration mentions, in addition to special wheel trim, that the GT model car must have a sunroof and a turbocharged engine. Since the effectivity values for the sunroof is not explicitly mentioned in this link, the sunroof effectivity category has no impact on the "fullcarbon" wheel trim instantiation.

Also note that the "catalog" field inherited from the *instanciation link* will be displayed in the id-card of this fullcarbon wheel trim instance in addition to the regular part metadata.

4.2.9 - Configuration document

A document of type *configuration* describes a valid configuration of the DMU in the form of a list of active effectivities. In the 3D Juump Infinite client application, the configurations will be presented as predefined read-only selectable items in a list of configurations. When the user selects one of these configurations, the DMU will be instantiated with the links which effectivities match with the ones enumerated in the *configuration document*.

Note : these predefined configurations will only appear after requesting a new build.



The configuration document is not compulsory. If your digital mock-up does not expose any configuration information, you can dispense with creating and sending this document to the ∞Server.

Here is the typical content of a configuration document:

```
{
  "id": "com.3djuump.conf:<id of the configuration>",
  "type": "conf",
  "ts": <timestamp>,
  "name": "some name",
  "description": "free text",
  "effectivity": {
    "field1": ["value1", "value2"],
    "field2": "value3",
    ...
  }
}
```

For configuration documents, the *type* field must be set to "conf".

- The *id* field must begin with *com.3djuump.conf:* and finish with a unique identifier for the configuration document. The id must not start with an underscore ('_').
- The *name* field contains a name labeling the configuration .
- The *description* field contains a description of the configuration.
- The *effectivity* field is an object with each of its field being a category of the effectivity option. The value of a given category must be either a string or an array of strings corresponding to the only valid option(s) for this configuration.

For example, this configuration document describes a particular set of options for a car:

```
{
  "id": "com.3djuump.conf:luxury",
  "type": "conf",
  "ts": 1,
  "name": "GT Moth Turbo II",
  "description": "Luxury high-performance car",
  "effectivity": {
    "model": "GT",
    "spoiler": "active",
    "horsepower": ["220", "250"]
  }
}
```

4.3 - Project-related documents

Aside from these types of documents dedicated to the description of the product structure, the ∞Server database also stores project-related documents.

4.3.1 - Metadata index mapping document

Metadata is stored in *partmetadata* and *linkmetadata* json documents that are indexed using Elasticsearch. By default type of fields will be automatically discovered. To ensure that fields

are correctly interpreted it is possible (and advisable) to specify the index mapping. This is done thanks to the *indexmapping* project document. When this document is updated, project index will automatically be rebuilt.

Here is the typical content of an *indexmapping* document:

```
{
  "id" : "com.3djuump:indexmapping",
  "type" : "projectdocument",
  "subtype" : "indexmapping",
  "ts" : <timestamp>,
  "version" : "7.0",
  "metadatamapping" : {
    ...
  },
  "dynamic_templates" : [
    ...
  ]
}
```

The *metadatamapping* object will be used to build index of *metadata* object found in *partmetadata*, *linkmetadata* and other project documents. The *dynamic_templates* array enables the declaration of additional dynamic templates (see Elasticsearch documentation).

Here is an example of *indexmapping* document

```
{
  "id" : "com.3djuump:indexmapping",
  "type" : "projectdocument",
  "subtype" : "indexmapping",
  "version" : "7.0",
  "ts" : 1,
  "metadatamapping" : {
    "properties" : {
      "MyIntegerField" : {
        "type" : "integer"
      },
      "MyDoubleField" : {
        "type" : "double"
      },
      "MyDateField" : {
        "type" : "date",
        "format" : "date_optional_time"
      },
      "MySubObject":{
        "properties":{
          "MyDateField2" : {
            "type" : "date",
            "format" : "yy/MM/dd"
          }
        }
      }
    }
  },
}
```


```

    "dynamic_templates" : [{
      "SpecificMd": {
        "path_match": "metadata.SpecificMd",
        "mapping": {
          "index": false,
          "dynamic": false,
          "type": "object"
        }
      }
    }]
  }

```

- *properties* object list fields for which we want to specify index policy.
 - *type* field should match an existing Elasticsearch type (*text*, *integer*, *double*, *boolean*, *date*, *object*).
 - *format* for *date* type, format could be specified to help date parsing, see Elasticsearch documentation.
 - *properties* field describes types of subfields. Again, see Elasticsearch documentation.

Note for now only text (*text*), numeric (*Long*, *integer*, *short*, *byte*, *double*, *float*), boolean (*boolean*) and date (*date*) types are handled by the client. Text fields are treated specifically through a dedicated predefined template that enables full-text search: do not interfere with the default indexation parameters.

 Text fields potentially containing more than 32kB (8000 UTF-8 characters) must not be indexed: make sure to explicitly exclude them from the mapping.

5 - Geometry Access Interface

5.1 - Overview

The *geometry access interface* allows the ∞Server to retrieve the geometry CAD files referenced in the Product Structure. The actual storage of these CAD files depends on the existing infrastructure, hence 3D Juump Infinite must rely on a user-specific access method. It is up to you to select and configure the proper geometry access interface implementation, called a *geometry sourcer*.

In this section, we examine:

- how to setup the generation and its sourcers
- let it resolve geometry CAD file paths
- the supported CAD file formats

5.2 - Build setup ●

The *geometry access interface* is configured thanks to *buildparameters* documents. The default *buildparameters* document must have "*com.3djuump:buildparameters*" for id, but it is also possible to create other alternative *buildparameters* documents for more specific purposes and then to trigger a build using these parameters instead of the default.

```

{
  "id": "com.3djuump:buildparameters",
  "type": "projectdocument",
  "version": "7.0",
  "ts": <timestamp>,
  "subtype": "buildparameters",
  "buildparameters": {
    "buildcomment": <comment>,
    "tags": <array of tags>,
    "rootstructuredocid" : <id of the root structure document>,
    "applicableconfigurations": <null or array of ids>,
    "lowdefctrlcount" : 1000000,
    "visiblersets" : [10000, 20000],
    "modelaabblimit" : {
      "xmin" : -3.3e38,
      "xmax" : 3.3e38,
      "ymin" : -3.3e38,
      "ymax" : 3.3e38,
      "zmin" : -3.3e38,
      "zmax" : 3.3e38
    },
    "xformtolerance" : {
      "translation" : 0.01,
      "rotation" : 0.001
    },
    "sourcers" : {
      <sourcer name> : {
        "type" : <type of sourcer>
        ...
      },
      ...
    },
    "defaultgeometrysettings" : {
      "sourcer" : <default sourcer name>,
      "etag":null,
      "allowstaticlowdef" : true,
      "connexitythreshold" : 2,
      "minobjsizeforstaticlowdef" : 300,
      "visibilityvoxelizationstep" : 175,
      "maxvoxelcount" : 200000,
      "alldynamiclowdef" : true,
      "dynamiclowdefvoxelizationstep" : 50,
      "minobjsizefordynamiclowdef" : 175,
      "minheuristicfordynamiclowdef" : 0.6,
      "minheuristicprioritizeddynamiclowdef" : 0.98
    },
    "workertimeoutsec" : 900,
    "workermemorylimitMB" : 3000,
    "workercount" : 8
  }
}

```

The *buildparameters* contains generation settings.

- *buildcomment* contains a description that will be associated to the next build. This text will be presented to the users to help them identify the build.
- *tags* contains an array of tags (i.e. text strings, containing no colon ":") to decorate the build with. These tags will be then used for access and replication rights. Important note: for security reasons, **it is not possible to alter the tags of a build afterwards** so it is strongly advised to properly define the tags and their meaning in advance.
- *rootstructuredocid* contains the *structure* document identifier to use as the root of the product structure. This field is usually set once and for all at the beginning of the project.
- *applicableconfigurations* references the applicable *configurations*. It contains a list of *configuration* document ids. Note: if set to *null* or if absent, all available configurations are considered applicable; if set to *[]*, then no configuration is applicable.
- *lowdeftrlcount* is an integer value defining the triangle budget for the low-definition representation of the DMU. 1000000 is a good all-round number. This field is usually set once and for all.
- *visiblersets* is not documented.
- *modelaabblimit* is optional and defines the limit of the exploitable DMU during the client session. Your DMU will not be truncated at the given boundaries but in the ∞Client 3D view, the camera will be properly centered and the end user will be able to extract geometric data out of these boundaries.
- *xformtolerance* defines the tolerance of the matrix-matching algorithm in terms of position (*translation* in scene units) and orientation (*rotation* in degrees).
- *sourcers* is a dictionary of *geometry sourcers* the ∞Server will use to retrieve geometries as explained below.
- *defaultgeometrysettings* describes the default geometry settings. These settings are applied by default for each geometry part found. It can be overwritten by an entry specified in the *geometry* document. The values can mostly remain untouched, unless you have a DMU geometry expressed with an unit different from the millimeter. If your DMU is plagued with rogue geometric parts impeding the rendering performance, you can also tweak these settings.
 - *sourcer*: contains the name of the sourcer to use by default for all geometries - it must be one of the sourcers declared in the *buildparameters.sourcers* entry (see above),
 - *etag*: if null this field will be ignored, else its string content will be used as etag to check if geometry should be updated (in this case, the sourcer will not be used to retrieve the geometry's etag),
 - *allowstaticlowdef*: the part will be baked in the low level of detail DMU model,
 - *connexitythreshold*: absolute minimum distance in millimeters for which two triangles can be merged,
 - *minobjsizeforstaticlowdef*: any part with a bounding sphere radius below this distance in millimeter will not appear in the low level of detail DMU model,
 - *subpartlevel*: acceptable values [*root,body,terminalpart,geometry*], define at which level of source model ps geometry will be merged at export,
 - *visibilityvoxelizationstep*: is not documented, it is advised to set it to 175,
 - *maxvoxelcount*: is not documented, it is advised to set it to 200000,

- *allowdynamiclowdef*: the part will be treated in the dynamic level of detail DMU model,
- *dynamiclowdefvoxelizationstep*: is not documented, it is advised to set it to 50,
- *minobjsizefordynamiclowdef*: is not documented, it is advised to set it to 175,
- *minheuristicfordynamiclowdef*: is not documented, it is advised to set it to 0.6,
- *minheuristicctoprioritizedynamiclowdef*: is not documented, it is advised to set it to 0.98.
- *workertimeoutsec*: maximum time in second that a worker could spend to process a job (mirror, psconverter, postprocess).
- *workermemorylimitMB*: maximum memory in MBytes that a worker could use to process a job (mirror, psconverter, postprocess).
- *workercount*: number of jobs that will be processed simultaneously. This value should depend on number of available CPU/Memory on the server.

5.3 - Souncers setup

The *buildparameters.souncers* dictionnary lets you configure the sourcer(s) to use for CAD model retrieval. You can declare any number of sourcers, each one with a unique name (*<sourcer name>*).

A sourcer is defined as a JSON object with at least one field: its *type*. The current version of 3D Juump Infinite provides two types of sourcer. The *type* must be set with either:

- *FileSystemSourcer* for a file system sourcer,
- *HTTPSourcer* for a HTTP server sourcer.

Depending on the type of sourcer, other fields may be required (see below).

5.3.1 - File system sourcer

The sourcer you mainly might use is based on file system storage. It interprets the *geometrysettings* fields found in *geometry* documents as file paths. These paths are all relative to a root directory defined by the *baseurl* field of the sourcer JSON object. Whenever the ∞Server needs to retrieve a geometry, it first checks its own cache, comparing the recorded timestamp to the one provided by the file-system.

5.3.1.1 - Configuration

```
{
  "id": "com.3djuump:buildparameters",
  "type": "projectdocument",
  "version": "7.0",
  "ts": 16891696,
  "subtype": "buildparameters",
  "buildparameters": {
    "sourcers": {
      "my_fs_repository" : {
        "type": "FileSystemSourcer",
        "baseurl": "<base directory>",
        "copybeforeload": true
      }
    }
  },
}
```

```


    ...
  }
}

```

- *baseurl*: defines the base path for the geometry representation.
- *copybeforeload*: if true, the sourcer will copy file in a local tmp folder before loading it. It could increase load performances from network shared folders.

5.3.1.2 - Usage ●

For a *FileSystemSourcer* to correctly resolve a *geometry* document into a CAD representation, the *geometrysettings* block must contain either one of the following fields: - *path*: the sourcer loads the CAD model from the file-system using a combination of its *baseurl* and the provided *path*. - *absolutePath*: the sourcer loads the CAD model from the file-system using the provided *absolutePath* (the *baseurl* of the sourcer is not used).

 Make sure the ∞Server service account has rights to access the target folders.

5.3.1.3 - Example

Given the following *buildparameters*:

```

{
  "id": "com.3djuump:buildparameters",
  "type": "projectdocument",
  "version": "7.0",
  "ts": 168794687,
  "subtype": "buildparameters",
  "buildparameters": {
    "sourcers": {
      "supercar_repository" : {
        "type": "FileSystemSourcer",
        "baseurl": "//cars_repos/supercar"
      }
    },
    ...
  }
}

```

The *geometry* document below will be resolved using the CAD file located at `\\cars_repos\\supercar\\wheels\\my_wheel_model.stp`.

```

{
  "id": "model_wheel",
  "type": "geometry",
  "ts": 15697463,
  "geometrysettings": {
    "sourcer": "supercar_repository",
    "path": "wheels/my_wheel_model.stp"
  }
}

```

5.3.2 - HTTP server sourcer

Another useful sourcer works with a distant HTTP file server. It interprets the *geometrysettings* fields found in *geometry* documents as URI. These URI are all relative to a file-server base URL, defined by the *baseurl* field of the of the sourcer JSON object.

Whenever the ∞ Server needs to retrieve a geometry representation file, it relies on standard HTTP cache-control (HTTP *304 Not Modified* status code). Depending on the remote HTTP server capabilities, it will use one of:

- *ETag* and *If-None-Match* HTTP headers
- *Last-Modified* and *If-Modified-Since* HTTP headers

5.3.2.1 - Configuration

```
{
  "id": "com.3djuump:buildparameters",
  "type": "projectdocument",
  "version": "7.0",
  "ts": 169896143,
  "subtype": "buildparameters",
  "buildparameters": {
    "sourcers": {
      "my_http_sourcer" : {
        "type": "HTTPSourcer",
        "baseurl": "<base url of the HTTP server>"
      }
    },
    ...
  }
}
```

Other optional parameters can be specified:

- The *skipheader* and *skipfooter* parameters are optional. They are only useful if the file-server inserts error-code in the datastream. For instance, if the actual payload is preceded and followed by a 4-bytes long error-code, *skipheader* and *skipfooter* must both be set to 4.

5.3.2.2 - Usage

For a HTTPSourcer to correctly resolve a *geometry* document into a CAD representation, the *geometrysettings* block must contain either one of the following fields:

- *path*: the sourcer loads the CAD model from the HTTP server using a URL built from a combination of its *baseurl* and the provided *path*.
- *absolutepath*: the sourcer loads the CAD model from the HTTP server using the provided *absolutepath* as URL (the *baseurl* of the sourcer is not used).



For now, the only HTTP authentication mechanism supported by HTTPSourcer is *Basic Authentication* with the credentials passed by URL (*http://user:password@domain*) in the *baseurl* field.

5.3.2.3 - Example

Given the following *buildparameters*:

```
{
  "id": "com.3djuump:buildparameters",
  "type": "projectdocument",
  "version": "7.0",
  "ts": 16947511,
  "subtype": "buildparameters",
  "buildparameters": {
    "sourcers": {
      "supercar_repository" : {
        "type": "HTTPSourcer",
        "baseurl": "https://johndoe:jd03@cars.mycompany.corp/supercar"
      },
      ...
    }
  }
}
```

The *geometry* document below will be resolved using the CAD data located at URL https://johndoe:jd03@cars.mycompany.corp/supercar/wheels/my_wheel_model.stp.

```
{
  "id": "model_wheel",
  "type": "geometry",
  "ts": 15698463,
  "geometrysettings": {
    "sourcer": "supercar_repository",
    "path": "wheels/my_wheel_model.stp"
  }
}
```

5.3.3 -Supported CAD file formats

3D Juump Infinite is able to process the following formats:

- ACIS (.sat) - all => R21
- ASC Medusa 3D (.asc)
- CADDs (explicit parts) & CAMU (._pd, ._ps) - 4 & 5
- FBX (.fbx)
- I-DEAS (.arc,.unv) - all => NX5
- IGES (.igs) 5.2 & 5.3
- Inventor (.ipt,.iam) - all => 2017
- CATIA V4 (.model,.dlv,.exp,.session) - all 4.xx
- CATIA V5 (.CATPART,.cgr,.CATProduct) - R10 - R26
- CATIA V6 (.3Dxml) - 2011x => 2013x
- 3D Experience (.3Dxml) - 2014 => 2015x
- JT-Format (JtOpen) (.jt) 7.0 => 10.2

- Matra Euclid 3 (.e3i) - 3.2
- Nastran (.nastran)
- NX Unigraphics (.prt) - 11 => NX11
- OBJ (.obj, .mtl)
- Parasolid XT-Format (.x_t) - all => 28
- ProEngineer (.asc, .prt, .neu) - part files: 13 => Creo 4 (F000) / neutral files: 13 => WF5
- Rhino 3D (.3dm)
- ROBCAD (.rf)
- Solidworks (.sldprt, .sldasm) - 99 => 2017
- STEP AP203, AP214 and AP242 (.stp) - 203/214/242
- Straessle EUKLID (.edx)
- STL (.stl)
- VDA (.vda)
- VRML (.wrl, .wrz, .vrm) - 97
- 3DS (.3ds)

It reads the surface information only. All other informations are ignored.

If your geometry representation file format is not listed above, it will not be converted during the build and will not be integrated in the DMU build. In addition, the ∞Client will not be able to display these representations.

5.4 - Example

As stated in the introduction of this chapter, one can mix and match several sourcers and configure the ∞Server process on a part by part basis to achieve a fine tuned DMU preparation. Here is a basic example that hopefully wraps all these concepts up. We will not cover the *metadata* and *effectivity* documents in this short example and rather focus on the geometry processing aspects.

We are preparing the DMU of a car composed of four wheels and a steering wheel. The steering wheel is provided by one of our subcontractor and its CAD representation is shipped in the form of a STEP file located in a shared folder (`\\procurements\drive\steering.stp`). The body of the car and its four wheels are designed in house and are located on our own PLM system which offers a HTTP API to access them.

In the *buildparameters* document, we declare two different sourcers, one for our PLM repository and one for our procurements repository. Most of the parts we assemble are designed in house so we choose our own PLM repository as the default sourcer for the project. The *buildparameters* document looks like this:

```
{
  "id": "com.3djuump:buildparameters",
  "type": "projectdocument",
  "version": "7.0",
  "ts": 1597316165,
  "subtype": "buildparameters",
  "buildparameters": {
```

```

"sourcers": {
  "own_repository" : {
    "type": "HTTPSourcer",
    "baseurl": "https://plmaccess:jd03@myplm/supercar"
  },
  "procurements_repository" {
    "type": "FileSystemSourcer",
    "baseurl": "\\procurements"
  }
},

"defaultgeometrysettings" : {
  "sourcer": "own_repository",
  "addtolowdef" : true,
  "connexitythreshold" : 2,
  "minobjsizeforlowdef" : 300,
  "subpartlevel": "terminalpart",
  "visibilityvoxelizationstep" : 175,
  "visibilitymaxvoxelcount": 5000
},

"rootstructuredocid" : "supercar",
"buildcomment" : "My first car",

"lowdefctrlcount" : 1000000,
"visiblersets" : [10000, 20000],
"modelaabblimit" : {
  "xmin" : -3.3e38,
  "xmax" : 3.3e38,
  "ymin" : -3.3e38,
  "ymax" : 3.3e38,
  "zmin" : -3.3e38,
  "zmax" : 3.3e38
}
}
}

```

Note that we have set the *rootstructuredocid* to "supercar": this is the root *structure* document that defines our Product Structure. Here it is:

```

{
  "id": "supercar",
  "type": "structure",
  "ts": 15978331,
  "partmdid": "partmd_supercar",
  "children": {
    "body": {
      "ref": "body"
    },
    "steeringwheel": {
      "ref": "steeringwheel",
      "translation": [1,0.5,1]
    },
    "front-right-wheel": {

```

```

        "ref": "wheel",
        "translation": [1.5,-1,0]
    },
    "front-left-wheel": {
        "ref": "wheel",
        "translation": [1.5,1,0]
    },
    "rear-right-wheel": {
        "ref": "wheel",
        "translation": [-1.5,-1,0]
    },
    "rear-left-wheel": {
        "ref": "wheel",
        "translation": [-1.5,1,0]
    }
}
]
}

```

It references three other parts:

- the *body* of the car
- the *steeringwheel*
- and a *wheel* part that is instantiated four times

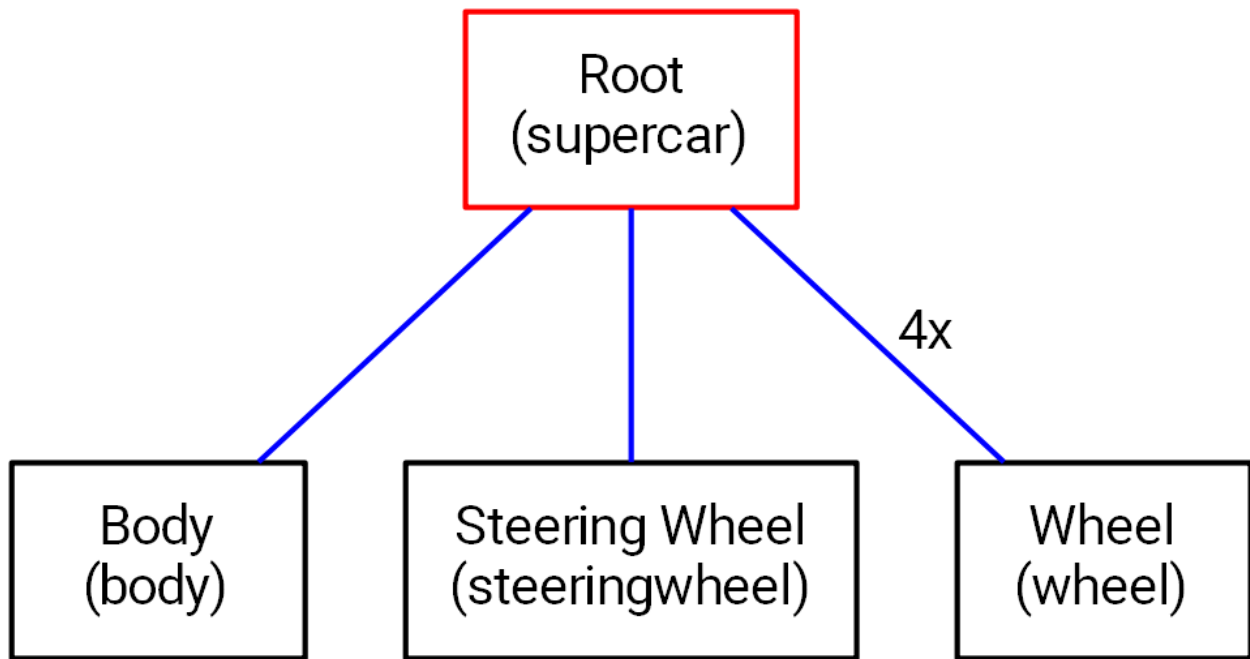
Here are the corresponding *structure* documents:

```

{
  "id": "body",
  "type": "structure",
  "ts": 1598761,
  "partmdid": "partmd_body",
  "geometry": "model_body"
}
{
  "id": "steeringwheel",
  "type": "structure",
  "ts": 1598762,
  "partmdid": "partmd_steeringwheel",
  "geometry": "model_steeringwheel"
}
{
  "id": "wheel",
  "type": "structure",
  "ts": 1598765,
  "partmdid": "partmd_wheel",
  "geometry": "model_wheel"
}

```

Note that, for simplicity sake, the three of them are single parts (with a *geometry* field) but of course 3D Juump Infinite supports complex Product Structure and intricate hierarchy of assemblies.



Simple Product Structure

Now we need to give a representation to these three single parts. For the body and the wheels, the *geometry* documents uses the default sourcer (our PLM):

```

{
  "id": "model_body",
  "type": "geometry",
  "ts": 16898461,
  "geometrysettings": {
    "path": "body/supercar.model"
  }
}
{
  "id": "model_wheel",
  "type": "geometry",
  "ts": 16898466,
  "geometrysettings": {
    "path": "wheels/my_wheel_model.CATPart"
  }
}

```

The steering wheel representation, on the other hand, comes from one of our subcontractors. The corresponding *geometry* document uses a specific sourcer. Moreover, it turns out that its CAD model is excessively heavy in terms of polygons and our use-case implies that we do not really need to see it all the time. To save up some of the polygon budget, we exclude this model from the “low-definition” DMU: the steering wheel will only become visible when the 3D Juump Infinite client application user comes near it. Here is the resulting *geometry* document:

```

{
  "id": "model_steeringwheel",
  "type": "geometry",
  "ts": 16898482,

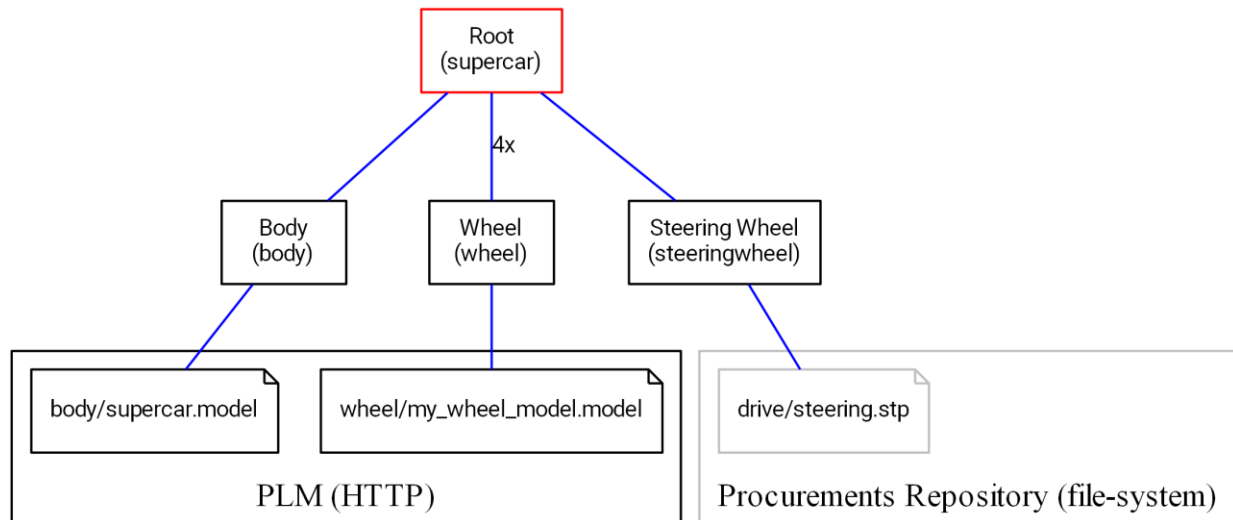
```

```

    "geometrysettings": {
      "sourcer": "procurements_repository",
      "path": "drive/steering.stp",
      "addtolowdef": false
    }
  }

```

Now we have the following DMU:



Simple DMU

6 - Control Interface ●

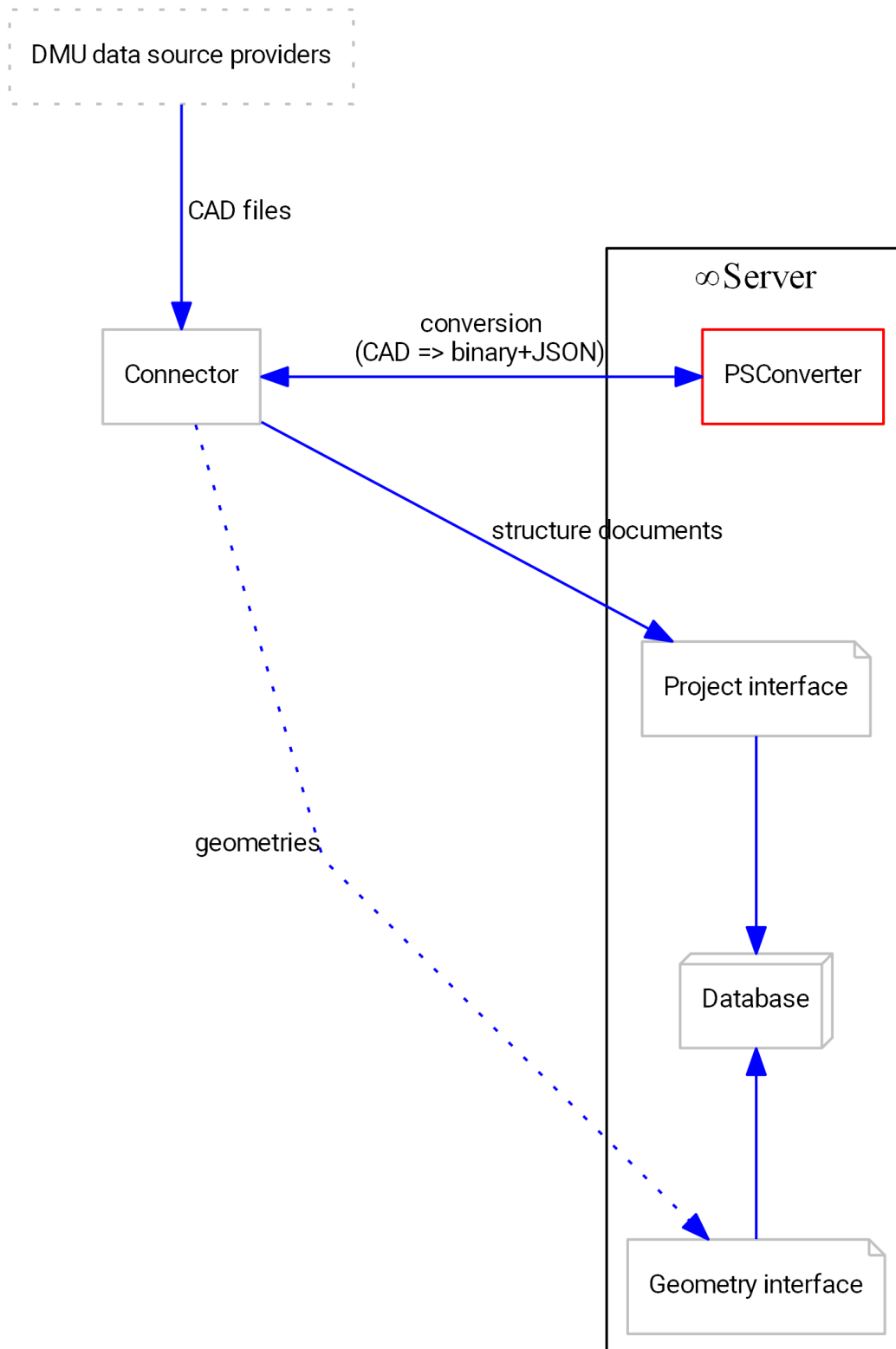
The *control interface* allows the *connector* to control how and when the ∞Server generates and publishes *builds*. This is achieved thanks to a web-based API. It is further described in a dedicated document.

7 - Product structure converter

7.1 - Introduction

As shown in the previous chapters, 3D Juump Infinite ∞Server expects the *connector* to provide it with several types of data: product structure data, metadata, configuration data and geometric data. Though geometric data can come in many different formats (see the list of supported formats), the other types of data are to be passed to the ∞Server in JSON format only.

Sometimes, your product structure definition data and metadata are stored in a set of CAD files, effectively unreadable without the right software, which makes it quite difficult to convert them to the expected JSON format. In order to ease the development of the *connector* and to assist you in extracting the product structure data and metadata from your CAD files, we provide a product structure converter helper (called **PSConverter** in the rest of document). The **PSConverter** is bundled with 3D Juump Infinite ∞Server and consists in one executable.



Product structure converter

In order to extract the product structure and metadata from CAD files, you basically pass the CAD files containing the product structure to the PSConverter executable. This helper then parses the files and returns the corresponding *structure* and *metadata* documents, plus any geometry data in binary blobs.

This PSConverter is provided as a helper, not as a *silver bullet* able to convert any data into 3D Juump Infinite *builds*. It should be completed with a proper *connector* developed upon your user requirements. In order to generate a DMU *build*, it is up to you to complete the documents associated to the geometry access and control interfaces, as explained in the corresponding chapters of this documentation.



For now, the PSConverter has been tested successfully on Dassault Systèmes CATIA V5 .catproduct/.catpart and Siemens .JT but more formats are to come.

7.2 - Using the PSConverter

The **PSConverter** executable takes a list of files to process and produces a set of JSON and binary files.

7.2.1 - Command-line

```
psconverter.exe -convert <jobs description file>
```

7.2.2 - Input

The jobs description file is a JSON document with the following structure:

```
{
  "system": {
    "workercount": 8,
    "maxramperworkermb": 4096,
    "directoryurl": "https://<directory host>/directoryapi/"
  },
  "jobs": [
    {
      "file": "d:/temp/myfile.CATPart",
      "rootid": "myfile-root",
      "extractannot": false,
      "extractannotoriginaldata": false,
      "extractmetadata": true,
      "extractlinkmetadata": true,
      "subpartlevel": "root",
      "rubfolder": "c:/infinitecache/myfile",
      "convresult": "c:/infinitecache/myfile/res.json"
    },
    ...
  ]
}
```

The *system* object contains the global runtime parameters:

- *workercount*: number of subprocess to spawn for batch treatment.
- *maxramperworkermb*: maximum memory available per worker (if a job ever exceed this number of MB of RAM, it is aborted).
- *directoryurl*: the URL of a valid ∞Directory API.

The *jobs* array contains job objects. Each job is defined thanks to:

- *file*: the path to the CAD file to process.

- *rootid*: the id to give to the root *structure* document that will be extracted from the CAD file.
- *extractannot*: whether the PSConverter should extract annotations from the CAD file (producing *annotation* documents).
- *extractannotoriginaldata*: whether the PSConverter should produce the raw annotation definition alongside the extracted annotations.
- *extractmetadata*: whether the PSConverter should extract metadata from the CAD file (producing *partmetadata* documents).
- *extractlinkmetadata*: whether the PSConverter should also extract link metadata from the CAD file (producing *linkmetadata* documents). Note that this flag has no effect if *extractmetadata* is set to false.
- *subpartLevel*: the granularity of the geometry extraction, must be one of:
 - *"nogeometry"*: no geometry will be extracted.
 - *"root"*: only one geometry is extracted for the whole CAD file.
 - *"body"*: produces one geometry per body found in the CAD file.
 - *"terminalpart"*: produces one geometry per terminal part found in the CAD file.
 - *"geometry"*: produces one geometry per body or terminal part found in the CAD file.
- *rubfolder*: the path of the folder where the geometry data are to be stored.
- *convresult*: the path of the file where the JSON documents are to be stored.

7.2.3 - Output

For each job, the PSConverter produces several files:

- one '.rub' file per geometry found (depending on the *subpartLevel* input parameter),
- one JSON file (whose name is set by the *convresult* input parameter) containing all the *structure*, *metadata* and *annotation* documents.

The JSON file has the following structure:


```
{
  "infos": {
    "convdate": "Tue Jun 26 14:23:58 2018",
    "converterversion": <version>,
    "extractannot": false,
    "extractannotoriginaldata": false,
    "extractgeom": "root",
    "extractmetadata": true,
    "extractlinkmetadata": true,
    "processingtimeins": 10.52499961853027,
    "rootid": "myfile-root",
    "ts": <timestamp>
  },
  "docs": [...]
}
```

The *infos* object contains information about the conversion:

- *convdate*: the date of the conversion.
- *converterversion*: the version of the PSConverter.

- *extractannot*: whether annotations were extracted.
- *extractannotoriginaldata*: whether raw annotation definitions were extracted.
- *extractmetadata*: whether part metadata was extracted.
- *extractlinkmetadata*: whether link metadata was extracted.
- *processingtimeins*: processing time (in seconds).
- *rootid*: the id of the root structure document.
- *ts*: timestamp of the conversion task.

The *docs* array contains all produced documents (*structure*, *partmetadata*, *linkmetadata*, *geometry*, *annotation*...).

 Important note: some *structure* documents may reference external CAD files. In this case, the child object will not contain a *ref* field but a *xref* field containing the path read from the CAD file. It is up to the connector to resolve these external references and to replace the *xref* field with a proper *ref* field.

8 - Migration

With each new version of the 3D Juump Infinite software, it is necessary to migrate existing *builds*. This is done through a specific migration procedure described in the administration manual. However, this procedure relies on the usage of a connector-defined script to migrate versioned project documents when needed (for example in case of API change). The mandatory entry-points in the *migration script* are detailed here below.

8.1 - Usage

```
3dJuumpInfiniteEvojuumpMigration -migrateevojuump <srcfile> <passphrase> <dst
file> <scriptfile>
```

Where:

- *srcfile* is the path to the evojuump file,
- *passphrase* is the corresponding passphrase as defined at the time of packing,
- *dstfile* is the path to the resulting migrated evojuump,
- *scriptfile* is the path to the migration script.

8.2 - Migration script

8.2.1 - Version

The *version_getVersion()* entry-point must return 1.

```
function version_getVersion()
{
    return 1;
}
```

8.2.2 - Patch document

The migration work itself is done in the `couchdb_patchDocument(doc)` entry-point. During migration, this entry-point is automatically called once for each document of the *project*.

8.2.2.1 - Input

Function arguments:

- `doc`: the content of the document being migrated.

8.2.2.2 - Output

This function must return the patched document (or `null` if no migration is needed for the given document).

```
function couchdb_patchDocument(pInputDoc)
{
  if(pInputDoc.type === "projectdocument" && pInputDoc.id === "com.3
djuump:scripts")
  {
    pInputDoc.scriptbase64 = "<old base64 script>";
    return pInputDoc;
  }
  else if(pInputDoc.type === "projectdocument" && pInputDoc.id === "
com.3djuump:converterscripts")
  {
    pInputDoc.scriptbase64 = "<patched base64 script>";
    return pInputDoc;
  }
  else if(pInputDoc.type === "projectdocument" && pInputDoc.id === "
com.3djuump:indexmapping")
  {
    if(pInputDoc.partmetadata !== undefined)
    {
      pInputDoc.metadatamapping = pInputDoc.partmetadata;
      delete pInputDoc.partmetadata;
      return pInputDoc;
    }
  }

  return null;
}
```

Customization

1 - Introduction

This chapter details how one can customize the way the 3D Juump Infinite client application displays the data (be it geometric data or metadata) thanks to user-defined documents and Javascript functions.

There are two documents that makes it possible to tune the behavior of the 3D Juump Infinite client application:

1. The *metadata usage customization* document is used by the 3D Juump Infinite client application (∞Client) to modify the metadata exploitation.
2. The *client default settings* document defines several ∞Client settings (display, export profiles...). These settings will be applied at the 3D Juump Infinite client on application start.

2 - Metadata usage customization

2.1 - Overview

The 3D Juump Infinite client application has several entry-points for customization. This is achieved thanks to external script functions that are called at different times during the

execution of the client application. The *customization script* may provide the implementation of many customization functions concerning:

- search
- ID card
- export
- filter

For further information on these ∞Client features, please refer to the 3D Juump Infinite user documentation.

For each project, there is exactly one *metadata usage customization* document that contains the *customization script* for the project. At the project creation, the script embedded in the *metadata usage customization* document is null, which means the 3D Juump Infinite client application maintains its default behavior.

2.2 - Customization document

The *metadata usage customization* document has a fixed id: "*com.3djuump:scripts*". Here is the typical content of this document:

```
{
  "id" : "com.3djuump:scripts",
  "type" : "projectdocument",
  "subtype" : "scripts",
  "version" : "7.0",
  "ts" : <timestamp>,
  "scriptbase64" : "...",
  "taskscripts": {
    "<task>": "...",
    ...
  }
}
```

The *scriptbase64* field must contain a JavaScript customization script. This script must be an **UTF-8** text content encoded in **Base64**¹⁷. The customization scripts for the different tasks are to be encoded likewise.

2.3 - Customization script


The customization script itself is a JavaScript based string providing the implementation of many functions regarding several features of 3D Juump Infinite client application.

2.3.1 -Version

The *version_getVersion()* function returns the version of the customization script. To run against the version 3.1 of 3D Juump Infinite client application, this function must return 7.

¹⁷ A radix-64 text encoding. [wikipedia](https://en.wikipedia.org/wiki/Base64)

```
function version_getVersion()
{
    return 7;
}
```

 It is compulsory to implement this function.

2.3.2 - User information

To further customize the client application, it may be useful to know which teams the user belongs to. Team information are available in javascript through a global variable named *sTeamsInfo*. It is an array containing an object for each team with its name and features (as defined in the ∞Directory's web application interface, see the corresponding chapter in the administration manual).

```
[
  {
    "name": <team name>,
    "features": [<list of features>]
  },
  ...
]
```

2.3.3 - Configuration information ●

Configuration information are also available through two variables:

- *sConfDict*: contains the dictionary of configurations,
- *sConfContext*: contains the array of active configuration ids.

2.3.4 - Attribute information

Before calling any script functions, the ∞Client gathers informations about all the available metadata. It is made available to the script through the *sAttributeInfo* global variable. This variable is a dictionary of all available metadata fields & subfields (or *attributes*) identified thanks to their dot-path notation (or *key*) - relative to the *metadata* block or to the nested block they belong to.

```
{
  <key> : {
    "nestedpath": <path of nested block>,
    "mappingtype": [numeric|bool|text|date|unknown]
  },
  ...
}
```

For each *key*, the following pieces of information are available:

- *nestedpath*: the path to the nested block containing this attribute.
- *mappingtype*: the type of attribute, either "*numeric*", "*bool*", "*text*" or "*date*".

2.3.5 - Attribute

The `attribute_getInfo` function defines how the `∞Client` manages and displays each attributes. The returned structure is a dictionary of all attributes (identified by their *key* just like in the `sAttributeInfo` structure described above).

```
{
  <key> : {
    "label": "Display name",
    "isquicksplithelper": false,
    "displaypriority": 0,
    "isinquickidcard": false,
    "hide":false
  },
  ...
}
```

For each *key*, the following fields are expected:

- *label*: the name that is displayed in the *filter* box when the user creates a *filter* on this *attribute* (default: *<key>*).
- *isquicksplithelper*: whether the *filter* bar should contain a *filter* on this *attribute* by default at start-up (default: *false*).
- *displaypriority*: manages the order of the default *filters* visible at start-up (the higher goes first, default: *0*).
- *isinquickidcard*: whether this attribute is visible by default in the *quick IDCard* (see below, default: *false*).
- *hide*: manage if this attribute will be displayed in auto completers and filter list (default: *false*).

```
function attribute_getInfo()
{
  var res = {};
  for (var k in sAttributeInfo)
  {
    lAttrInfo={
      "label": k,
      "isquicksplithelper":false,
      "displaypriority":0,
      "isinquickidcard":false,
      "hide":false
    };
    if (k == "Name")
      lAttrInfo.isinquickidcard = true;
    res[k] = lAttrInfo;
  }
  return res;
}
```

2.3.6 - Search

These functions customize the label to display for each search result.

2.3.6.1 - Search result label

The `search_computeResultLabel` function computes the search result label from its part/link metadata document.

It also generates a *literal filter* bound to this search result. In the ∞Client window, whenever an end-user drags this search result to a *layer* or *bucket*, it will create a *literal filter* to constraint the content of the given layer or bucket with the search result. See the *user manual* for more details on how *literal filter* expressions are formed.

```
function search_computeResultLabel(partmetadatadocument)
{
    return [<label>, <literal filter>];
}
```

2.3.6.2 - Search key set

The `search_getMetadataDocFilter` function filters the metadata that will be passed to the `search_computeResultLabel` function. This function can improve database retrieval performance by reducing the metadata volume transferred from the ∞Server/∞Proxy.

The returned value must be a valid ElasticSearch source filtering expression (see [ElasticSearch documentation](#)). For example, if you return `["metadata.*"]`, all metadata keys will be available to the function `search_computeResultLabel`. On the contrary, if it is empty, no metadata will be retrieved.

```
function search_getMetadataDocFilter()
{
    return ["metadata.description"];
}
```



It is strongly advised to limit the number of retrieved metadata fields to the ones effectively useful to the `search_computeResultLabel` function.

2.3.7 - Identification card ●

An identification card called *IDCard* is a list of *attributes* reflecting the *metadata* associated with a part instance. A *full IDCard* is the list of all the *IDCards* of a given part instance and of all its ancestors. The `idcard_compute` function is responsible for the construction of this list from the incoming *metadata*.

```
function idcard_compute(pMetadataHierarchies)
{
    var res = {};
    res.selectedBranchIndex = 0;
    res.hierarchicalMetadataBranches = [];
    ...
    return res;
}
```

2.3.7.1 - Input

As an input, the `idcard_compute` function receives an array of objects, each containing all the *metadata* of a given hierarchy branch.

```
[
  {
    "partmd" : [ <part metadata documents>... ],
    "linkmd" : [ <link metadata documents>... ],
    "hasannotations": [false,true,...],
    "hasattacheddocuments": [true,false,...]
  },
  ...
]
```

Several fields are available:

- `partmd` contains an array of [part metadata documents](#) ordered from top to bottom of product structure.
- `linkmd` contains an array of [effectivity documents](#) ordered from top to bottom of product structure.
- `hasannotations` is a top to bottom ordered boolean list, that indicates if a given level of product structure bears any *annotations*.
- `hasattacheddocuments` is a top to bottom ordered boolean list, that indicates if a given level of product structure bears any *attached documents*.

2.3.7.2 - Output

The `idcard_compute` function must return an object of the form:

```
{
  "hierarchicalMetadataBranches": [],
  "selectedBranchIndex": 0
}
```

The following fields are expected:

- `selectedBranchIndex` contains the index of the default selected hierarchy,
- `hierarchicalMetadataBranches` contains the *full IDCard* corresponding to each of the input hierarchy of metadata, that is to say an array of array of *IDCard* (starting from the root).

Each *IDCard* is an object of the form:

```
[
  {
    "name": "name of the level",
    "metadata": <metadata block>,
    "documents": <document block>,
    "cascadeattacheddocs": true,
    "cascadeannotations": true,
    "highlightlevel": 0
  },
  ...
]
```


Each *IDCard* object has the following fields:

- *name*: the header label that will be displayed for this level of product structure.
- *metadata*: the dictionary of *attributes* to display for this level of product structure (detailed below).
- *documents*: the dictionary of *documents* to display for this level of product structure (detailed below).
- *cascadeattacheddocs*: whether *attached documents* of this level should be cascaded to sub levels (default: true).
- *cascadeannotations*: whether *annotations* of this level should be cascaded to sub levels (default: true)
- *highlightlevel*: an integer in range [0;5] allowing to highlight this level in the hierarchy view. If missing, default to 0. Possible values are:
 - 0: no highlight,
 - 1: the instance is highlighted in *green*,
 - 2: the instance is highlighted in *yellow*,
 - 3: the instance is highlighted in *orange*,
 - 4: the instance is highlighted in *pink*,
 - 5: the instance is highlighted in *red*.

Metadata

As stated above, the *metadata* field contains a dictionary of *attributes* to display in the *IDCard*.

```
{
  <display name>: {
    "valuepath": "partmd.metadata.Name",
    "url": <url>,
    "highlightlevel": 0,
    "items": [...],
    "properties": {
      ...
    }
  },
  ...
}
```

Each entry correspond to one line of the *IDCard*, the key being the label displayed in the *IDCard* and the value containing a *description* of the way the attribute is displayed. The *description* contains at least:

- *valuepath*: the path (in dot notation) to the metadata to display and to use as *filter* when the user right-click on this *IDCard* entry. The path is relative to the input object described above.
- *url*: an optional url to make this entry clickable.
- *highlightlevel*: an optional integer in range [0;5] allowing to highlight this entry (default value: 0). Possible values are:
 - 0: no highlight,
 - 1: the entry is highlighted in *green*,

- 2: the entry is highlighted in *yellow*,
- 3: the entry is highlighted in *orange*,
- 4: the entry is highlighted in *pink*,
- 5: the entry is highlighted in *red*.

Depending on the type of the attribute, the *description* may also contain:

- for complex *attributes* containing an array of values:
 - *items*: an array of *description* sub-objects (with the regular *valuepath*, *url*, *highlightlevel*, etc.).
- for complex *attributes* containing a dictionary of values:
 - *properties*: a dictionary of *description* sub-objects.

Example:

```
{
  "Name": {
    "valuepath": "partmd.metadata.Name",
    "highlightlevel": 1
  },
  "Series": {
    "valuepath": "linkmd.metadata.MSN",
    "items": [
      {"valuepath": "linkmd.metadata.MSN.0"},
      {"valuepath": "linkmd.metadata.MSN.1"},
      {"valuepath": "linkmd.metadata.MSN.2"}
    ]
  },
  "Quality": {
    "valuepath": "partmd.metadata.Name",
    "properties": {
      "Litiges": {
        "valuepath": "partmd.metadata.quality",
        "items": [
          {"valuepath": "partmd.metadata.quality.3"},
          {"valuepath": "partmd.metadata.quality.0"},
        ]
      },
      "Concessions": {
        "valuepath": "partmd.metadata.quality",
        "items": [
          {"valuepath": "partmd.metadata.quality.1", "url":
"http://dqn.mycompany.org/13A4F5EEC3"},
          {"valuepath": "partmd.metadata.quality.2", "url":
"http://dqn.mycompany.org/45D92F0B34"}
        ]
      }
    }
  }
}
```

Note: for the moment, the *valuepath* field is only exploited for simple attributes and ignored for complex attributes (array or dictionary).

Document

As stated above, the `document` field contains a list of *documents* to display in the *IDCard* in the form:

```
{
  <name>: <url>,
  ...
}
```

where each document corresponds to one entry. The key is the display name of the document while the value is the URL to the document.

2.3.8 - Export

The `export_getPartBom` function customizes the 3D Juump Infinite client export features.

- The geometry can be exported as a file containing a hierarchy. To define the node naming of this exported hierarchy, you can assign the `id` property with your own customized label, given the part *partmetadata* document. The example function below builds the exported node label as a concatenation of two part metadata. This `id` must be unique.
- For the bill of material export, you can define the set of metadata to be exported. In the example below, the function filters out non interesting BOM metadata keys.

```
function export_getPartBom(partmetadatadocument, isleaf)
{
  var lResult = {};
  if (partmetadatadocument.metadata !== undefined) {
    var lAttributeKeys = ["model", "pn", "complexity", "type_mat", "srcfile"];
    for (var lAttributeKey in lAttributeKeys) {
      lResult[lAttributeKey] = partmetadatadocument.metadata[lAttributeKey];
    }
    lResult['_id'] = partmetadatadocument.metadata['model']+"."+partmetadatadocument.metadata['pn'];
    return lResult;
  }
}
```

2.3.9 - Formatting coordinate

The function `coordinates_getFormatting` customizes the 3D Juump Infinite client coordinate tool features. It define how the coordinates will be stored in the clipboard.

```
function coordinates_getFormatting(pX, pY, pZ)
{
  var map = {};
  map["text/plain"] = "[ " + pX + " ; " + pY + " ; " + pZ + " ]";
  return map;
}
```

2.4 - Task-specific scripts

The *taskscripts* entry is a dictionary of scripts, one for each customizable task type.

2.4.1 - Annotation task customization

"AnnotationTask": "<script in base64>"

The Annotation task supports a lot of customization. In particular, it lets the integrator defines its own annotation templates. This is achieved through the implementation of the following script entries.

2.4.1.1 - Version

The *getVersion()* entry point must return *1*.

```
function getVersion()
{
    return 1;
}
```

2.4.1.2 - Context information

For this script, several global variables are available, including [user information](#) and [attribute information](#). A specific *sCompleteIdCard* variable also bears the content of *IDCard* (same input as [idcard compute](#)).

2.4.1.3 - Annotation types

The *getAnnotationType()* entry point returns a list of strings, one for each type of annotations supported by this script.

```
function getAnnotationType()
{
    return ["Standard", "Quality", "Stress", "Report"];
}
```

2.4.1.4 - Raster data

If the customized annotation templates require the use of specific images (or rasters), the *getRasterData()* entry point is supposed to declare these rasters.

Each raster is defined as a JSON object:

```
{
  "id": "<unique identifier of the raster>",
  "name": "<label of the image>",
  "extension": "<file extension of the image>",
  "data": "<image encoded in base64>"
}
```

Supported extensions are:

- png,
- jpg

```
function getRasterData()
{
    var rasterData = [];
    rasterData.push( {id:"rasterInf", name:"Information", extension:
"png", data:"<base64 image data>"});
    rasterData.push( {id:"rasterAsk", name:"Question", extension:
"png", data:"<base64 image data>"});
    rasterData.push( {id:"rasterWar", name:"Warning", extension:
"png", data:"<base64 image data>"});
    return rasterData;
}
```

2.4.1.5 - Form editor

The `getFormEditor(type)` entry point is called each time 3D Juump Infinite tries to display an annotation form to the end-user. Its role is to provide a description of the fields to be displayed to the end-user.

Input

Function arguments:

- `type`: type of annotation to be edited.

Output

It must return a JSON array of UI items. Each UI item is defined as a JSON object with the following fields:

- `label`: the text displayed for the UI item.
- `type`: the type of UI item, either:
 - "string" (a single line of text),
 - "text" (a longer text),
 - "double" (a floating-point value),
 - "int" (an integer value),
 - "bool" (a boolean value, either true or false),
 - "enum" (a value taken from a set range of options),
 - "date" (a simple date),
 - "datetime" (a date and a time),
 - or "url" (a link to an external resource).
- `target`: the path where to store the user input is the annotation's idcard.
- `range`: an optional field restricting the lower and upper value of a value.
- `enum`: an optional list of available option values for this field (for `enum` fields), in the form of a JSON object with two values:
 - `type`: the actual type of the enum (either `raster` or `text`),
 - `value`: the value for the enum option.

```
function getFormEditor(type)
{
    var ui = [];
```

```

    if( type === "Standard")
    {
        ui.push({"label":"Title","type":"string","target":"title"});
        ui.push({"label":"Image","type":"enum","target":"rasterId","enum":[{"type":"raster","value":"rasterInf"}, {"type":"raster","value":"rasterAsk"}, {"type":"raster","value":"rasterWar"}]});
        ui.push({"label":"Comments","type":"text","target":"comments"});
    };
    }
    else if( type === "Quality")
    {
        ui.push({"label":"Title","type":"string","target":"title"});
        ui.push({"label":"Criticity","type":"enum","target":"criticity","enum":[{"type":"text","value":"Low"}, {"type":"text","value":"Medium"}, {"type":"text","value":"High"}]});
        ui.push({"label":"Fixed","type":"bool","target":"fixed"});
        ui.push({"label":"Comments","type":"text","target":"comments"});
    };
    }
    else if( type === "Stress")
    {
        ui.push({"label":"Title","type":"string","target":"title"});
        ui.push({"label":"Integer","type":"int","target":"int","range":[-25,50]});
        ui.push({"label":"Double","type":"double","target":"double","range":[0.75,12.65]});
        ui.push({"label":"Comments","type":"text","target":"comments"});
    };
    }
    else if( type === "Report")
    {
        ui.push({"label":"Title","type":"string","target":"title"});
        ui.push({"label":"date","type":"date","target":"myDate"});
        ui.push({"label":"datetime","type":"datetime","target":"myDateTime"});
        ui.push({"label":"Link","type":"url","target":"link"});
        ui.push({"label":"Comments","type":"text","target":"comments"});
    };
    }
    return ui;
}

```

2.4.1.6 - Information display

The `getInformationDisplay(type,metaAnnot,annot3D)` entry point is responsible for the creation of the annotation's idcard.

Input

Function arguments:

- `type`: the type of the annotation being displayed,
- `metaAnnot`: the content of the metadata attached to this annotation (see [above][#form-editor]),

- *annot3D*: the definition of the 3D annotation.

Output

It shall return a list of key/value pairs, one for each line of the idcard.

```
function getInformationDisplay(type, metaAnnotationJson, annotation3DJson)
{
    var ui;

    if( type === "Standard")
    {
        ui = {"header":metaAnnotationJson["title"], "listContent" : [{
"key":"Comments", "value":metaAnnotationJson["comments"]}]}];
    }
    else if( type === "Quality")
    {
        ui = {"header":metaAnnotationJson["title"], "listContent" : [{
"key":"Criticity", "value":metaAnnotationJson["criticity"]},{ "key":"co
mments", "value":metaAnnotationJson["comments"]}]}];
    }
    else if( type === "Stress")
    {
        ui = {"header":metaAnnotationJson["title"], "listContent" : [{
"key":"Integer", "value":metaAnnotationJson["int"]},{ "key":"Double", "
value":metaAnnotationJson["double"]},{ "key":"Comments", "value":metaAn
notationJson["comments"]}]}];
    }
    else if( type === "Report")
    {
        ui = {"header":metaAnnotationJson["title"], "listContent" : [
{"key":"Date", "value":metaAnnotationJson["myDate"]}, {"key":"Datetime
", "value":metaAnnotationJson["myDateTime"]}, {"key":"Link", "value":"
<a href=\""+metaAnnotationJson["link"]+"\">"+metaAnnotationJson["link"
]+"</a>"}, {"key":"Comments", "value":metaAnnotationJson["comments"]}]}
    }
}

return ui;
}
```

2.4.1.7 - Events

The main event function *onEvent(event, type, metaannot, annot3D, payload)* is responsible for the handling of user events.

Input

Function arguments:

- *event*: the type of event (either *create*, *update* or *migrate*),
- *type*: the type of annotation,
- *metaannot* & *annot3D*: the current definition of the annotation (metadata and 3D),

- *payload*, which is a JSON object depending on the type of event:
 - for *create* events: the payload contains the initial definition of the annotation, that is basically its *position* (as an array of three numbers) and orientation (*cameraDir*, *cameraRight* and *cameraUp*, each as an array of three numbers). Note that the current definition of the annotation is of no use in this case.
 - for *update* events: the payload contains the values entered by the end-user in the proposed form (see [above][#form-editor]).
 - for *migrate* events: the payload is empty.

Output

Whatever the type of event, this end-point shall return a valid annotation definition, that is to say its metadata and 3D definition.

```
function onEvent(eventType, type, metaAnnotationJson, annotation3DJson, payload)
{
    var lOut;
    if(eventType === "create")
    {
        lOut = onCreate(type, payload);
    }
    else if( eventType === "update")
    {
        lOut = onUpdate(type, metaAnnotationJson, annotation3DJson, payload);
    }
    else if(eventType === "migrate")
    {
        lOut = onMigrate(type, metaAnnotationJson, annotation3DJson);
    }
    return lOut;
}
function onCreate(type, payload)
{
    // METADATA
    var lMetaAnnotationData;
    var date = new Date();
    if( type === "Standard")
    {
        lMetaAnnotationData = {"title":"","rasterId":"","comments":"","creationDate": date.toISOString(), "modificationDate":date.toISOString()};
    }
    else if( type === "Quality")
    {
        lMetaAnnotationData = {"title":"","criticity":"Low", "fixed":false, "comments":"","creationDate":date.toISOString(), "modificationDate":date.toISOString()};
    }
    else if( type === "Stress")
    {
```



```

        lMetaAnnotationData = {"title":"","int":0,"double":5.56,"comments":"","creationDate":date.toISOString(),"modificationDate":date.toISOString()};
    }
    else if( type === "Report")
    {
        lMetaAnnotationData = {"title":"","myDate":"1985-11-13","myDateTime":"2017-05-22T17:57:56","link":"","photo":{},"comments":""};
    }
    // 3D
    var lAnnotation3D={
        "name":"","
        "axis":[1.,0.,0.],
        "normal":[0.,0.,1.],
        "origin":[0.,0.,0.],
        "annotations":[{
            "color": "ffffff",
            "defaulttextprop": {
                "bold": false,
                "font": "Arial",
                "heightem": 1.,
                "italic": false
            },
            "frameshape": "rectangle",
            "isframelocked": false,
            "leaders": [{
                "head": "circle",
                "pos": payload["position"]
            }
        ],
        "lines": [[{
            "text": "",
            "type": "std"
        }
        ]],
        "measures": [],
        "metadata": {},
        "name": "<the name of the annotation>",
        "position": payload["position"],
        "type": "text"
    },
    ],
    "ver":"6.22"
}
return [lMetaAnnotationData,lAnnotation3D];
}
function onUpdate(type,metaAnnotationJson,annotation3DJson,payload)
{
    // METADATA
    var date = new Date();
    if( type === "Standard")
    {
        metaAnnotationJson["title"] = payload["title"];
        metaAnnotationJson["rasterId"] = payload["rasterId"];
    }

```

```

        metaAnnotationJson["comments"] = payload["comments"];
        metaAnnotationJson["modificationDate"] = date.toISOString();
    }
    else if( type === "Quality")
    {
        metaAnnotationJson["title"] = payload["title"];
        metaAnnotationJson["criticity"] = payload["criticity"];
        metaAnnotationJson["fixed"] = payload["fixed"];
        metaAnnotationJson["comments"] = payload["comments"];
        metaAnnotationJson["modificationDate"] = date.toISOString();
    }
    else if( type === "Stress")
    {
        metaAnnotationJson["title"] = payload["title"];
        metaAnnotationJson["int"] = payload["int"];
        metaAnnotationJson["double"] = payload["double"];
        metaAnnotationJson["comments"] = payload["comments"];
        metaAnnotationJson["modificationDate"] = date.toISOString();
    }
    else if( type === "Report")
    {
        metaAnnotationJson["title"] = payload["title"];
        metaAnnotationJson["myDate"] = payload["myDate"];
        metaAnnotationJson["myDateTime"] = payload["myDateTime"];
        metaAnnotationJson["link"] = payload["link"];
        metaAnnotationJson["photo"] = payload["photo"];
        metaAnnotationJson["comments"] = payload["comments"];
        metaAnnotationJson["modificationDate"] = date.toISOString();
    }
    // 3D
    if( type === "Standard")
    {
        if(metaAnnotationJson["rasterId"])
        {
            if(annotation3DJson["annotations"][0]["type"] === "text")
            {
                delete annotation3DJson["annotations"][0]["defaulttext
prop"];

                delete annotation3DJson["annotations"][0]["lines"];
                annotation3DJson["annotations"][0]["type"] = "raster";
                annotation3DJson["annotations"][0]["sizeem"] = [2, 2];
            }
            annotation3DJson["annotations"][0]["rastername"] = metaAnn
otationJson["rasterId"];
        }
        else
        {
            annotation3DJson["annotations"][0]["lines"] = [{"text": m
etaAnnotationJson["title"], "type": "std"}];
        }
    }
    else if( type === "Quality")
    {

```

```

        annotation3DJson["annotations"][0]["lines"] = [[{"text": metaA
nnotationJson["title"],"type": "std"}]];
    }
    else if( type === "Stress")
    {
        annotation3DJson["annotations"][0]["lines"] = [[{"text": metaA
nnotationJson["title"],"type": "std"}]];
    }
    else if( type === "Report")
    {
        annotation3DJson["annotations"][0]["lines"] = [[{"text": metaA
nnotationJson["title"],"type": "std"}]];
    }
    return [metaAnnotationJson,annotation3DJson];
}
function onMigrate(type,metaAnnotationJson,annotation3DJson)
{
    return [metaAnnotationJson,annotation3DJson];
}

```



The “name” field of the 3D annotation is especially useful when copying an annotation from an Annotation task to a Presentation task.

2.5 - Utility functions

The Javascript environment running the above scripts also features several built-in functions beyond the standard ECMA functions:

- `btoa(input)` : returns its *input* encoded in base64.
- `atob(input)` : returns its *input* decoded from base64.
- `uuid()` : generate an uuid.
- `log(msg)` : prints the content of *msg* into application log.
- `fileexists(path)` : returns true if given file *path* exists.
- `folderexists(path)` : returns true if given folder *path* exists.

3 - Client default settings

The *client default settings* document assigns default values to a subset of 3D Juump Infinite client application settings. This document has a fixed id: `"com.3djuump:defaultsettings"`. Here is the typical content of this document:

```

{
  "id": "com.3djuump:defaultsettings",
  "type": "projectdocument",
  "subtype": "defaultsettings",
  "ts": <timestamp>,
  "version": "7.0",
  "settings": {
    "frameorientation": {
      "up" : [0.0, 1.0, 0.0],
      "dir": [0.0, 0.0, -1.0]
    }
  }
}

```

```

    }
    "backfaceculling": false,
    "fieldofview": {
      "degree": 25,
      "orientation": "vertical"
    },
    "dynamiclowdefprofiles": {
      "standard": 1048576,
      "high": 2097152
    }
  },
  "profiles": [<list of profiles>],
  "tasksettings": {
    "<task>": { <task settings> },
    ...
  }
}

```

Since this document already exists, you can update its values *backfaceculling* and *frameorientation* respectively for the backface culling and the frame orientation.

3.1 - Backface culling

Simply bind the boolean true or false to un(set) the backface culling during the rendering.

3.2 - Frame orientation

If your DMU orientation is not correct, you might try to rotate the up and direction vector around the right one to fix its orientation. You can specify the *up* and *direction* vectors. The value of the *right* vector will be calculated as the cross product $dir \times up$. These vectors are expressed in the default frame defined as:

- *right* : [1.0, 0.0, 0.0],
- *up* : [0.0, 1.0, 0.0],
- *direction* : [0.0, 0.0, -1.0]



Please remember the DMU content is not modified; only its view is. That implies any geometry will be exported with its incorrect but original frame. In the same way, the back face culling will not cull any face during the geometry export. It is up to you to propagate the frame orientation data or to cull the backfacing triangles thorough your different applications exploiting the exported data.

3.3 - Field of view

This setting allows to choose a specific field of view for a project. Field of view is specified in degrees for *vertical* or *horizontal orientation*.

3.4 - Dynamic low-def ●

This setting allows to tune the polygon budgets allocated to the dynamic low-def algorithm. It supports two budgets that the user will be able to chose from: *standard* and *high* that we advise to respectively set to 1048576 and 2097152.

3.5 - Profiles

This settings makes it possible to define project-wide export profiles. When a user explores the DMU of a given project, the settings of the ∞Client automatically populates with the corresponding project-wide export profiles for the user to chose from.

The *profiles* field is an array of profile objects. Each profile object has at least the following fields:

```
{
  "version": 1,
  "name": <readable name of the profile>,
  "type": <type of profile>,
  ...
}
```

- *version* is for internal use and must be set to 1,
- *name* is the name of the profile as seen by the user in the ∞Client,
- *type* is either:
 - *"geometry"* for geometric export profiles,
 - *"image"* for image export profiles,
 - *"metadata"* for metadata export profiles,
 - *"datapackage"* for datapackage export profiles,
 - *"presentation"* for presentation export profiles.

Depending on the *type* of a profile, other fields are expected.


3.5.1 -Geometric export profile

```
{
  "version": 1,
  "name": <readable name of the profile>,
  "type": "geometry",
  "formatname": <id of the codec to use>,
  "simplification": <simplification factor>,
  "commandline": <post-process command line>
}
```

- *formatname* is one of the supported geometric output format:
 - *"Format_3D_FBX"* for FBX format,
 - *"Format_3D_OBJ"* for OBJ format,
 - *"Format_3D_VRML"* for VRML format,
 - *"Format_3D_JT"* for JT format,
 - *"Format_3D_STEP+JT"* for composite STEP/JT format (the product structure is exported in STEP AP242 Part 21 and the geometries in JT),
 - *"Format_3D_STEP+WRL"* for composite STEP/VRML format (the product structure is exported in STEP AP242 Part 21 and the geometries in VRML),
 - *"Format_3D_WRL+WRL"* for composite VRML/VRML format.
 - *"Format_3D_WRZ+WRZ"* for composite compressed VRML/VRML format.

- *simplification* is a number denoting the desired simplification factor:
 - set to *0*, it means no simplification,
 - between *0* and *1*, it gives a *ratio* of simplification,
 - above *1*, it is interpreted as a *target* number of triangles.
- *commandLine* is a string defining an optional post-process command line. If left empty, it means no post-process. When not empty, its content will be interpreted as a command line in the context of the user's computer. In particular, all paths used in this *commandLine* field must exist on the target computers, thus it is advised that the post-process tools are either available as global executables (in the %PATH% environment variable on Windows, for instance) or on a common network file share.

Note: to reference the exported file in the command line definition, one can use the *%s* notation. 3D Juump Infinite will automatically replace *%s* with the actual path to the exported file before it executes the command line.

 The provided command-line will be executed as-is on the user's computer and in the user's context. It is your responsibility to make sure that the command-line is properly tested and does not alter the user's computer or any connected computers accessible in the user's context in any damaging way.

3.5.2 - Image export profile

```
{
  "version": 1,
  "name": <readable name of the profile>,
  "type": "image",
  "formatname": <id of the codec to use>,
  "highQuality": <generation mode>,
  "resolution": <predefined resolution>,
  "width": <width of the image in pixels>,
  "height": <height of the image in pixels>,
  "useBackgroundColor": <image background mode>,
  "backgroundColor": <image background color>,
  "commandline": <post-process command line>
}
```

- *formatname* is one of the supported image output format:
 - *"Format_Image_PNG"* for PNG format,
 - *"Format_Image_JPEG"* for JPEG format,
 - *"Format_Image_TIFF"* for TIFF format.
- *highQuality* is a logical value:
 - *false* means that the image is directly shot from the scene as seen by the user (geometry level of detail is not guaranteed, the background is set to the current 3D view background and the resolution is fixed to the 3D view dimensions),
 - *true* means that the generation process makes sure that every visible geometries is loaded in its highly detailed version and rendered at the chosen resolution - shooting high-quality images can be significantly longer but the level of detail is guaranteed and the final image background and resolution can be freely chosen (see below).

- *resolution* is a number denoting the desired predefined resolution:
 - *0* means that the resolution is not set and should match the current 3D view resolution,
 - *1* corresponds to 4096x2160,
 - *2* corresponds to 2560x1600,
 - *3* corresponds to 1920x1080,
 - *4* corresponds to 1600x1200,
 - *5* corresponds to 1280x720,
 - *6* corresponds to 800x600,
 - *7* corresponds to 320x200,
 - *8* means that the resolution is none of the above and should be customized thanks to the *width* and *height* fields.

Note : when *highQuality* is set to *false*, this field is ignored and defaults to *0*.

- *width* is the width of the image in pixels and is only useful with custom resolution,
- *height* is the height of the image in pixels and is only useful with custom resolution,
- *useBackgroundColor* is only useful in *highQuality* mode and tells whether the image uses the same background as the 3D view or if it uses a defined color,
- *backgroundColor* contains a string coding the color to use when *useBackgroundColor* is set to *true* (HTML encoding in the form “#AARRGGBB”, where AA is the alpha channel - a value of *00* means transparent),
- *commandLine* is a string defining an optional post-process command line. If left empty, it means no post-process. When not empty, its content will be interpreted as a command line in the context of the user's computer. In particular, all paths used in this *commandLine* field must exist on the target computers, thus it is advised that the post-process tools are either available as global executables (in the %PATH% environment variable on Windows, for instance) or on a common network file share.

Note: to reference the exported file in the command line definition, one can use the *%s* notation. 3D Juump Infinite will automatically replace *%s* with the actual path to the exported file before it executes the command line.



The provided command-line will be executed as-is on the user's computer and in the user's context. It is your responsibility to make sure that the command-line is properly tested and does not alter the user's computer or any connected computers accessible in the user's context in any damaging way.

3.5.3 - Metadata export profile

```
{
  "version": 1,
```


```

    "name": <readable name of the profile>,
    "type": "metadata",
    "formatname": <id of the codec to use>,
    "commandline": <post-process command line>
  }

```

- *formatname* is one of the supported metadata output format:
 - "Format_Metadata_CSV" for international CSV format,
 - "Format_Metadata_CSVFR" for French CSV format,
 - "Format_Metadata_XML" for XML format,
 - "Format_Metadata_JSON" for JSON format.
- *commandline* is a string defining an optional post-process command line. If left empty, it means no post-process. When not empty, its content will be interpreted as a command line in the context of the user's computer. In particular, all paths used in this *commandline* field must exist on the target computers, thus it is advised that the post-process tools are either available as global executables (in the %PATH% environment variable on Windows, for instance) or on a common network file share.

Note: to reference the exported file in the command line definition, one can use the %s notation. 3D Juump Infinite will automatically replace %s with the actual path to the exported file before it executes the command line.

 The provided command-line will be executed as-is on the user's computer and in the user's context. It is your responsibility to make sure that the command-line is properly tested and does not alter the user's computer or any connected computers accessible in the user's context in any damaging way.

Note: the JSON format exports the product structure and metadata as JSON documents (see [Data documents](#)).

3.5.4 - Datapackage export profile

```


{
  "version": 1,
  "name": <readable name of the profile>,
  "type": "datapackage",
  "formatname": <id of the codec to use>,
  "geometry": {<geometry format>},
  "metadata": {<metadata format>},
  "exportBom": true,
  "commandline": <post-process command line>
}

```

- *formatname* is one of the supported datapackage output format:
 - "Format_Datapackage" is the only supported format.
- *geometry* is a valid geometry export profile as defined [above](#).
- *metadata* is a valid metadata export profile as defined [above](#).
- *exportBom* tells whether this datapackage export profile exports the metadata alongside the geometries.

- *commandLine* is a string defining an optional post-process command line. If left empty, it means no post-process. When not empty, its content will be interpreted as a command line in the context of the user's computer. In particular, all paths used in this *commandLine* field must exist on the target computers, thus it is advised that the post-process tools are either available as global executables (in the %PATH% environment variable on Windows, for instance) or on a common network file share.

Note: to reference the exported file in the command line definition, one can use the %s notation. 3D Juump Infinite will automatically replace %s with the actual path to the exported file before it executes the command line.


 The provided command-line will be executed as-is on the user's computer and in the user's context. It is your responsibility to make sure that the command-line is properly tested and does not alter the user's computer or any connected computers accessible in the user's context in any damaging way.

3.5.5 - Presentation export profile

```
{
  "version": 1,
  "name": <readable name of the profile>,
  "type": "presentation",
  "formatname": <id of the codec to use>,
  "template": <template to use>,
  "image": {<image format>},
  "commandline": <post-process command line>
}
```

- *formatname* is one of the supported presentation output format:
 - "Format_Presentation_JSON" for json format,
- *image* is a valid image export profile as defined [above](#),
- *commandLine* is a string defining an optional post-process command line. If left empty, it means no post-process. When not empty, its content will be interpreted as a command line in the context of the user's computer. In particular, all paths used in this *commandLine* field must exist on the target computers, thus it is advised that the post-process tools are either available as global executables (in the %PATH% environment variable on Windows, for instance) or on a common network file share.

Note: to reference the exported file in the command line definition, one can use the %s notation. 3D Juump Infinite will automatically replace %s with the actual path to the exported file before it executes the command line.

 The provided command-line will be executed as-is on the user's computer and in the user's context. It is your responsibility to make sure that the command-line is properly tested and does not alter the user's computer or any connected computers accessible in the user's context in any damaging way.

3.5.6 - Alternative profile declaration

It is also possible to declare profiles that are not project-related. This is done on the client side by deploying custom executables alongside 3D Juump Infinite client application.

The ∞Client installation includes a *postexports* folder containing custom profile declarator executables stored in subfolders. At startup, ∞Client calls each executable with the following parameters *-generatedefaultprofiles <filename>.json* where *<filename>.json* is a destination file where the executable should write an array of profile objects.

3.6 - Task-specific settings

The *tasksettings* field regroups all the task-specific settings.

3.6.1 - Presentation task settings

```
"PresentationTask": {
  "aspectratio": [
    16,
    9
  ]
}
```

For *PresentationTask*, the possible settings include the default aspect ratio of the presentation. It is defined as an array of two numbers (width vs. height) under the key *aspectratio*.

4 - Interoperability

3D Juump Infinite client application is interoperable via URL. Technically, the client application installer registers a system-wide custom handler for a specific URI scheme. All URL with the *infinite* scheme are then automatically redirected to the client application. Such URL only supports *localhost* or *127.0.0.1* host.

4.1 - Search URL

The *infinite://localhost/search?qql=...* URL lets other applications trigger a search in 3D Juump Infinite client application. It requires one query parameter:

- *qql*: this is the CQL search string to use (see the User Manual for a detailed explanation on CQL). Don't forget to %-encode this string.

4.2 - Select URL

The *infinite://localhost/select* URL allows to select part instances in the client application. It requires several query parameters:

- *type*: allows to chose the means of selection, either:
 - *metadata*: selects part instances according to their metadata thanks to a CQL filter,
 - *boxing*: selects part instances according to their 3D bounding box.
- *qql* (for *metadata* selection only): this is the CQL search string to use (see the User Manual for a detailed explanation on CQL). Don't forget to %-encode this string.

- *xmin* (for *boxing* selection only): lowest world coordinate of the box along the X axis.
- *xmax* (for *boxing* selection only): highest world coordinate of the box along the X axis.
- *ymin* (for *boxing* selection only): lowest world coordinate of the box along the Y axis.
- *ymax* (for *boxing* selection only): highest world coordinate of the box along the Y axis.
- *zmin* (for *boxing* selection only): lowest world coordinate of the box along the Z axis.
- *zmax* (for *boxing* selection only): highest world coordinate of the box along the Z axis.
- *goto* (optional, default *false*): whether the client application should also move the camera to the resulting selection.
- *ghost* (optional, default *false*): whether the client application should also ghost the rest of the DMU.

Annexes

1 - Range of use

This annex summarizes the range of use of the software.

1.1 - Minimum requirements

The ∞Client runs on Windows 7 (or higher) x64. Hardware accelerated 3D rendering requires an OpenGL 3.1 and GLSL 140 compatible graphics card.

For the application to run, the local computer must meet some minimal requirements:

- Windows 7/8/10 64-bit version
- 2GB RAM
- 4GB disk space for binaries and caches

1.2 - Supported input formats

3D Juump Infinite is able to process the following formats:

- ACIS (.sat) - all => R21
- ASC Medusa 3D (.asc)
- CADD5 (explicit parts) & CAMU (._pd, ._ps) - 4 & 5

- FBX (.fbx)
- I-DEAS (.arc,.unv) - all => NX5
- IGES (.igs) 5.2 & 5.3
- Inventor (.ipt,.iam) - all => 2017
- CATIA V4 (.model,.dlv,.exp,.session) - all 4.xx
- CATIA V5 (.CATPART,.cgr,.CATProduct) - R10 - R26
- CATIA V6 (.3Dxml) - 2011x => 2013x
- 3D Experience (.3Dxml) - 2014 => 2015x
- JT-Format (JtOpen) (.jt) 7.0 => 10.2
- Matra Euclid 3 (.e3i) - 3.2
- Nastran (.nastran)
- NX Unigraphics (.prt) - 11 => NX11
- OBJ (.obj, .mtl)
- Parasolid XT-Format (.x_t) - all => 28
- ProEngineer (.asc,.prt, .neu) - part files: 13 => Creo 4 (F000) / neutral files: 13 => WF5
- Rhino 3D (.3dm)
- ROBCAD (.rf)
- Solidworks (.sldprt,.sldasm) - 99 => 2017
- STEP AP203, AP214 and AP242 (.stp) - 203/214/242
- Straessle EUKLID (.edx)
- STL (.stl)
- VDA (.vda)
- VRML (.wrl, .wrz, .vrml) - 97
- 3DS (.3ds)

1.2.1 - Geometry

3D Juump Infinite only accepts surface information. All other geometric informations are ignored (in particular, vector and point data are not supported).

1.2.2 - Metadata

3D Juump Infinite is able to extract product structure, including external references. It also extracts textual, numeric and datum metadata, plus any combination of lists and maps of the above.

1.2.3 - Annotation

3D Juump Infinite is able to retrieve both FTA and PMI. It is limited in the number of options it supports regarding the FTA and PMI representation. Supported data includes:

- text content,
- fonts,
- colors,
- shapes (limited to square and flags),

Symbols (NOA) are not automatically read from input files and require a manual customization to circumvent the lack of universal symbol catalog.

All glyphs visible in a given Annotation View must fit in a 2048x2048 pixels image.

1.3 - Supported output formats

1.3.1 - Geometry

Supported output formats for geometry export are:

- OBJ
- VRML
- FBX
- JT
- STEP¹⁸+JT
- STEP+WRL
- WRL+WRL
- WRZ+WRZ

Exported geometries are tessellated. Annotations are not exported.

1.3.2 - Image

Supported output formats for image export are:

- PNG
- JPEG
- TIFF

Transparency support depends on the output format.

1.3.3 - Metadata

Supported output formats for metadata export are:

- CSV (comma separated)
- CSV (semi-colon separated)
- XML
- JSON

1.3.4 - Presentation

Supported output formats for presentation export are:

- Markdown
- HTML

¹⁸ STEP AP242 Part 21

- HTML with DZSlide embedded viewer
- ODP
- JSON

Note: Markdown and ODP do not support rich-text, thus when slide comments containing rich-text are found, they are exported as raw-text instead.

1.4 - Limits

Known limits include:

- Minimum number of assemblies¹⁹: 1
- Minimum number of single parts²⁰: 1
- Maximum number of parts²¹: 536 870 912
- Maximum number of links²²: 536 870 912
- Maximum number of ids²³: 4 294 967 294
- Maximum number of geometry instances²⁴: 16 777 215
- Maximum number of distinct materials: 32 768
- Maximum number of annotation views: 1 073 741 823
- Maximum number of annotation types: 1 024
- Maximum number of rasters: 4 096
- Maximum number of annotations visible at the same time: 262 143
- Maximum number of conf declared by the connector: 10 000

2 - Third-party software licenses

The details of licenses is available on the 3D Juump Infinite Third Party License manual provided with the software.

3 - Export control classification

The Software, which integrates dual use information security items of American origin (ECCN 5D992.c <10%), is subject to the US Export Administrative Regulations (EAR) 15 C.F.R. part 730 et seq. for the country Group E:1 and E:2 which are, at the date of the License Terms and

¹⁹ Structure document with children

²⁰ Structure document with geometric representation

²¹ PartMetadata documents

²² LinkMetadata documents

²³ Distinct *id* fields

²⁴ One source model instantiated at one world position

Conditions: Iran, North Korea, Sudan, Syria and Cuba. In particular, the User shall not use, export or re-export the Software in those countries and with end users or for end uses in breach of the US export control regulations.

The Software has been the subject of a declaration of operations relating to a means of cryptology to the ANSSI (Declaration N ° 17070363). However, the Software does not come under Regulation (EC) N ° 428/2009 of May 5, 2009, setting up a Community regime for the control of exports, transfer, brokering and transit of dual-use items, as confirmed by the Direction Générales des Entreprises / Services des Biens à Double-Usage Goods in his mail N ° FR 80404.