Guide to creating custom exporters

|  |  |  |
| --- | --- | --- |
| Version | Date | Comments |
| 0.1 | 21 August 2024 | First draft for comments |
| 1.0 | 24 September 2024 | Removed ‘draft’ designation |

Contents

[Introduction 2](#_Toc175169702)

[Data Scrubbing 2](#_Toc175169703)

[Expanders 3](#_Toc175169704)

[Schematic of the GSRS Export process 3](#_Toc175169705)

[Steps in developing an exporter: 3](#_Toc175169706)

[Testing and deploying your exporter 5](#_Toc175169707)

# Introduction

In the context of GSRS, an *exporter* is a software component responsible for outputting a specific projection of data from the system in a specified format, generally to a file on disk.

Exporters may offer options to the user (for example, whether to include certain fields or collections). When the exporter does offer options, it should provide a *schema* (through the factory class explained below) that describes the options.

Exporters typically consist of 2 classes:

1. The exporter that does the actual work of writing entity data to a file
2. The exporter-specific ‘factory’ class which sets up the exporter and communicates with GSRS.

This document walks the reader through the process of creating an Exporter.

GSRS entities can be very complex, consisting of a base class with a set of fields as well as one or more subordinate collections that may contain several levels of hierarchical data. For many purposes, a user needs only a small subset of the data. For example, a user who wants to run a predictive model needs the structure for all the chemicals on a list of records, along with an identifier. An exporter selects some subset of the data, for example, the names associated with a substance and writes these to disk and GSRS makes the disk file available to the user as a download.

Exporters typically rely on the GSRS export framework to prepare the set of records:

1. Providing a way to select records
2. Removing records or fields and collections that should not be included in the export operation for confidentiality issues. The removal of data is controlled by a set of options that the user can select. This is explained below.
3. Expanding the set of records to include some additional records closely linked to the user-selected records. This process is explained below.

## Data Scrubbing

GSRS defines a *RecordScrubber* interface. Classes that implement this interface can be configured to clean data for export. RecordScrubber has a single method:

Optional<T> scrub(T object);

There is currently one implementation, the BasicSubstanceScrubber within the substance module. However, it’s possible to create a scrubber for all types of entities.

The options available for scrubbing data during export are available through a URL such as …/api/v1/substances/export/scrubber/@schema

Note that scrubber options apply to all exporters in the system and are different from options offered by individual exporters.

Some examples of supported scrubbing operations:

* An entire record must be removed from the exported data list because it is confidential. (In this case, the scrubber will return an empty Optional.)
* One name assigned to a substance is for in-house use only and should not be exported. Delete the name from the copy of the record being exported.
* The approval ID of a substance should be copied to a new code and removed.

## Expanders

An *expander* adds records to a list based on user-selected requirements. For example, a set of records includes a substance of type mixture. To fully understand the mixture, the user wants to also export the component substances. The expander adds mixture components (and/or Specified Substance Group 1 constituents, parent substances of structurally diverse substances, and other related substances) to the list based on user selections. The options available for data expansions are available through a URL such as

…/api/v1/substances/export/expander/@schema

## Schematic of the GSRS Export process

A close-up of a sign

Description automatically generated

Notes:

1. ‘Records’ may come from a search, faceted browsing (where you start browsing substances and then use the facet menu to reduce the number of records) or other operation.
2. The first scrubbing process removes private records and/or fields based on user selections.
3. Expand adds related records to the list based on user-selected criteria.
4. The second scrubbing removes any records added to the list during the expansion that happen to be marked confidential as well as parts of newly-added records that should be removed.

## Steps in developing an exporter:

1. Define your required output
   1. Understand the fields required in the output file
   2. Understand the connection of the fields to the substance data model
      1. which fields are part of the core object, which are part of a collection, and which are calculated from one or more of the other fields.
   3. Determine the formatting of data in the output file
2. Understand which options you will provide for your users
   1. For example, some parts of the output may be optional. In this case, add a parameter to your exporter ‘includeOptionalData’ that may be true or false.
   2. You will need to add a getSchema() method to your factory class, as explained below.
3. Figure out what common errors you will anticipate and how you will handle each.
4. Figure out the file format your exporter will use.
   1. Note: GSRS architecture requires that each file extension map to at most ONE data exporter. If you want your exporter to create, for example, text files, you will need to create a new extension. (It is fine to use a long extension with 2 sections, such as ‘.codes.txt’)
5. Write the exporter class.
   1. It must implement the Exporter interface.
   2. It should have a private member of type BufferedWriter.
   3. Do any required set-up in the constructor, including instantiating the BufferedWriter from the supplied OutputStream parameter.

this.out = new BufferedWriter(new OutputStreamWriter(outStream));

Your constructor may also have additional parameters. The only calls to the constructor will come from your factory class (below) and any unit tests you write.

* 1. In the export method, extract the fields you want and write them to the BufferedWriter.

e.g.,

out.write(substance.getName());

* 1. In the close method, close the BufferedWriter.

1. Write the exporter factory class.
   1. It must implement the ExporterFactory interface.
   2. Write the supports() method, evaluating the supplied Parameters input parameter to see if it shows the required extension.
   3. Make the getSupportedFormats() return the set of OutputFormats you support.

Note: you can create a subclass of OutputFormat or instantiate one or more OutputFormat objects with the required fields: extension and display name. for example,

new OutputFormat("data.txt", "Substance Property File(.data.txt)")

* 1. Make the createNewExporter() method return a new instance of the exporter class.
  2. If you have any runtime options, write a getSchema() method that returns a JsonNode (ObjectNode) with a listing of the available options.

For example,

@Override

public JsonNode getSchema() {

ObjectNode parameters = JsonNodeFactory.instance.objectNode();

ObjectNode parameterNode = JsonNodeFactory.instance.objectNode();

parameterNode.put("type", "boolean");

parameterNode.put("title", "Restrict output to defining properties?");

parameterNode.put("comments", "Limit output to properties marked as defining?");

parameters.set(DEFINING\_PARAMETER\_NAME, parameterNode);

return generateSchemaNode("Substance Property Exporter Parameters", parameters);

}

As an example, we have created a SubstancePropertyExporter which is available at <https://github.com/ChemMitch/GSRSSampleExporter/>

Requirements for this exporter:

* List all Properties associated with a substance
* Fields: UUID (substance), primary name (substance.getName()), number of properties, for each property
  + Property name, date created, property name, property type, value (see below), defining (Boolean flag)
  + Value output:
    - When present, include average and units
    - When no average but high and/or low, include this with units when present.
    - When neither average nor high/low, include non-numeric value
* Parameters
  + onlyDefining – when true, only output parameters when the defining flag is set to true.

**A note about parameter handling:**

To determine what value(s) the user has supplied for the input parameters of your exporter, you must check the Parameters-type input parameter of your createNewExporter method in the factory class. That method has this signature:

public Exporter createNewExporter(OutputStream out, Parameters params)

(The Parameters type is created in the ExporterFactory interface within the GSRS Starter and has a *detailedParameters* field, of type JsonNode. You will see one child node called detailedParameters for each parameter. Use the value of the child nodes to determine the value of the exporter parameters. You cannot rely on the runtime calling any parameter setters on your Exporter or ExporterFactory classes!)

## Testing and deploying your exporter

You can write your exporter in a Java project separate from GSRS. Make sure the compiled JAR file is available to your local Maven repo.

Add the exporter to the application.conf file for your substance service

ix.ginas.export.exporterfactories.substances +=

{

"exporterFactoryClass" : "gsrs.module.substance.exporters.SubstancePropertyExporterFactory"

}