Loading GSRS JSON within the REST API

28 April 2022

# Introduction:

GSRS uses a REST API architecture where communication with the database involves exchange of records in JSON format. This document is a practical example-based guide to some of the basic operations and procedures used to create, load, validate and update GSRS JSON substance records. While the focus of this guide is on GSRS substances, much of it is applicable to other GSRS entity services as well (including vocabularies, products, and clinical trials).

# Section 1: Creating a GSRS Substance JSON Record

GSRS substances records (as well as other entity records) use a specific JSON schema model for creating, retrieving, and updating data to the database via the REST API. The supplied JSON is converted to Java objects and then persisted in the underlying database using Object Relational Mapping (ORM). When retrieving data from the REST API, the java objects are fetched from the underlying database tables and serialized to JSON format in the API HTTP responses.

## Section 1.1: Reference Resources

When submitting a new GSRS substance record to a GSRS instance using the REST API, you must first build a GSRS JSON object in compliance with the expected model. The details of this model can be found in the following resources:

**GSRS Substance JSON Schema:**

<https://gsrs.ncats.nih.gov/downloads/GSRS_schema_2_0_0.json>

**GSRS Data Dictionary:**

<https://gsrs.ncats.nih.gov/downloads/GSRS_data_dictionary_2021_12_08.xlsx>

**GSRS API Interactive Documentation (JSON Examples):**

https://gsrs.ncats.nih.gov/#/api

**GSRS Substance Browse UI (UI examples, JSON can be downloaded):**

<https://gsrs.ncats.nih.gov/ginas/app/beta/browse-substance>

The **JSON schema** describes the expected fields, data types and composition of a GSRS substance JSON record in the JSON Schema format. This schema is useful for computers and automation tools, but less useful for humans who want to understand the purpose of the fields. The **GSRS Data Dictionary** contains information about those same fields in a flat tabular format, along with a description of each field’s purpose and some other properties that may be helpful when using the REST API. The dictionary includes: descriptive columns for controlled vocabulary resources, searching terminology conventions, and some plain-language usage notes and descriptions for how the fields work.

The **GSRS API Interactive Documentation** page gives some query and retrieval example calls which can be used to obtain example GSRS Substance JSON for specific records. The **GSRS Substance Browse UI** link to an actual instance of a GSRS website can also be used to explore real data with a user interface. Check for buttons and UI features that allow you to download JSON for specific substance records.

## Section 1.2: Simple JSON Example

Below, we present a basic example JSON for a “concept” substance record:

|  |
| --- |
| {  "substanceClass": "concept",  "names": [  {  "name": "DRINKING WATER EXAMPLE",  "references": [  "e7dcc059-7f47-4815-8444-2157381b8f17"  ]  }  ],  "references": [  {  "uuid": "e7dcc059-7f47-4815-8444-2157381b8f17",  "citation": "Some Citatation for Drinking Water Example",  "docType": "WEBSITE",  "publicDomain": true  }  ],  "access": [  "protected"  ]  } |

This record has a single name “DRINKING WATER EXAMPLE”, as well as a reference (data source) for where that name was found. This record is also flagged as “protected”, signaling it’s not intended to be exported in a “public” data exchange operation. Many more fields are available in GSRS, but the fields above are sufficient to register a basic concept record.

Of special note is how the “references” are handled in the JSON. Each reference used in the JSON must be added to the root-level “references” section as objects. These reference objects must have a “docType” (type of reference), a “citation” (free-text description) and a “uuid” (“Universally unique identifier”). The “uuid” should be generated by the client composing the JSON. There are many free and open-source tools to generate uuids as needed. When a reference object is added to the “references” section, it can then be linked to specific data elements using the uuid. In the above example, the name object explicitly states that it uses this “e7dcc059-7f47-4815-8444-2157381b8f17” reference as its source. Depending on how a local instance of GSRS is configured, many data elements will require at least one linked reference to pass validation.

While there are few *core* fields needed for GSRS JSON, there are many more fields available. After validation and submission of this JSON record above, the GSRS system will auto-generate some other important metadata, uuids, audit information and operational fields with default values.

## Section 1.3: Chemical and Protein JSON Examples

The example above is for a substance “concept” record. For GSRS Substances, there are 8 broad kinds of substance JSON models supported:

* Concept (substanceClass = “concept”)
* Chemical (substanceClass = “chemical”)
* Protein (substanceClass = “protein”)
* Nucleic Acid (substanceClass = “nucleicAcid”)
* Structurally Diverse (substanceClass = “structurallyDiverse”)
* Polymer (substanceClass = “polymer”)
* Mixture (substanceClass = “mixture”)
* Group 1 Specified Substance (substanceClass = “specifiedSubstanceG1”)

The model differs in specifics for each kind of substance, but all substances share certain core fields like names, codes, relationships, and references.

For a chemical JSON, there are 2 additional root-level elements that can be used to capture chemical information: “structure” and “moieties”. An example chemical JSON is shown below:

|  |
| --- |
| {  "substanceClass": "chemical",  "names": [  {  "name": "METHANE",  "references": [  "67c0913e-aa4f-4c81-858f-26f4a1bfa6ab"  ]  }  ],  "structure": {  "molfile": "\n JSDraw204262214582D\n\n 1 0 0 0 0 0 0 V2000\n 14.6640 -6.2400 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0\nM END\n",  "stereochemistry": "ACHIRAL",  "opticalActivity": "none",  "references": [  "67c0913e-aa4f-4c81-858f-26f4a1bfa6ab"  ]  },  "references": [  {  "uuid": "67c0913e-aa4f-4c81-858f-26f4a1bfa6ab",  "docType": "WEBSITE",  "citation": "methane lovers club 2020",  "publicDomain": true  }  ],  "access": [  "protected"  ]  } |

This example shows a JSON record for “METHANE”, with a supplied chemical structure as a V2000 CTAB (molfile) representation in the “molfile” JSON property. In addition, it also specifies 2 additional optional properties called “stereochemistry” and “opticalActivity”. The structure “molfile” is typically the only required field for specifying a chemical substance definition. Other chemical fields will be generated automatically by the GSRS software if not supplied. Other fields like “moieties”, “properties”, “atropisomerism” and “stereoComments” are also occasionally necessary to fully specify a chemical substance definition.

The GSRS JSON for protein records is built similarly to chemical substances, but instead of using the “structure” field, it uses the “protein” field.This field can be used to specify definitional information about the protein definition. An example is shown below:

|  |
| --- |
| {  "substanceClass": "protein",  "names": [  {  "name": "OXYTOCIN",  "references": [  "f7d500d1-12bd-4141-8195-f12f30de1974"  ]  }  ],  "protein": {  "subunits": [  {  "subunitIndex": 1,  "sequence": "CYIQNCPLG"  }  ],  "disulfideLinks": [  {  "sites": [  {  "residueIndex": 1,  "subunitIndex": 1  },  {  "residueIndex": 6,  "subunitIndex": 1  }  ]  }  ],  "references": ["f7d500d1-12bd-4141-8195-f12f30de1974"]  },  "references": [  {  "uuid": "f7d500d1-12bd-4141-8195-f12f30de1974",  "docType": "WEBSITE",  "citation": "oxytocin lovers club 2021",  "publicDomain": true  }  ],  "access": [  "protected"  ]  } |

Much like with the chemical and concept examples, this protein example includes a name and a reference. In this case it also contains an amino acid sequence “CYIQNCPLG” found as a protein ”subunit”. The record also contains a “disulfide link” which specifies how the cysteine residues link to each other. This example is for the substance “oxytocin” which is a small peptide that can either be represented in GSRS as a chemical record or as a protein record (there are mechanisms to allow it to be stored both ways as well, but this is beyond the scope of this document).

Each of the other substance types have their own models as well. We encourage you to explore and download JSON examples from the GSRS web site hosted by NCATS.

# Section 2: Using Bulk GSRS JSON Format (.gsrs files)

In addition to single-record JSON, GSRS software also allows bulk download and bulk upload of “.gsrs” extension files. These files are simply compressed forms of the single record JSON with the following conventions:

1. GZIP compression is used
2. There is one JSON record per line
3. All JSON records inside have no unescaped newline/carriage return characters
4. The file is tab-delimited with the first two columns ignored and the third containing the GSRS JSON record. The first two columns can be used for identifiers or comments, but there are no strict requirements.

NCATS makes a full GSRS substance dataset available as a “.gsrs” file on the release web page here, under the “public data” button:

<https://gsrs.ncats.nih.gov/#/release>

# Section 3: How to Load a Single JSON Record

GSRS JSON records can be registered via the REST API using the HTTP POST endpoint. The recommended process is to first construct the record JSON, then validate the record by doing an HTTP POST request to the `/@validate` API endpoint, and then submit the record via an HTTP POST to the submission endpoint. A simple example of this process is provided below.

For this example, we recommend you use "**bash**" to run these commands in a terminal. If you use MS Windows, you can install **Cygwin** or **Git Bash**. Alternatively, you can use a program like "Postman" or install the "Rest Client" extension on **Firefox, or use HTTP libraries available in most modern programming languages**.

The example below creates a substance “concept”, validates it, persists it, and then modifies it. A substance "concept" is a very non-specific substance class, typically used as a placeholder or generic record. We are using “concepts” in this example only because they have the fewest requirements in terms of data fields/collections, but any other class of substance would also work using the same process. The example chemical and protein JSON used in Section 1 could also be used here.

For this example, we assume:

* You have not removed the default "admin" username after installation, nor changed the default password, "admin"
* Either the substances service, or the gateway service with a route to the substance microservice, is running on localhost:8081
* You won't keep the substances in a “real” system.

## Step 1: Prepare a GSRS JSON record

A JSON record can be prepared as described in section 1 of this guide.

## Step 2: Validate the JSON record via POST to the REST API

### To run validation on a substance record without submitting, you can embed the prepared JSON within a POST request to the “/@validate” endpoint, as shown below:

|  |
| --- |
| curl -X POST -H 'Content-Type: application/json' -H 'auth-username: admin' -H 'auth-password: admin' -i http://localhost:8081/api/v1/substances/@validate --data '  {  "substanceClass": "concept",  "names": [  {  "name": "DRINKING WATER EXAMPLE",  "references": [  "e7dcc059-7f47-4815-8444-2157381b8f17"  ]  }  ],  "references": [  {  "uuid": "e7dcc059-7f47-4815-8444-2157381b8f17",  "citation": "Some Citatation for Drinking Water Example",  "docType": "WEBSITE",  "publicDomain": true,  "access": []  }  ],  "access": [  "protected"  ]  }  ' |

This will return a JSON response that includes validation warnings, errors, and other information that may be of use in confirming the accuracy of the JSON submission. Once all validation messages of type “ERROR” are resolved, you can instead POST the JSON directly to the REST API to be created and persisted in the database.

## Step 3: Submit GSRS JSON to persist in the database

### To persist a new substance record using the GSRS REST API, you can run the following curl command:

|  |
| --- |
| curl -X POST -H 'Content-Type: application/json' -H 'auth-username: admin' -H 'auth-password: admin' -i http://localhost:8081/api/v1/substances --data '  {   "uuid": "f6391de8-19ad-489c-a6e8-c156fdfbf91b",  "substanceClass": "concept",  "names": [  {  "name": "DRINKING WATER EXAMPLE",  "references": [  "e7dcc059-7f47-4815-8444-2157381b8f17"  ]  }  ],  "references": [  {  "uuid": "e7dcc059-7f47-4815-8444-2157381b8f17",  "citation": "Some Citatation for Drinking Water Example",  "docType": "WEBSITE",  "publicDomain": true,  "access": []  }  ],  "access": [  "protected"  ]  }  ' |

If all goes well, and there are no validation errors, this should return a full JSON representation of the persisted record. This version of the JSON will include some auto-generated inventory fields like *uuid*, *createdBy*, and *version*. These fields aren’t necessary when creating a record but are very useful for retrievals and updates.  
  
If you want to be absolutely certain the record has been persisted, do a GET on the record and verify the content of the new JSON as explained below.

**A note on UUIDs:** In the above example the following **uuid** was specified at the root-level of the JSON object:

|  |
| --- |
| "uuid": "f6391de8-19ad-489c-a6e8-c156fdfbf91b", |

GSRS only requires UUIDs on references for initial submissions (so they can be linked to other data elements in the same submission), however a client-side generated UUID can be generated for any data element before submission and GSRS will use that UUID as long as it’s valid and unique within the tables where it’s used. The advantage of generating the root-level UUID client-side is that further communication/updates can be anticipated even before GSRS responds, but this is not required.

## Step 4: Retrieve the record that was just persisted

In the above POST example, we hard-coded the UUID for the root record to be f6391de8-19ad-489c-a6e8-c156fdfbf91b, so we can use that UUID to fetch the full record via HTTP GET request. If you let GSRS generate the UUID on the server, you can instead use the UUID that came back with the record after submission in step 3.

To fetch a simplified/ “preview” version of the record, use the following GET command:

|  |
| --- |
| curl -X GET -H 'auth-username: admin' -H 'auth-password: admin' -i "http://localhost:8081/api/v1/substances(f6391de8-19ad-489c-a6e8-c156fdfbf91b)" |

To fetch the full version of the JSON, instead use the “view=full” parameter:

|  |
| --- |
| curl -X GET -H 'auth-username: admin' -H 'auth-password: admin' -i "http://localhost:8081/api/v1/substances(f6391de8-19ad-489c-a6e8-c156fdfbf91b)?view=full" |

Sometimes it’s useful to see the more complete “internal” view of the record (containing more derived and system fields). This can be obtained with the “view=internal” parameter:

|  |
| --- |
| curl -X GET -H 'auth-username: admin' -H 'auth-password: admin' -i "http://localhost:8081/api/v1/substances(f6391de8-19ad-489c-a6e8-c156fdfbf91b)?view=internal" |

## Step 5: Make an update and re-save a record

After retrieving a record via GET as above (using either view=internal or view=full), that JSON can then be modified to change values and resubmitted to store an update. In this example, we will consider the case of adding a new name (synonym) to the record. To do this, we first fetch the internal JSON (view=full would also work here) and redirect it to a file called “tmp.json”:

|  |
| --- |
| curl -X GET -H 'auth-username: admin' -H 'auth-password: admin' -i "http://localhost:8081/api/v1/substances(f6391de8-19ad-489c-a6e8-c156fdfbf91b)?view=internal" > tmp.json |

Next, open the file for editing using nano:

|  |
| --- |
| nano tmp.json |

To add a new name, find the section of the JSON labeled “names”, and right after the first “[“ but before the first “{“, add this new name object:

|  |
| --- |
| {"name": "H2O FOR DRINKING EXAMPLE","references": ["e7dcc059-7f47-4815-8444-2157381b8f17"]} |

The resulting JSON should have a section that now looks like this (“…” signifies there’s more before and after):

|  |
| --- |
| ..."names":[{"name": "H2O FOR DRINKING EXAMPLE","references": ["e7dcc059-7f47-4815-8444-2157381b8f17"]},{... |

Save the tmp.json file using control+O, press enter, and then exit nano with control+X.

Next, to validate the change via HTTP POST to the `@validate` endpoint:

|  |
| --- |
| curl -X POST -H 'Content-Type: application/json' -H 'auth-username: admin' -H 'auth-password: admin' -i "http://localhost:8081/api/v1/substances/@validate" --data @./tmp.json |

Review any warnings and confirm there are no errors. When ready, submit the actual update operation using the HTTP PUT command:

|  |
| --- |
| curl -X PUT -H 'Content-Type: application/json' -H 'auth-username: admin' -H 'auth-password: admin' -i "http://localhost:8081/api/v1/substances" --data @./tmp.json |

On both commands above note the use of the `@` mechanism to specify the file rather than directly sending the JSON in a single line.

Once the update is submitted, you can verify that it works as expected by re-retrieving the full record again here:

|  |
| --- |
| curl -X GET -H 'auth-username: admin' -H 'auth-password: admin' -i "http://localhost:8081/api/v1/substances(f6391de8-19ad-489c-a6e8-c156fdfbf91b)?view=internal" |

The GSRS substance JSON will include a “version” field. If that field is anything other than the current value stored in the database at update time, the update PUT request will fail, as this is a sign that the record has been updated since the time the JSON was checked out. To make subsequent edits, then, it’s best to re-fetch the full JSON each time, make an edit, and re-PUT the full JSON with changes made. Future versions of GSRS REST API may support PATCH or partial updates.

# Section 4: Loading multi-record JSON files

There is a facility within GSRS for importing multiple-record JSON files (“.gsrs” files). To access this, log into your GSRS instance with an administrative account (the default ‘admin’ account will work). Navigate to the Admin panel and then to the Data Management tab. Use the “Select File to Upload” button to locate a “.gsrs” file.

Note: the extension of the disk file does not actually matter, but .gsrs extensions are used by default when exporting from a GSRS system. Also, GSRS can handle the file whether it’s gzipped or uncompressed but it must be tab-delimited as described in section 2.

Once you have selected your file, click the ‘Upload’ to start the import process. The system should display progress information as the file is processed.

# Section 5: The future

Note: the GSRS team is planning major enhancements to data exchange within the next year. Please let us know what types of data exchanges you have in mind!