DRUMBEAT - REST interface

Abstract - The REST interface to DRUMBEAT platform provides a way to access information about several interrelated data sources - such as BIM models or event sources - that are published on a DRUMBEAT platform using the Web of Data representations. The inter-related models do not have to reside on the same server, and usually they do not. The REST interface is not a substitute to a general access to DRUMBEAT models using the SPARQL query language. Rather, REST interface provides a simple but more restricted access to building information. However, this kind of access is designed to be sufficient to practical use cases, such as sharing

Table of contents

1 Overview
2 Concepts
3 Principles
  3.1 Use of HTTP methods
  3.2 URI scheme for platform resources
  3.3 Data formats in HTTP requests and replies
4 Application interface
  4.1 Version
    4.1.1 Get the version information
  4.2 Collections
    4.2.1 Get all collections
    4.2.2 Create a collection
    4.2.3 Get the description of one collection
    4.2.4 Delete a collection
  4.3 Models
    4.3.1 Get the datasources belonging to a collection
    4.3.2 Create a new datasource to a collection
    4.3.3 Get the description of a datasource
    4.3.4 Delete a datasource
  4.4 Datasets
    4.4.1 Get the list of datasets belonging to a datasource
    4.4.2 Create a dataset to a datasource
    4.4.3 Get the description of one dataset
    4.4.4 Upload a file from the server
    4.4.5 Upload a file from an URL
    4.4.6 Upload a file from the client
    4.4.7 Upload content directly in the body of the call
    4.4.8 Get the contents of one dataset
  4.5 Objects
    4.5.1 Get the contents of an object
    4.5.2 Get the type of an object
  4.6 Properties
    4.6.1 Get the property value of an object
5 Server-to-server interface
1 Overview

DRUMBEAT platform can be used to publish building data on the Web so that users can access the data in a granular manner over the Web. It is based on two families of technologies: Web of Data and REST.

Both the Web of Data and REST are based on the idea of the primacy of resources. In both approaches the resources are identified with URIs. The main difference is that Web of Data is focused on the representation of data - a graph-based data model, schema/ontology languages, serialization formats, and query/reasoning languages - while REST is meant for the specification of service interfaces. These two families of technologies can be integrated in a natural manner:

- **REST**: A simple request-response interface to access building information.
- **Web of Data**: A semantic/structural representation for the information passed over the interface.

The data is created, managed, and published in a decentralized way as shown in Figure 1. The figure shows two different options for publication of BIM models. In Option 1 the Architect has an own Web server that has the DRUMBEAT platform middleware installed. Once a new version of an architectural model is created and exported into IFC, the IFC data is uploaded to the server. The server automatically converts the IFC model into a Web of Data representation, and generates a Web address (a URI) for each building element in the model.

![Figure 1 - Publishing building-related datasets on different DRUMBEAT servers](image_url)
In the Option 2, the Structural engineer and MEP engineer use a DRUMBEAT publication platform that is hosted by an external service provider. Again, once they produce a new version of a model, they upload that model in IFC format to the server, and the server takes care of the necessary conversions and URI generation.

Once the models are published in the Web, all building objects have global addresses that can be used to retrieve information about them. The addresses can also be used to establish links between elements in different models.

2 Concepts
The basic concepts of the DRUMBEAT platform are the following:

- **Host**: A server running the DRUMBEAT platform. A host has a domain name, and contains information about a set of sites.
- **Collection**: Information belonging together (such as to a same project, building, site, customer order, or a product category) can be grouped into a same collection.
- **Model**: Each Model represents one identified origin of published data: where the data comes from, what is its type, and how it relates to other Models. The data is produced possibly in a number of pieces that logically belong together. Examples are BIM models (Architectural/Structural/MEP), a stream of Status Events, and so on. Subclasses of a Model are BIM Model, Event Source, and Link Source.
- **Dataset**: The content of a Model, possibly generated from an external representation such as an IFC file. A Model can contain several Datasets. Each Dataset can describe a different version of a model, or a different subset of the model.
- **Object**: An entity such as a building component, cartesian point, or an approval. The object model is based on the RDF data model.
- **Property**: An attribute or relation of an object. Based on the RDF data model.

Figure 2 - Conceptual organization of data in a DRUMBEAT platform
Figure 2 shows an example of how the data in a DRUMBEAT host is organized. There can be many different Collections containing multiple Models, and the content of each Model is represented in a number of datasets. The information about one object may be represented in multiple datasets (containing for example different versions or parts of data), and has to be selected/combined from those in an appropriate way. The concepts presented in Figure 2 are metadata - data about data - and are defined in the LBDHO-ontology that is published in OWL and in JSON-LD. A figure about the details of LBDHO is shown in Figure 3.

![Figure 3 - LBDHO (Linked Building Data Host Ontology)](image)

It should be noted that there can be Collections, Models and DataSets related to the same project or site on multiple DRUMBEAT servers. Each of the servers would have its own representation for the site. The architectural model may reside in a different server than the structural model of the same site. Also some additional properties of objects may be defined in a dataset at another server from where the object is originally hosted.

Figure 4 shows the organization of the data in an RDF store from the implementation perspective. In general, a RDF store consists of a RDF dataset which has a set of RDF graphs: one default graph, and a number of named graphs. The default graph (the upper layer in Figure 4) is used in DRUMBEAT platform to store the metadata of actual content, and this metadata is based on the LBDHO. In the figure there is an example of a Collection site1, Model structural, and two Datasets v1 and columns. The DataSets indicate the named graph in which the content of the dataset is stored. Named graphs are shown at the lower layer in Figure 4.
3 Principles

3.1 URI scheme for resources

In the Linked Data approach the URI of an entity contains the domain name that identifies the server where the representation of the object is stored. If the URI of an object is typed into the address field of a Web browser, the request is sent to the domain specified in the URI to get the representation of the object. For this reason, it is not possible to mint URIs from another domain than the domain of the server itself because the requests for these objects would go to a domain that cannot provide the representation of the object.

The basic URI scheme for IFC objects is the following:

    <protocol>://<domain>/<interface>/<identifier>?<parameters>

As a whole the URI acts as a globally unique identifier. When used for the retrieval of information, the URI can be deconstructed to parts whose roles are as follows:

- <protocol> defines how the information can be retrieved over the Internet,
- <domain> specifies where the information is (the host or server),
- <interface> selects the particular interface, the kind of information, in the host,
- <identifier> is the specification of what information to access in the host, and
- <parameters> specifies additional information, for instance, to filter the returned result.

In DRUMBEAT platform the following more specific conventions is used:

- <interface> is the <resource type> that is manipulated (in plural); currently it is one of collections, Models, datasets, objects, properties.
• `<identifier>` is the `<resource path>` starting from a collection; it is a sequence of names/identifiers of resources.

Therefore, the more specific pattern is the following:

```
<protocol>://<domain>/<resource type>/<resource path>?<parameters>
```

Using these conventions, the following are examples of URIs for a collection, a model, a dataset and an object (for instance, a precast element):

- `https://example.com/collections/s1`
- `https://example.com/datasources/s1/struct`
- `https://example.com/datasets/s1/struct/ds1`

### 3.2 URLs and versions/views

The URIs are not specific to versions or views of a model. That is, the URI of a same object should stay the same in the next version of the model, or in the different views of the model.

The rationale is to make the external use of URIs more natural, and to prevent unnecessary broken links. A URI can be stored outside of the system and accessed later on, resulting with the up-to-date description of the object. Also links can be made to objects, without the fear that they will soon end up referring to an obsolete version of an object.

### 3.3 Data formats

When the result of an HTTP request contains structural data, this data will be represented in RDF serialized in either of the following formats:

- **JSON-LD** (JSON for Linking Data) - JSON-LD is based on JSON format that is enriched by a set of keywords to represent the `identity` (`@id`) and `type` (`@type`) of an object, and to indicate which `vocabularies` (`@context`) are used. It includes also some more advanced features. JSON-LD can be parsed and processed with libraries such as `jsonld.js` in JavaScript clients or `Jena` in Java applications. For more details, see [JSON-LD specification](https://www.w3.org/TR/2018/REC-json-ld-20180214/) of W3C.
- **Turtle** (Terse RDF Triple Language) is a simple human readable serialization of RDF data. It is particularly useful when the data is written by hand.

The vocabularies used (as specified in the `@context` keyword in JSON structure, can be found in Aalto DRUMBEAT server (http://drumbeat.cs.hut.fi/) for instance as follows:

- **LBDHO**
  - in OWL: `http://drumbeat.cs.hut.fi/owl/lbdho.ttl`

- **SEO** (Status Event Ontology)
  - in OWL: `http://drumbeat.cs.hut.fi/owl/seo.ttl`

The data that is uploaded to server can be multiple different representations (as long as that is supported by the platform). In addition to RDF (in Turtle and JSON-LD serializations), the supported
formats are IFC (in Step Physical File Format). In the future, CVS - and possibly various XML formats - can be supported.

3.4 HTTP methods

Different HTTP methods should be used to implement CRUD operations as follows:

- Create: POST or PUT
- Read: GET
- Update: PUT or PATCH
- Delete: DELETE

There has been significant discussion whether to use PUT or POST for creating objects. The most important criteria to make the choice are:

1. Will the URIs be decided by the client or the server?
2. Are the benefits of the idempotence of PUT important?

In many REST-based applications the server may have better information about the proper URIs. However, in Linked Data applications the actual URIs are important, and the clients may have a better understanding of them. For this reason, the PUT method appears to be a better choice for creation of objects. But this choice still needs to be validated in practical use cases.

3.5 HTTP headers

The REST calls can return results in different formats, as specified in the headers of http calls.

The supported formats in the DRUMBEAT REST API  Accept headers are:

- JSON-LD: "application/ld+json"
- Turtle: "text/turtle"
- HTML: "text/html"

For example, the following is an Accept header string that gives the preference to JSON-LD result:

- Accept: application/ld+json;q=1.0, text/turtle;q=0.6, text/html;q=0.2

3.6 HTTP errors codes

The following HTTP errors codes are used in the API:

Successful statuses:

- 200 OK - query succeeded
- 201 CREATED - creating resources succeeded
- 204 NO CONTENT - deleting resources succeeded

Client request errors:

- 404 NOT FOUND - resource not found
- 409 CONFLICT - resource already exists and cannot be created
- 403 FORBIDDEN - resource has children and cannot be removed

Server errors:

- 500 INTERNAL SERVER ERROR - unexpected server error
4 Application interface

4.1 Version

4.1.1 Get the version information

**GET /version**

Get the version information of the DRUMBEAT platform.

Example:

```
GET http://construct.com/version

->
{
   "@context": {
      "version": { "@id": "http://purl.org/dc/terms/version", "@type": "@id" },
      "xsd": "http://www.w3.org/2001/XMLSchema#",
      "dcterms": "http://purl.org/dc/terms/"
   },
   "@graph": [
      { "@id": "apps/", "dcterms:version": "0.8.0-SNAPSHOT" },
      { "@id": "apps/ifc2ld", "dcterms:version": "0.9.1-SNAPSHOT" },
      { "@id": "apps/jena", "dcterms:version": "0.7.3-SNAPSHOT" },
      { "@id": "apps/rest-api", "dcterms:version": "0.8.0-SNAPSHOT" } ]
}
```

4.2 Collections

4.2.1 Get all collections

**GET /collections**

Get the URIs of all collections maintained in a host.

Example:

```
GET http://construct.com/collections

->
{
   "@context": { "@vocab": "http://drumbeat.cs.hut.fi/owl/lbdho.owl#"},
   "@graph": [
      { "@id": "http://construct.com/collections/c1", "@type": "Collection" },
      { "@id": "http://construct.com/collections/site2", "@type": "Collection" },
      { "@id": "http://construct.com/collections/z345", "@type": "Collection" }
   ]
}
```

4.2.2 Create a collection
PUT /collections/<collectionname>

→ URI

Create a new collection with <collectionname>. The creation is done with the specific name given by the client, and it is an error if that name is already in use.

Example:

PUT
http://construct.com/collections/c1

→

{ "@context" : { "@vocab" : "http://drumbeat.cs.hut.fi/owl/lbdho.owl"},
  "@id": "http://construct.com/collections/c1",
  "@type": "Collection" }
4.3 Models

4.3.1 Get the datasources belonging to a collection

**GET /datasources/<collectionname>**

Get the list of all datasources belonging to a collection.

Example:

```json
http://construct.com/datasources/s1
-->
{ "@context" : { "@vocab" : "http://drumbeat.cs.hut.fi/owl/lbdho.owl#" },
  "@graph": [
    { "@id": "http://construct.com/datasources/s1/structural", "@type": "Model" },
    { "@id": "http://construct.com/datasources/s1/status", "@type": "Model" },
    { "@id": "http://construct.com/datasources/s1/elements", "@type": "Model" }
  ]
}
```

4.3.2 Create a new datasource to a collection

**PUT /datasources/<collectionname>/<datasourcename>**

{...}

Create a new datasources to a collection. The name of the datasource is given in the command.

Example:

```json
PUT http://construct.com/datasources/s1/structural
{ "@context": "http://drumbeat.cs.hut.fi/jsonld/lbdho.jsonld",
  "@type": "BimModel",
  "discipline": "Structural",
  "vocabulary": "http://drumbeat.cs.hut.fi/owl/IFC4_Standard.ttl"
} -->
http://construct.com/datasources/s1/structural
```

4.3.3 Get the description of a datasource

**GET /datasources/<collectionname>/<datasourcename>**
4.3.4 Delete a datasource

DELETE /datasources/<collectionname>/<datasourcename>

Delete a datasource, together with all its datasets.

Example:

DELETE http://construct.com/datasources/s1/structural

4.4 Datasets

4.4.1 Get the list of datasets belonging to a datasource

GET /datasets/<collectionname>/<datasourcename>

Get the list of all datasets that belong to a datasource.

Example:

http://construct.com/datasets/s1/structural

-->
{ "@context" : { "@vocab" : "http://drumbeat.cs.hut.fi/owl/lbdho.owl#"},
 "@graph": [ 
 { "@id": "http://construct.com/datasets/s1/structural/v1", "@type": "DataSet" },
 { "@id": "http://construct.com/datasets/s1/structural/v2", "@type": "DataSet" },
 { "@id": "http://construct.com/datasets/s1/structural/walls", "@type": "DataSet" }
 ]
}
### 4.4.2 Create a dataset to a datasource

```json
PUT /datasets/<collectionname>/<datasourcename>/<datasetname>
{
    "@type": <datasetclass>,
    ...
}
```

Create a description of a dataset. Creates a new named graph for the contents.

**Example:**

```json
PUT http://construct.com/datasets/s1/structural/v1
{
    "@type": "StaticDataset",
    "creationTime": "2015-11-18-15:30",
    "origin": { "a": "File", "address": "file:///Users/aaa/IFC/s1-structural-v1.ifc" },
    "dataformat": "IFC-SPFv"
}
```

→
http://construct.com/datasets/s1/structural/v1

### 4.4.3 Get the description of one dataset

```json
GET /datasets/<collectionname>/<datasourcename>/<datasetname>
```

Get the description of one dataset.

**Example:**

```json
http://construct.com/datasets/s1/structural/v1
```

→
```json
{
    "@type": "StaticDataset",
    "creationTime": "2015-11-18-15:30",
    "origin": "file:///Users/aaa/IFC/s1-structural-v1.ifc",
    "dataformat": "IFC-SPFv"
}
```

### 4.4.4 Upload a file from the server

```json
POST datasets/{collectionId}/{dataSourceId}/{dataSetId}/uploadServerFile
```

**Form content type:** application/x-www-form-urlencoded

**Form parameters:** filePath, dataType, dataFormat, compressionFormat
4.4.5 Upload a file from an URL

```plaintext
POST datasets/{collectionId}/{dataSourceId}/{dataSetId}/uploadUrl
Form content type: application/x-www-form-urlencoded
Form parameters: url, dataType, dataFormat, compressionFormat
```

4.4.6 Upload a file from the client

```plaintext
POST datasets/{collectionId}/{dataSourceId}/{dataSetId}/uploadClientFile
Form content type: multipart/form-data
Form parameters: file, dataType, dataFormat, compressionFormat
```

4.4.7 Upload content directly in the body of the call

```plaintext
POST datasets/{collectionId}/{dataSourceId}/{dataSetId}/uploadContent
Form content type: multipart/form-data
Form parameters: content, dataType, dataFormat, compressionFormat
```

Insert data into a dataset. The data will be posted in the body of the http request (whose maximum size can be set in the server configurations). If data is originally in a binary format it must be base64 encoded/decoded. Before the insertion the data is converted in the server to the target format.

The content should be inserted as one batch to static datasets, while it can be inserted in smaller pieces to a dynamic dataset.

The meanings and allowed values of form parameters are:

<table>
<thead>
<tr>
<th>Form param</th>
<th>Description</th>
<th>Allowed values</th>
</tr>
</thead>
<tbody>
<tr>
<td>filePath</td>
<td>path to a server file</td>
<td>Example: c:\test\sample.ifc</td>
</tr>
<tr>
<td>file</td>
<td>client file</td>
<td></td>
</tr>
<tr>
<td>url</td>
<td>url to download file from</td>
<td>Example: <a href="http://drumbeat">http://drumbeat</a>.</td>
</tr>
<tr>
<td>content</td>
<td>binary data</td>
<td></td>
</tr>
<tr>
<td>dataType</td>
<td>data type</td>
<td>IFC, RDF</td>
</tr>
<tr>
<td>dataFormat</td>
<td>RDF data format</td>
<td>TURTLE, NTRIPLES, NQUADS, JSONLD, RDFXML, or other field names specified in class org.apache.jena.riot.Lang</td>
</tr>
</tbody>
</table>
4.4.8 Get the contents of one dataset

GET /datasets/<collection>/<datasource>/<dataset>/content?format=<format>

Get the serialization of a whole dataset. This call should be used only in special cases (e.g., debugging, backups) since the size of the serialization can be prohibitively large.

Example:

http://construct.com/datasets/s1/structural/v1/content?format=rdf

{ ... data ... }

4.5 Objects

4.5.1 Get the contents of an object

GET /objects/<collection>/<datasource>/<guid>

Get the contents of an object, including its links.

Example:


  a ifc:ifcWall
  ifc:globalId "B445F4F2-4D02-4B2A-B612-5E456BEF927"^^xsd:string ;
  ifc:name [ a ifc:IfcLabel ; rdf:value "Wall 23"^^xsd:string ] ;
  ifc:objectPlacement [... ] ;
  ifc:representation [ a ifc:IfcProductDefinitionShape ; ... ] ;
4.5.2 Get the type of an object

**GET /objects/<collection>/<datasource>/<guid>/type**

Get the type of an object.

Example:

```
```

> "ifc:ifcWall"

4.6 Properties

4.6.1 Get the property value of an object

**GET /property/<collection>/<datasource>/<guid>/<propertyname>**

Get the value of the property `<propertyname>` of an object.

Example:

```
```

> "89.00"

5 Server-to-server interface

5.1 Notifications

When a link to an external object is created, deleted, or modified, this target object is notified about the event. The target object can decide what will be done with the notification. It can, for instance, create a backlink to the source object. The notification is implemented as a REST call to a URI derived from the URI of the referred object.

5.1.1 Notify about a created link
PUT /objects/<collection>/<datasource>/<guid>/linkCreated
{  "originalObject: <uri>,
    "predicate": <predicate>}

An method to call when a link to a remote URI is created. The call can result in a creation of a backlink, other internal notifications, or other such things as appropriate for the functionality of the application.

Example:

{"@context": { "@vocab": "http://drumbeat.cs.hut.fi/owl/seo.ttl#"},
 "id@": "http://precast.com/objects/s1/status/C233FBB2-3A3B-EFF4-C123-DE22ABC8488",
} -->
results in the creation of a backlink, internal notifications, or other such things as appropriate for the functionality of the application

5.1.2 Notify about a deleted link

PUT /objects/<collection>/<datasource>/<guid>/linkDeleted
{  "originalObject: <uri>,
    "predicate": <predicate>}

An method to call when a link to a remote object is deleted. The call can result in a deletion of a backlink, other internal notifications, or other such things as appropriate for the functionality of the application.

Example:

{"@context": { "@vocab": "http://drumbeat.cs.hut.fi/owl/seo.ttl#"},
 "id@": "http://precast.com/objects/s1/statusevents/C233FBB2-3A3B-EFF4-C123-DE22ABC8488",
} -->
results in the deletion of a backlink, internal notifications, or other such things as appropriate for the functionality of the application

5.2 Backlinking

Whenever an RDF statement (a triple of the form <subject, predicate, object>) is created, deleted, or modified, it is checked whether the subject or object is a remote entity, that is, not hosted in the local domain. For instance, in a domain http://construct.com the entity http://precast.com/objects/n22/mep/B445F4F2-4D02-4B2A-B612-5E456BEF927 is a remote one, as it is hosted in the domain http://precast.com.
5.2.1 General policy

In the Web architecture a remote entity (or a remote domain in general) is not aware of the links created to it from elsewhere. This awareness could, in principle, be created using semantic search engines, but in DRUMBEAT platform it is managed with notifications in order to make the behavior of the system more predictable and accurate.

In the DRUMBEAT platform the following policy is obeyed:

1. **Whenever an RDF statement is created that mentions a remote resource, this remote resource is notified about the created RDF statement.** This allows the remote domain to become aware of referring entities and to store a backlink to them.

2. **When an object receives a notification of a remote reference, it stores a backlink to the referrer.** This allows the domain to continue to be aware of the reference in the future.

3. **The notification is propagated to all entities that have been declared sameAs with the notified entity.** If there are several identifiers - local and remote ones - for the same entity, or if identifiers have changed over the time, this can be managed with sameAs links between entities.

4. **When a notification is processed, other processes may be started.** These processes can be specific to particular use cases or applications. They can include caching of information from the other domain, reasoning or derivation of additional properties based on the linked entity, detecting deviations, sending of notifications to people, and so on.

5.2.2 SameAs links

SameAs links enable the use of different identifiers for a same entity. An identifier A is declared sameAs with identifier B (in the domain of A). Since the linksets are maintained locally, this means that also B needs to be declared sameAs A (in the domain of B). Furthermore, if A and B had previous sameAs links with other objects, all other sameAs links need to be created so that each object in the equivalence class is directly sameAs with every other. (The assumption is that the equivalence classes are not large, but consist at most 4-5 nodes).

The sameAs links need to be managed as follows:

\[
X.create(X \ sameAs \ Y) \quad \# \ domain \ of \ X \ is \ about \ to \ create \ a \ link \ X \ sameAs \ Y
\]

- X checks whether it already has a link to \(Y\)
- unless such link already exists:
  - X stores the link: \(X.store(X \ sameAs \ Y)\)
  - X notifies Y about
    - the sameAs link from X to Y: \(Y.notify(X \ sameAs \ Y)\)
    - every other sameAs links from X to some Z: \(Y.notify(Z \ sameAs \ Y)\)

\[
X.notify(Y \ sameAs \ X) \quad \# \ domain \ of \ X \ receives \ a \ notification \ Y \ sameAs \ X:
\]

- X creates a link: \(X.create(X \ sameAs \ Y)\)

An example consisting of four nodes, A, B, C and D, all potentially (but not necessarily) residing in different hosts. Initially A sameAs B and C sameAs D. Consider now what happens when a new sameAs link is created between B and C. Figure below shown some of the initial steps in the sameAs link update process, and the final state after all notifications.
After the initial stage the link B sameAs C is stored at B, followed by two notifications from B to C: first about the new link B sameAs C and second about another link that should be there, which is A sameAs C. On receiving the first notification C stores a link C sameAs B. It also sends a notification back to B based on a previous link, that D sameAs B. B will store a link to D and so on. Finally all links between nodes have been established. By default we do not record the reflexive links (such as A sameAs A).

5.2.3 Other link types
There are the following differences in the handling of other link types, when compared with the method presented for sameAs links:
1. Unless the relationship is symmetric, the backlink must be the inverse relation of the original link.
2. Unless the relationship is transitive, the other linked nodes need not be considered.

5.3 Caching