# EUDAT CDI HTTP REST API

Carl Johan Håkansson

Rome 2016-02-23

# REST API design principles:

- Reflect the EUDAT data model
  - Transparent design for EUDAT developers, sysadmins and decision makers
  - Transparent design for user side application programmers (it's an API!)
- Aspire towards 100% REST
  - "REST" is not only a buzzword - Representational State Transfer
  - Scalability is good
  - REST is for scalability
  - The Internet is REST and seems to scale

# ARCHITECTURE

Two alternative API architectures have been discussed for EUDAT:

1. A common API standard to be developed and deployed for each service instance.
2. A single API access point to be used and interconnected with all service instances within the EUDAT CDI, and potentially other resources as well. The API can itself be used as a service, providing access to various storage and data management resources.

EUDAT API development will first focus on the first alternative and develop APIs for all services. The second approach can become a continuation project, building an API as a single access point for all the EUDAT services.

# ARCHITECTURE

- Provide a common API access point for a group of services (one or more)
- Deploy with web server, e.g. Apache
- Connected with a group of EUDAT services e.g. at a certain site
- Supports EUDAT services via their native libraries or APIs
- Can be extended to support other resources over time

# ARCHITECTURE

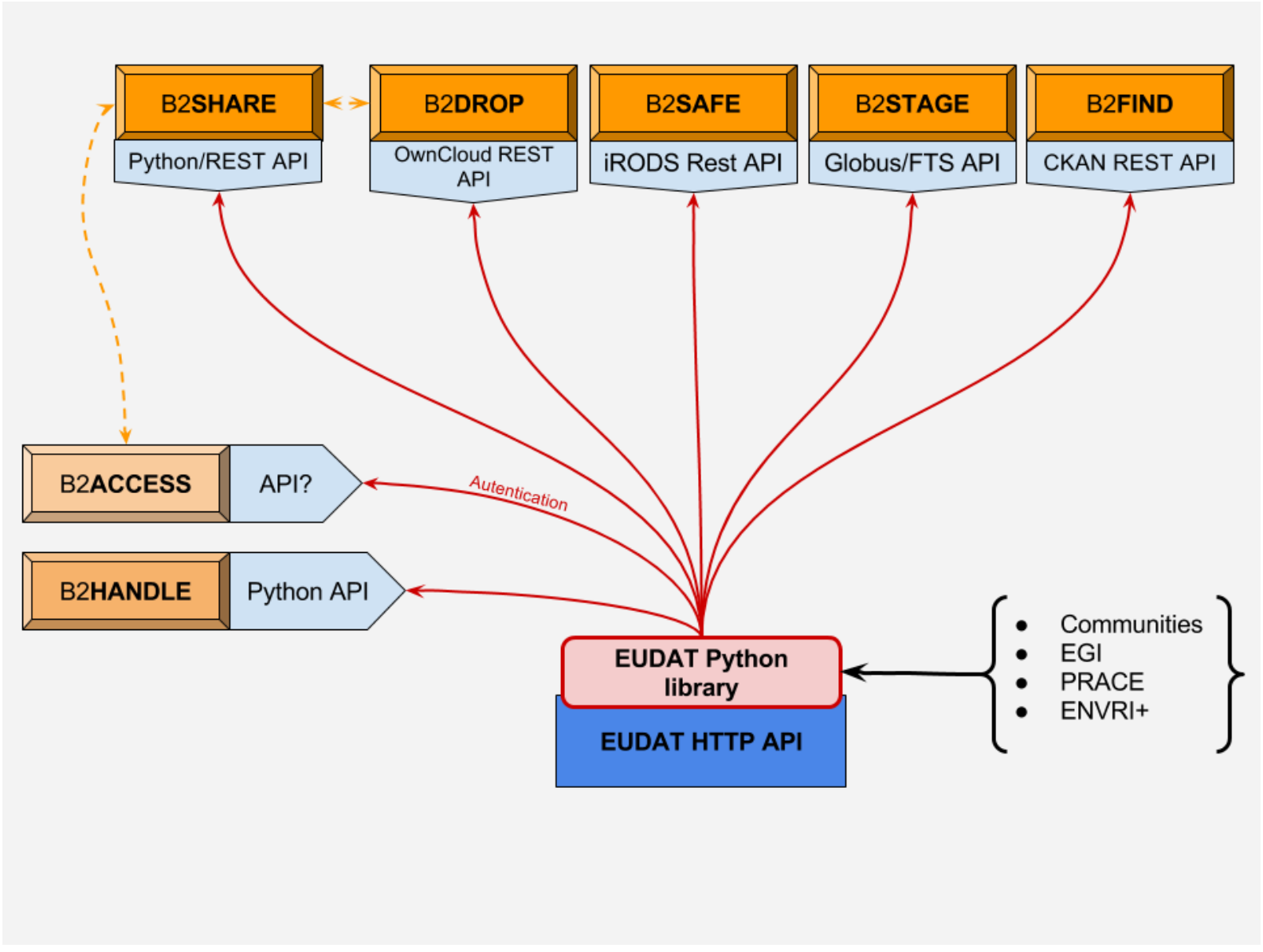


Figure 1:API Architecture

# IMPLEMENTATION

- The API is implemented in Python 3 using the Flask framework
- Flask can be used with most web-servers via the WSGI-standard.
- The API interconnects with EUDAT services' native APIs or libraries
- For iRODS, initially implemented as wrappers for icommands
- Direct communication via iRODS protocol to be investigated
- If the API is itself provided as a service, other resources can be accessed via the API, for example the Agave API

# AUTHENTICATION

- Oauth2 via B2ACCESS
- x.509 certificates issued by B2ACCESS for iRODS
- Tokens are transmitted in the HTTP header
- Authentication is required for every request.

# AUTHENTICATION

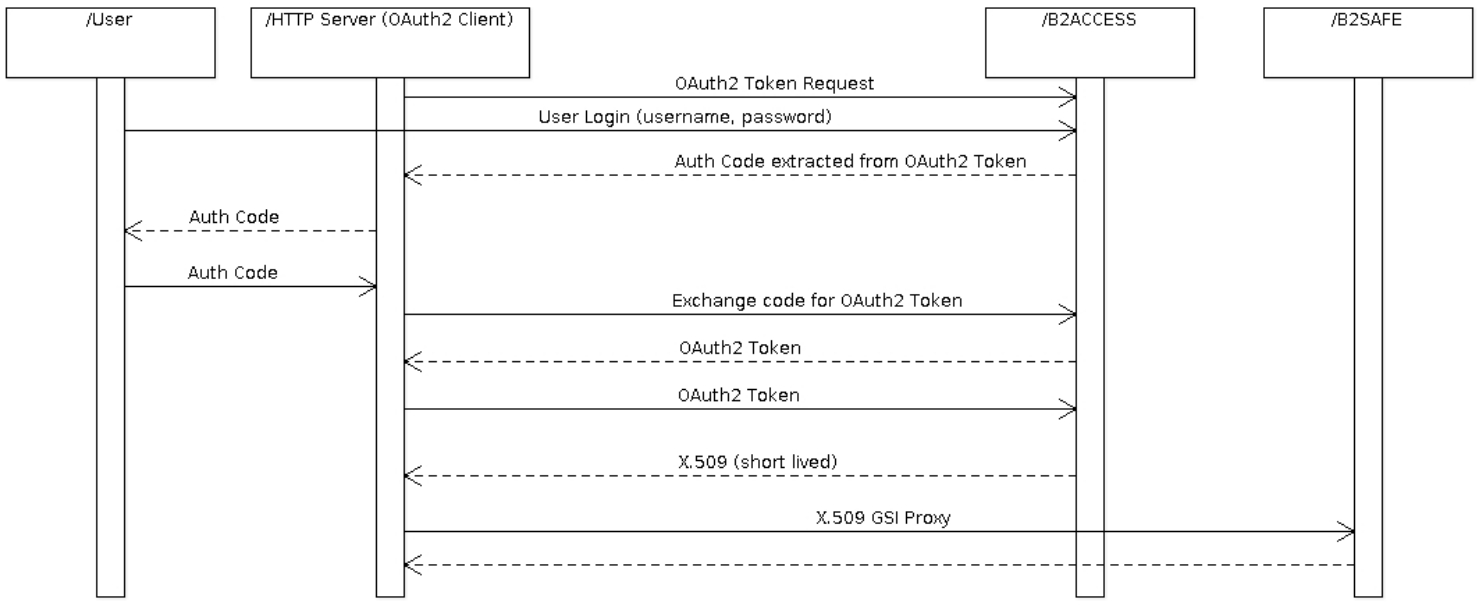


Figure 2:Authentication sequence for B2SAFE

# DATA MODEL

- A Digital Entity is anything that can be represented by (or encoded as) a sequence of bits (called a “bit-sequence” or “bitstream”)
- Digital Objects:
  - Has some content, which can be either one or more Digital Entities and/or other Digital Objects
  - Has a persistent identifier (PID)
  - Has metadata describing its content

# DATA MODEL

- There are three types of Digital Objects:
  - Data objects - represents physical files which are uploaded, described (has metadata) and registered (have been assigned a persistent identifier) in the EUDAT CDI
  - Metadata objects - Digital Objects describing other Digital Objects
  - Collections - Digital Objects containing other Digital Objects
- Some properties of Digital Objects:
  - A Digital Object can exists in exact one of four states: draft, committed, published or deleted.
  - Only Digital Objects in a draft state can be empty or lack PIDs. This is strictly by definition not a Digital Object. However, the draft state can be a transition state during Digital Object creation.
  - A committed or published Digital Object is always described by metadata and has a persistent identifier.
- The full state schema is mainly useful for publishing, but digital objects may also change state during creation.

# DATA MODEL

- Some system properties:
  - Data integrity has to be verified on the application (service) level. Thus e.g. transmission errors and data degradation need to be detected and handled adequately by the API. Only Digital Objects in a draft state can be allowed to contain corrupted data.
  - Digital Entities belonging to Digital Objects in a draft state may be deleted by the system (garbage collection).
  - A Digital Entity has a checksum [Is this required for Objects and collections?]

# DATA MODEL - PID oddities

In the current (in progress) EUDAT Data Model, there are some open questions about PID resolution:

- In some cases (unclear which) a PID of a Digital Object should resolve to the Data Entity of the Digital Object by default, while the Metadata Entity is accessible via a PID fragment - the Digital Object has no PID of its own
- Digital Objects which are collections should have a PID of its own, but it is under investigation what PIDs in collections should resolve to:
  - What should a collection PID resolve to?
  - What should a PID for the data content of a collection resolve to?
  - "Collections" can also be constructed from "packages", this should also be taken into account but has not been discussed in the API group

# API description - Design Principles

- 100% REST for best possible scalbility
- Corollary: the REST principle “Hypermedia As The Engine Of Application State” (HATEOAS) is always followed
- Full support for HTTP 1.1 (rfc2616)
- Full support for the PATCH method (rfc5879)
- Full support for HTTP Range Requests (rfc7233)
- Full support for HTTP 2 (rfc7540)
- Transfers are resumable (using HTTP Range Requests)

# API description - Design Principles

- Object hierarchies are supported and can be traversed in either direction
- Network communication supports parallell streams and multiplexing (HTTP 2)
- "No version-versioning": no explicit version unless absolutely necessary
  - No versioning in URIs
  - Request/Response versioning *can* be communicated in the http-header if absolutely necessary. This should only be a last resort.
  - Development should guarantee backward compatibility by never changeing or deleting functionality, only adding
  - New resources, responses or requests need no version number

# Naming conventions

- All names in URIs and parameters should be in lower case
- All general names should be in plural
  - example: /api/users

# Response codes

The API always respondes to HTTP requests with standard HTTP status codes as defined in IETF RFC 2616.

# Basic API

The basic API should be supported by all services.

# Digital Objects - Search/list

- URI path: /api/digitalobjects/
- HTTP method: GET
- Optional parameters: [ search parameters ]
- Returns: list of Digital Object IDs
- Return format:JSON object, {[“id”: “<ID>”, . . . ] }

# Retrieve a specific Digital Object

- URI path: /api/digitalobjects/<ID>
- HTTP method: GET
- Returns: information about the Digital Object
- Return format: JSON object, {“id”: “<ID>”, “published_by”: “<USER_ID>”, “files_count”: “42”, “metadata”: [ { “schema”: “<SCHEMA_ID>”, “fields”: {} } ] }

# Create a Digital Object

- URI path: /api/digitalobjects
- HTTP method: POST
- Request message: object metadata in JSON format
- Returns: id for the created Digital Object
- Return format: JSON object, { “id”: “<ID>” }

# Change state of a Digital Object

- URI path: /api/digitalobjects/<ID>/
- HTTP method: PATCH (or POST - cf rfc7231)
- Returns: status message

# Digital Entities (Files) - Search

- Search for an entity or a set of entities in a Digital Object
- URI path: /api/digitalobjects/<ID>/entities
- HTTP method: GET
- Optional parameters:
  - parent_id: id of a Digital Object to search
  - recursive: list all the entities in the Digital Object and child Digital Objects recursively.
  - name: file name
- Returns: entity ID
- Return format: JSON object, { “id”: “<FILE_ID>” }

# Digital Entities (Files) - Retrieve a file

- URI path: /api/digitalobjects/<ID>/entities/<ID>
- HTTP method: GET
- Optional parameters:
- Returns: a file
- Return format: physical file

# Digital Entities (Files) - Delete a file

- URI path: /api/digitalobjects/<ID>/entities/<ID>
- HTTP method: DELETE
- Returns: HTTP 1.1 status message

Deletion is only possible if the Digital Object is in a draft state.

# Change file name

- URI path: /api/digitalobjects/<ID>/entities/<ID>
- HTTP method: PATCH
- Returns: status message

# Add a Digital Entity (a file) to a Digital Object

- URI path: /api/digitalobjects/<ID>/entities/
- HTTP Method: POST
- Required input data: the file, sent as multipart/form-data
- Returns: status message with name and size of the uploaded file

# Extended API

The extended API is supporting some service specific funtions.

# List all Digital Objects for a community

- URI path: /api/communities/<ID>/digitalobjects
- HTTP method: GET
- Optional parameters: *from*, *size* - retrieve *size* number of items from starting point *from*
- Returns: list of objects belonging to the community
- Return format: JSON array of objects, [{ ... }, { ... }]

Each searching/listing should indicate if there are any further objects by returning the total number of matching objects in the response header.

Example:

curl -H "Authorization: Bearer mF_9.B5f-4.1JqM" -i http://example.org/api/communities/EUDAT?from=10&size=50

# Communities - Search and retrieve

- URI path: /api/communities
- HTTP method: GET
- Returns: list of communities with community information
- Return format: JSON array of objects

# Create a community

- URI path: /api/communities
- HTTP method: POST
- Returns: list of communities with community information
- Return format: JSON object, { “id”: “<COMMUNITY_ID>” }

# Retrieve information for a specific community

- URI path: /api/communities/<ID>
- HTTP method: GET
- Returns: community information
- Return format: JSON object, { “id”: “<COMMUNITY_ID>”, “name”: “...”, “desc”: “...”, “schemas”: [ <SCHEMA_ID> ] }

# Modify information for a specific community

- URI path: /api/communities/<ID>
- HTTP method: PATCH
- Returns: updated community information
- Return format: JSON object, { “id”: “<COMMUNITY_ID>”, “name”: “...”, “desc”: “...”, “schemas”: [ <SCHEMA_ID> ] }

# Schemas

A community has a list of metadata schemas.

# Search/list community schemas

- URI path: /api/schemas
- HTTP method: GET
- Returns: list of schemas
- Return format: JSON object

# Create a new community schema

- URI path: /api/schemas
- HTTP method: POST
- Returns: new schema
- Return format: JSON object

# Retrieve a specific schema

- URI path: /api/schemas/<ID>
- HTTP method: GET
- Returns: schema information
- Return format: JSON object

# Deprecate and (optionally ) replace a schema

- URI path: /api/schemas/<ID>/deprecate?new_version=<ID>
- HTTP method: POST
- Optional parameter: new_version=<ID>
- Returns: schema information
- Return format: JSON object, { “id”: “<SCHEMA_ID>”, “managed_by”: “<ROLE>”, “is_deprecated”: “true/false”, “new_version”: “<SCHEMA_ID>”, “fields”: [ { name: “...”, “type”: “...”, “desc”: “...”, “required”: “true/false”, “cardinality”: “34” , “controlled_vocabulary”: [list of allowed values] } ] }

A schema cannot be modified. Only a new schema with a new ID can be created. A schema can be deprecated. Then it cannot be used to create new metadata objects. When it is deprecated, a new

# Retrieve a file from a service that does not support PIDs (B2DROP)

- URI path: /api/files/<ID>
- HTTP method: GET
- Optional parameters:
- Returns: file handle
- Return format: file handle

The preferred method is retrieval of files via their PID handles.

# Delete a file from a service that does not support PIDs (B2DROP)

- URI path: /api/files/<ID>
- HTTP method: DELETE
- Returns: status message

# Upload a new file to a service that does not support PIDs (B2DROP)

- URI path: /api/files/
- HTTP Method: POST
- Required input data: the file, sent as multipart/form-data
- Returns: status message with name and size of the uploaded file

# User management

- URI path: /api/users
- HTTP method: GET
- Optional parameters: -
- Returns: list of users profile information
- Return format: JSON list of objects, [ { “id”: <USER_ID>, “name”: “...”, “email”: “...” }, ... ]
- Each searching/listing should indicate if there are any further search results by returning the total number of matching items.

# User management

- URI path: /api/users/<ID>
- HTTP method: GET
- Required parameters: access_token
- Optional parameters: -
- Returns: users profile information for user <ID>
- Return format: JSON object, { “id”: <USER_ID>, “name”: “...”, “email”: “...” }

User accounts can only be created via B2ACCESS, the REST API does not support creation.

# JSON Schemas

- Return formats: JSON objects or lists of objects
- JSON schemas not defined - there are some candidate standards
  - http://json-schema.org
  - http://jsonapi.org
  - HAL
  - SIREN
- Perhaps an EUDAT JSON standard is best, a case for the DTR?
- The JSON objects/lists represents the Digital Objects, Digital Entities, etc
- 100% REST is 100% Representational State Transfer:
  - links to previous and next state is always required
  - Always link to parent Digital Objects
  - Always link to child Digital Objects/Entities
  - Use direct links, not PIDs - we want to link to the representation, not to the bitstream

# Example: HAL

```
{
    "_links": {
      "self": { "href": "/orders" },
      "next": { "href": "/orders?page=2" },
      "find": { "href": "/orders{?id}", "templated": true
    },
    "_embedded": {
      "orders": [{
          "_links": {
            "self": { "href": "/orders/123" },
            "basket": { "href": "/baskets/98712" },
            "customer": { "href": "/customers/7809" }
          },
          "total": 30.00,
          "currency": "USD",
          "status": "shipped",
        },{
          "_links": {
```

# Example: Siren

```
{
  "class": "orders",
  "properties": {
    "currentlyProcessing": 14,
    "shippedToday": 20
  },
  "entities": [
    {
      "class": "order list-item",
      "rel": "order",
      "properties": {
        "total": 30.00,
        "currency": "USD",
        "status": "shipped"
      },
      "entities": [
        { "rel": "basket", "href": "/baskets/98712" },
        { "rel": "customer", "href": "/customers/7809" }
```

# Next step: specification in swagger?

```
# Basic API spec in YAML for Swagger
swagger: '2.0'
info:
  title: EUDAT CDI Basic HTTP API
  description: API for the most basic functionality of the
  version: "0.7"
# will be prefixed to all paths
basePath: /api
produces:
  - application/json
paths:
  /digitalobjects:
    get:
        summary: Digital Objects listing
        description: |
            Search and list Digital Objects filtered by som
            parameters
        parameters:
```