GSRS History and Design Rationale

Danny Katzel

Dec 2021

|  |
| --- |
| **Note:** This document is an informal draft-form document and is likely to be updated with corrections and clarifications in the future. |

Table Of Contents

[Introduction 2](#_Toc1132970360)

[GSRS vs GSRS vs GInAS 3](#_Toc1334362416)

[Evolution of GSRS codebase and other NCATS Projects 4](#_Toc1260228423)

[Why Some Packages are in Ix and others are in GSRS 6](#_Toc1051628223)

[Why Rest API doesn’t match the standard Restful Patterns 9](#_Toc1374878500)

[POJO Diff 9](#_Toc1716022843)

[Description 10](#_Toc635169607)

[Rationale 10](#_Toc1016354687)

[JSON Bean Views 10](#_Toc2119233284)

[Description 11](#_Toc1952678525)

[Why Partial Updates and JSON Patch of Entities are not Supported 12](#_Toc1026767770)

[Description 12](#_Toc1365894832)

[Rationale 12](#_Toc1502133330)

[@SingleParent 12](#_Toc373543053)

[@ParentReference 13](#_Toc451067058)

[Relationships 15](#_Toc657456040)

[Why Relationships Are Not Top Level Entities 15](#_Toc1856303604)

[Relationships and Versioning 17](#_Toc1354402565)

[Relationships Can Be Dangling 17](#_Toc95424445)

[Problems Editing Relationships Because There are Two Sides 19](#_Toc2137443088)

[Why Relationships have an Owner 19](#_Toc1765221295)

[Why do Names have name and full name fields 22](#_Toc1893790685)

[What is Name.name vs Name.stdName 22](#_Toc1389720495)

[Molwitch 23](#_Toc1460576651)

[Converting an Entity Object into a GSRS 3 Route 24](#_Toc2089041062)

[Why is there no code in Entity classes that directly link to Controller classes to get routes 25](#_Toc1410026105)

[GsrsUnwrappedEntityModelProcessor and annotations @EntityMapperOptions and @GsrsApiAction 26](#_Toc1344998976)

# Introduction

This document describes the general system architecture and rationales behind some of the design decisions of GSRS throughout its history. Design decisions made several years ago still affect new or future releases due to reasons not limited to API backwards compatibility, limitations on dependent frameworks or libraries, and bug fixes. Some design decisions may seem strange until the reasoning behind the decisions is explained. Some alternative designs are also mentioned along with the reasoning for not choosing the alternatives.

This document is a historical guide through the trials and tribulations of past and present versions of GSRS so that design mistakes are not forgotten or made again.

Each Architecture Element will be described in the same style:

1. Summary – each section will provide a brief summary in bold to give readers an overview of this element.
2. Detailed description – a technical description of the element and sub-elements. This may include diagrams and/or tables or code
3. Design Rationales – Explanation of why certain design choices were made or not made. The design history of this element might also be explained to provide supporting evidence for why the design has evolved into its current form.

# G-SRS vs GSRS vs GInAS

**This project has had at least 3 names over the years of development. Originally the project, consortium, *and* software were all ambiguously called “GInAS”. The software core was eventually changed to “G-SRS” with a hyphen and now it is exclusively called “GSRS”. The terms “ginas” and “GInAS” is still sometimes used for the overall project, but are largely being phased out.**

### Description/Rationale

The Global Ingredient Archival System (GInAS) was the original name of the ISO 11238 substance management endeavor that brought many stakeholders from around the world together. This term was used somewhat ambiguously for the project, the software, specific installations of the software at FDA, the FDA data itself, and for the team of stakeholders themselves. This led to a lot of confusion about the *specific* meaning of the term. The specific meaning was often only determinable via context. Eventually, we agreed to call the core software tool itself “G-SRS”/”GSRS” (eventually “GSRS”). The FDA’s implementation of and installation of the GSRS is also simply called “GSRS” or sometimes “FDA-GSRS”.

The previous ingredient registration system and software in the FDA prior to 2017 was called “SRS”. It was natural to have the new GInAS version of the software called “G-SRS” to reflect the newer “global” perspective of the software while still maintaining the roots in the FDA’s work composing the SRS (which itself helped inspire the ISO 11238 standard). Other installations of the GInAS codebase (now called GSRS) were installed in the European Union as EU-SRS, and it’s become common to replace the “G” in “G-SRS” with a specific region or agency to describe a GSRS installation.

“GInAS” and “ginas” are still sometimes used in the source code, URLs and database schema. The intention was for the overall project to still retain that name as well, but this has slowly become less common. Besides ambiguity in terms, the common pronunciation of “ginas” as “ji-nas” instead of the intended “jee-nus” led to some linguistic discomfort, so terms like “The GSRS Project” or “GSRS consortium” are more common now than “GInAS”.

Some people called the software G-SRS and others called it GSRS. Starting around 2019, we were more careful to make sure we were consistent in calling the NCATS version GSRS without the hyphen. However, there are still many cases where a hyphen may be found.

# Evolution of GSRS codebase and other NCATS Projects

**Once upon a time, there was a common NCATS codebase that was used for many NCATS DPI projects. Classes, design decisions, and naming conventions all come from this codebase which was called Inxight.**

### Description

The Inxight codebase was originally intended to be for common reusable components of NCATS webservices. Several NCATS projects including: GSRS, BARD, Pharos, Tox21 and Inxight:Drugs, and an app which tracked NCATS Publications can all be traced back to this project. All of these projects were originally written in Java using the Play! Framework and were designed by the same developers. They all used the same Domain Model objects for common components such as Users (Principals and UserProfile), linkouts to publications, and common abstract Model classes (IxModel).

The Java package names for Inxight started with `ix` which is why a lot of the oldest fundamental classes of GSRS are in the ix package as well. GSRS also still has a few different ways to send messages around to other model objects that aren’t used much except in small corners of the code can also be traced back to the Inxight codebase. For example, GSRS still uses Akka to notify objects.

Over the years, there have been a few security vulnerabilities discovered in this original codebase that affected all the services mentioned above because they all reused the same code.

Eventually, the unique feature demands and limitations of the Play! Framework caused each project to splinter off into their own code repository and go in their own direction without needing to worry about making breaking changes to the other projects. But the impact of those early decisions in both object model, database design and REST API can still be felt in GSRS today nearly 10 years later.

During the transition from GSRS 1 to GSRS 2, as much of the unneeded code from these unrelated projects was excised from the GSRS codebase, but anything that even indirectly affected table design remained. Also, many of the GSRS 2 conventions of RouteFactories and naming conventions of routes and method names such as “instrument” remain.

### Rationale

It made a lot of sense in the beginning to make a common framework for all NCATS projects. It made it very easy in the early days for developers to jump into unfamiliar projects because much of the design and objects were the same.

Code reuse and following the “Don’t Repeat Yourself” (DRY) principles are good ideas to follow but trying to shoehorn every feature of every project into a design that fit the others as well was very hard. This also bloats the code of GSRS with tons of classes that aren’t needed but get accidentally used and incorporated into data models that can’t be undone for backwards compatibility reasons. Many of the model classes had JPA annotations and therefore when GSRS auto generated the database schema, many of these unused model objects were immortalized into database tables.

# Why Some Packages are in Ix and others are in GSRS

**GSRS Java code is split between two main base packages: ix and gsrs. Code in the ix package is older code from the GSRS 1 and 2 days that couldn’t be moved for backwards compatibility reasons.**

### 

### Description

GSRS Java code is split between two main base packages: ix and gsrs. As stated in Evolution of GSRS codebase and other NCATS Projects, originally the GSRS codebase was part of a project called Inxight and all code in that project had a base package name of “ix”. When the GSRS code was forked off of Inxight the ix package convention remained and most of the code was either in ix.core or ix.ginas.

Newer code has tried to move away from the ix packages as much as possible and instead use the gsrs package, but we couldn’t move the old code out of the ix package so there is still a mix of old ix code and new gsrs code.

One way that we have limited the need to see ix package code is to try to make parent interfaces in the gsrs packages so only “legacy” code needs to know about the ix package. Spring Boot auto wiring makes this even easier so that the auto wired interfaces are mostly now in the gsrs package even if the implementations that get injected into the Spring Beans are in the ix package.

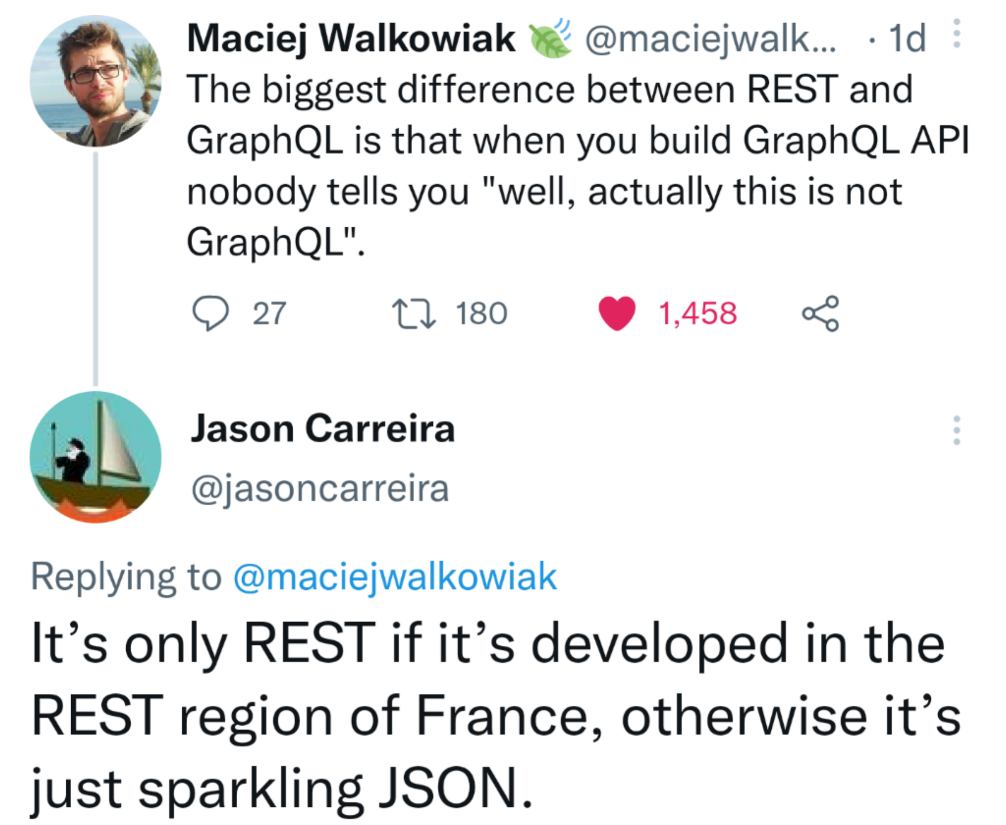
Rationale  
The GSRS 3 rewrite tries to maintain backwards compatibility as much as possible. Several places in the GSRS JSON, which is the definitive definition of what a Substance is, API queries, and search facet records sometimes contains java package names! So, if we moved those classes out of the ix package, old JSON data, saved queries and facet drill downs users were used it would no longer correctly work.

|  |  |
| --- | --- |
| Search API kind filter is the fully qualified package name | app/api/v1/substances/search/@facets?kind=ix.ginas.models.v1.Substance |
| Controlled Vocabulary JSON contains qualified class names | { "id": 1, "deprecated": false, "domain": "ACCESS\_GROUP", "vocabularyTermType": "ix.ginas.models.v1.ControlledVocabulary", "editable": false, "filterable": false, |
| Substance Facets contain qualified class names | "name": "ix.Class", "values": [ { "label": "ix.core.models.Keyword", "count": 17 }, { "label": "ix.ginas.models.v1.Name", "count": 17 }, { "label": "ix.ginas.models.v1.Reference", "count": 17 }, |

Some developers have also extended GSRS to make their own IndexValueMakers, ScheduledTasks and Exporters, in order to allow their code to be binary compatible and not have to be rewritten, we had to keep those interfaces in the ix package.

Another reason to keep the ix package around at least initially during the GSRS 3 back-end design rewrite was to make it easier to port the code over. Development of GSRS 2.6, 2.7, and 2.8 happened concurrently with the GSRS 3.0 re-design effort, so bug fixes and new features would be added to both code bases and it was just easier to keep the packages and folder structures the same so that diffing the code and checking to see what was missing from one or the other could be done more easily.

# Why Rest API doesn’t match the more common Restful Patterns



**A GSRS entity is exposed as a web resource via a REST API. The GSRS REST API uses a syntax inspired by ODATA** [**https://www.odata.org/**](https://www.odata.org/) **The format of the URLs is distinct from many of the most popular REST APIs. Perhaps most striking is the use of parentheses around an entity’s id in order to retrive that record.**

### Description

Common Restful API syntax

/api/substances/myId

GSRS API

/api/substances(myId)

### Rationale

When NCATS was helping to create the BioAssay Research Database (BARD), we agreed on using an ODATA-like syntax with IDs in parentheses. This format was gaining traction at the time, and while not all conventions from ODATA were adopted the general mechanism of having a resolving term surrounded by parentheses was used. One of the big advantages of this syntax was the clear separation between operational endpoints vs single record retrievals. Early versions of both the prototype GSRS REST API and the InXight framework were based off of some of these same conventions, and GSRS has maintained those conventions.

For added compatibility, many GSRS endpoints also support the more common slash separating format of most Restful APIs, but most hard-coded tools and server-side generated links use the parenthetical REST API syntax.

### Advantages:

1. Using a pattern where an ID/resolution term is found within parentheses makes it exceptionally clear that the term is meant as an ID rather than an endpoint. GSRS REST API has several high-level endpoints for operations like the **“/substances/search”** and **“/substances/structureSearch”** endpoints, as well as many others starting with an “@” character, and new high-level search/retrieval endpoints are sometimes added for all entities in this manner. Supporting this syntax while also supporting the **“/substances/<identifier>”** endpoint is inherently ambiguous. If a substance or other entity were to have the “<identifier>” of “search”, “structureSearch”, or especially a character starting with the “@” sign, it becomes unclear whether endpoint operation or single record retrieval was intended. Using parentheses makes it clear that a single record with a very specific identifier was intended, even when simply looking at the URL in a log, script, or response.
2. Regular expressions to parse out the resolution ID are simpler when they are surrounded by “(“ and “)” characters.
3. Historical GSRS tools and scripts expect the use of this pattern primarily.

### Disadvantages:

1. It's more common in most REST APIs to either prefer the “/<identifier>” notation or notations such as “?identifier=<identifier>”.
2. Spring boot and other frameworks are not typically set up to easily support the parenthetical notation, so some extra development work is needed.
3. While the pattern is inspired by ODATA, it doesn’t follow all conventions of ODATA, and may be misleading to those who have libraries and tools prepared to interact with ODATA patterns.

POJO Diff

**One of the core components of GSRS is to keep figuring out what has been changed in an entity’s JSON and apply those changes to the persistent java bean objects so that the database is updated.**

### Description

PojoDiff (POJO = Plain Old Java Object) is custom software component that tries to find all the changes in GSRS JSON entities that are relevant and require database updates. PojoDiff goes beyond just comparing changes to the JSON before the edit and after as some differences in the JSON aren’t actually modifying any real data. For example, if the order of elements in a list of changes that shouldn’t be considered a change.

### Rationale

Reminder that the GSRS official entity record is the JSON, not necessarily how it’s stored in some relational database. The preferred update operation is via the REST API using an HTTP PUT request, replacing the entire old JSON with an entirely new JSON object. The new JSON is validated and business rules and triggers are applied based on how the object has been changed. If a NoSQL-like solution were used, this would be a fairly trivial operation. However, GSRS *also* does ORM to store the “true” meaning of the data into RDBMS tables while backing up the serialized JSON into another table. The RDBMS storage of this data is used to allow custom SQL queries, as well as special indexing and reports, especially for legacy systems at FDA which require the existence of these RDBMS tables. In short, submitting a “new” full JSON is not a simple operation of replacing one large JSON field, but must instead be changed into a series of “delta” SQL update, insert and delete statements. Neither Hibernate not Ebean ORM directly support the concept of computing the necessary “delta” SQL statements between 2 complex nested objects, nor do they directly support the concept of mutating one complex nested object into equivalence with another complex nested object such that any change operations are appropriately flagged as “dirty” operations. Doing this is not always trivial. PojoDiff is a tool to aide this process.

When a JSON submission comes in via the REST API, it is deserialized to a POJO, and compared to the existing POJO deserialized from the database. A set of update operations (using the getters and setters of the POJOs) are applied to mutate the “attached” POJO pulled from the database into equivalence with the POJO obtained from the REST API submission. These changes employ the use of a JSONDiff library to detect changes between the JSON representations of these objects, but also pay special attention to identifier fields and certain annotations about which fields are required for mutation. One especially important element of POJODiff is its ability to ignore changes which are simple re-ordering of saved sub-entities (like names and codes) within the POJOs, which would have no specific meaning in the RDBMS sense.

In some cases, POJODiff cannot account for all the changes that may happen when a single record is updated with a new JSON object. This typically happens when the fundamental object class of the root object changes (as is the case when an update changes a substance class from a chemical to a protein, for example). In such cases, rather than mutate and resave the original object, the full old object is completely removed from all tables (which may require dozens or even hundreds of SQL delete and update statements), and then the new POJO object is saved, typically resulting in repopulating most of those rows which were just deleted.

## JSON Bean Views

**GSRS has URL parameters to change how much JSON data should be returned by the API.**

### Description

Some Substance records have a lot of data or link to very large files and in order to be the fundamental definition of a Substance we need to include all this data; but sometimes we don’t need the full data and only need part of it. The view parameter allows us to tell the backend what data can be collapsed or hidden.

The Jackson JSON library has the concept of a Bean View. We can specify on POJO fields or methods which views that field/method should be included on. Jackson also respects java inheritance of views so subclasses also work so some view classes are subclasses of other view classes which allow us to have finer grained views.

While Jackson Views do most of what we want when restricting views, there are some special case views we also need to support that go even farther than what Bean View can provide. See the Key View for more information.

Bean View also does not handle the access restrictions.

### Rationale

The JSON Bean View is built into Jackson so it was used early on but it doesn’t do as much as we need to restrict access to some fields in GSRS. Other types of views like Key view don’t even use the Java Object fields at all so GSRS software has to know intimately about what view(s) are requested.

Why Partial Updates and JSON Patch of Entities are not Supported

**The JSON of a GSRS Substance is only allowed to be passed around as a complete record. Therefore, when creating or updating a Substance, the complete JSON must be passed in.**

### Description

In the ISO 11238 Standard, the Substance JSON must be “fully defining” so everything that goes into that definition of what a substance is must be present. Therefore, when creating or updating a Substance, only the complete JSON is passed in. Even if a user is only updating one of the names of a Substance, the entire Substance JSON with the complete definition of what that Substance is required.

### Rationale

There are many reasons for this. In general, supporting updates for such a complex data structure as those stored in GSRS can be achieved in several ways. GSRS chose a default approach which allowed for ease-of-use for an API consumer / UI, simplification of backend validation rules and triggers, protection against double-editing, and simplification of explicit backend endpoints. A brief overview of the approaches considered are described below:

1. “**Ad-hoc**": Create specific endpoints for each kind of data update operation. For example, `substances/@addName` and `substances/@updateName`, etc. This is how the legacy FDA SRS system worked, with specific logic coded for each endpoint, new UI and new logic added whenever there were new additions to the model or changes to business logic. All endpoints responsible for their own data validation. While one-off custom data operations can have success with this approach, it is hard to maintain this approach systematically.
2. **“Field-specific-PUT”**: To update any specific element of a record, you specify a URL pattern for that identified record and then give the JSON-pointer-like notation to explicitly find the field/object intended to update, replacing its contents with the supplied JSON present in the body of the PUT HTTP request (triggering whatever updates and validation rules may be necessary). For example, a PUT to “/substances(benzene)/status” would replace the value of the “status” field of the substance record found in the benzene record with whatever value is specified in the PUT request. Early versions of GSRS used this convention and future versions may support it as well. The main challenges here are that the exact syntax for specifying elements in arrays that do not have completely deterministic ordering is challenging. So, for example, trying to change only the deprecation status of the 1st naming organization found on the 1st name of the benzene record may look like a PUT to “/substances(benzene)/names/0/nameOrgs/0/deprecated”. However, there is no rigorously enforced ORM requirement that the order to the names or nameOrgs (or any list formed from a joining table) be deterministic, so it’s possible that the wrong piece of data may be updated. Similarly, there is no inherent check that another user hasn’t added / removed / modified that same record between the time the request was constructed and the time it was executed.
3. **“JSON Patch format, using HTTP PATCH”**: Use the PATCH HTTP request type sending a [JSON-Patch](https://datatracker.ietf.org/doc/html/rfc6902) formatted JSON object which describes all the delta operations intended, to be applied as mutations to the source object. This approach uses the application/json-patch+json mime type. This is a very useful convention that’s recently become more popular, and GSRS has had some experimental code to help do just this for some time. The principal reason this approach isn’t used is for the same reason the “Field-specific-PUT” approach isn’t used: the JSON pointer notation to specify individual updates requires strictly deterministic ordering of elements within an array (the first name/code/relationship must ALWAYS be the first one, etc). As before, there’s no inherent check that another user hasn’t changed the data. These deficits are addressable with a more strictly enforced canonical ordering of object arrays pulled via ORM (e.g. requiring all sub-objects to implement Comparable and be sorted on post-load). They also may be addressable via slightly modifying the JSON Patch format to have operations based on the IDs of sub-objects rather than their array index. Requiring that each PATCH operation *explicitly* specify a version number which is one greater than the previous version would also address the double-edit issue.
4. **“JSON Merge Patch, using HTTP PATCH”**: Each time a change is intended *only* the fields of the JSON intended to change are supplied, but the structure of the JSON is as-if it is the full JSON, with non-existent fields treated as non-changes. This is the “JSON Merge Patch” standard format described [here,](https://datatracker.ietf.org/doc/html/rfc7386) using the application/merge-patch+json mime type. This is sometimes conflated with the JSON-PATCH mechanism described above. In general, this approach is quite popular for especially “flat” JSON objects, but is less beneficial for cases where, for example, a single addition to a large nested array is needed. Such a case would still typically require posting the full nested structure of all parents and siblings of that single addition. Nevertheless, this approach is beneficial in that it behaves very similar to the “Full-object-PUT” convention described below and could easily overcome issues with array ordering and double editing by enforcing a few conventions that are almost already implied. The largest issue with this approach, besides the need for a UI or other API consumer to encode the same information in reliable ways, is that the libraries used within GSRS to serialize and deserialize JSON have been tuned to consider missing JSON elements equivalent to “null” elements, and special care would be needed to ensure that those intended meanings are supported in some contexts but not all contexts.
5. **“Full-object-PUT”**: Each time a change is made to a JSON object that *full* JSON object is reposted via HTTP PUT request and is updated accordingly based on any implied delta operations needed. **This is the approach GSRS currently uses.** There are several advantages to this approach. For one, the inclusion of all fields allows for explicit version fields which can be compared against the current stored version of a record helping to prevent double editing. For another, the general pipeline which allows for creating new JSON records can also be repurposed for updates to those records (though a delta detector /mutator is needed for reasons discussed in POJOPatch section). This approach also avoids the non-canonical ordering problem of the previous described methods. Finally, and perhaps most critically at this time, the needs for a UI form which can import and modify the JSON for editing are quite straight-forward, without the frontend needing to do any special delta computations on what, specifically, the user intends to change in a given update. The main disadvantage, of course, is that HTTP traffic, at times, can even be many megabytes in order to make extremely trivial changes.

In summary: Using the “Full-object-PUT” method described above simplifies a lot of the JSON processing and JSON->POJO conversion because the entire record is passed around. A lot of the problems with updating Substances and other records comes from re-ordering of lists of items inside the JSON record; trying to update a Substance where some elements are re-ordered AND PARTIAL updates are present would make this processing even harder.

## @SingleParent

**Entities are sometimes annotated with a GSRS annotation @SingleParent to denote that it will only ever have 1 parent.**

### Description

Entities annotated with @SingleParent are specially handled in GSRS during updates in part of the POJO Diff phase and anything marked as removed during the POJO Diff that is also annotated with @SingleParent is explicitly deleted from the database.

Entity classes whose instances will only ever belong to one and only one other entity (ex: Name, Reference, Subunit etc.) should be annotated with @SingleParent.

### Rationale

Older versions of GSRS used a legacy Ebean library from around 2013 (even for GSRS in 2020) for its ORM library to map Java Objects to the database. One problem with that version of Ebean was it could not delete (or maybe even detect) orphan records via JPA annotations alone. So, if an entity had a one-to-many mapping to a List of other entities and an update removed one of those children, we probably want to also delete that row from the database if it’s not referenced by anything else anymore (an orphan).

Since the version of Ebean could not detect or delete them we would have lots of extra database rows taking up memory and possibly coming back in database queries when we don’t want it to. GSRS developers needed a way to hint to the application that something was an orphan. @SingleParent was the solution chosen since it was an easy way to check at runtime of that entity class has a SingleParent annotation.

GSRS 3 moved away from Ebean and uses Hibernate which CAN detect and remove orphan records so perhaps @SingleParent is no longer required. However, it doesn’t hurt to explicitly delete the record so it shouldn’t cause any harm to still use that annotation or change any business logic that uses it. On top of that, it is a nice marker annotation to inform developers that this class is intended to only have a single parent.

## @ParentReference

**@ParentReference is a marker annotation to map what field is used to map a child entity back to its parent or owner.**

### Description

@ParentReference was added in GSRS 3.0 to more easily denote which field is the “owner” field in a parent-child relationship. Usually @ParentReference is used to annotate a field directly, however it also supports adding this annotation to a method as well which will be explained why below.

Only use @ParentReference on a single field XOR a single method as once GSRS finds something annotated with @ParentReference it won’t look any farther to see if there are more. It also doesn’t make sense to have more than 1 parent.

In some cases of sub-optimal database design, there might be multiple fields for an owner depending on the polymorphic type. In these cases, GSRS supports adding a @ParentReference annotation to a method that returns void and takes a single Object parameter. That method implementation should check the type of the passed in parent and programmatically determine which field (if any) should be used.

### Rationale

The GSRS JSON format for Substances and most other entities do not explicitly have fields for children to refer to their parents. Since all children really have single parents, the parent is implied through the nested structure of the JSON.

In older versions of GSRS that used the Play! Framework and Ebean, somehow the “owner” fields (Many-To-One fields) within the ORM POJO would automatically get set from the JSON structure, but Spring and Hibernate don’t do this for you so the owner fields during GSRS 3 development were not getting set. Therefore, as a hint to tell the GSRS business logic which field is to point to the parent, @ParentReference annotation was created.

There is a process in GSRS 3 during JSON-> entity conversion where GSRS will not explicitly traverse the entity tree looking for @ParentReference annotated methods and fields and set them to the parent object it points to from the JSON structure.

@ParentReference was originally only designed to set a specific field but we ran into problems on new GSRS entities created in GSRS 3 that had object and table designs where there were multiple possible fields to set depending on some set of conditions. This led to allowing a method to be set with @ParentReference so that the method logic could figure out which field to set.

# Relationships

Relationships are set up in a particular way that can be confusing, error prone and at one point the source of most of the business logic bugs. This section describes why Relationships were designed the way they were.

One of the biggest strengths of GSRS is the fact that it has ways to specify how substances are related and how. Relationship objects link two substances together. Different types of Relationships include: Active Moieties, Targets, different Salt forms, sub-concepts etc. Relationships can also have other properties such as amounts and references.

Relationships have a type field that is usually of the form X -> B on one side, and B <- X on the other. Keeping both sides in reflected synchronization has caused lots of bugs and headaches.

## Why Relationships Are Not Top-Level Entities

**As of this writing, because a Relationship is not a top-level entity, it is fully owned by its Substance (or at least that side of the Relationship, see Relationships Can Be Dangling) so any changes to a Relationship have to come from its parent Substance.**

### Description

In GSRS 3.0 and below, Relationships are not their own top-level entities (or what Domain Driven Design calls an “Aggregate Root”). In GSRS, the Substance JSON specifies everything involved in defining that substance and that includes any relationships so the Substance JSON has a list of all of its relationships in addition to other lists of names and codes etc. This is a great way for making sure the Substance JSON contains everything it needs but comes at a price.

## Relationships and Versioning

**Since Relationships are not top-level entities and are instead part of their parent Substance, and are included as part of that Substance’s JSON record; any edits or changes to a Relationship end up bumping up the version of that parent Substance!**

### Description

Relationships are edited a lot as most Relationships start off as private, proprietary data and as time goes on and knowledge of that relationship becomes more public, more references or updates to that Relationship’s fields are updated. This can make some Substance’s versions go into the hundreds! This is very confusing to users who see all these version updates without anything really fundamental about the Substance changing.

One way to solve this problem would be to pull out Relationships to their own top-level entity so they don’t cause Substance’s to get a version bump.

Another way would be to have 2 or 3 different kinds of versions. One version is a lower level “how many times has this JSON been updated” which would increment with every change including Relationship updates. And another “definitive version” that would only get updated if something important to the definition got changed. A third layer, in between, could be any case where a name, code, or reference was added or updated. The hard part with that aside from all the data model changes that would cause would be ways to determine what is a defining change vs not to update the definitional version. Perhaps using the Definitional Hash?

### Rationale

This is more just a side effect of making Relationships under Substances. Any solution like those proposed above would require data model changes and some radical changes to Substance updates. One especially strong reason this hasn’t been done is that it would likely make data exchange of substances which have relationships to be performed via 2 kinds of messages. This is especially an issue when, rarely, the relationship itself is considered somewhat defining or at least a proxy for defining information (as is the case sometimes with sub concepts and MABs which have a defined relationship target but no specific sequence known).

## Relationships Can Be Dangling

**Substances can be related to other Substances that don’t yet exist in the GSRS system.**

### Description

Sometimes a Substance is related to something that hasn’t be loaded into GSRS yet. This is most often done during “partial” loads where only a portion of the full public Substance data is loaded for testing or debugging purposes and the few records loaded point to other records that are not included in the load. This was common in early GSRS data loads as data was still being ported and cleaned up from the legacy FDA systems into a GSRS and some Substances that were loaded relate to other substances that weren’t cleanup for loading yet.

When a Substance is created, there is a Post Commit processor that will look through all the Relationships and try to see if an already existing Relationship is a Dangling Relationship that is meant to point to this newly added Substance. If so, it will create a new Relationship object on the newly added Substance that is the reflection of that Dangling Relationship.

### Rationale

GSRS needed a way to be able to load a relationship where one side pointed to a Substance that didn’t exist yet. The solution was to split a relationship into 2 objects: one for each side. This way a Dangling Relationship can be loaded and additional business logic would be added to GSRS to connect to two sides together once the other side was eventually loaded which might be days, weeks or even years later.

Problems Editing Relationships Because There are Two Sides

**Because The concept of a Relationship between 2 Substances is really modeled in GSRS as 2 separate almost identical Relationship objects with only the type being a reflection of each other A -> B vs B <-A, making edits to Relationships is hard.**

### Description

When a Relationship edit is detected, GSRS must find the corresponding “other side” of that Relationship. There isn’t a simple database join table (perhaps there should be) so a slower text-based query is made where we must search the Substance on the other side for a Relationship with the same but reflected type.

What happens when there are multiple relationships with matching type but other differences in their properties (such as amount)? What happens if the Relationship change changed the String value of that type such that finding the not yet updated, reflected side no longer matches?

Problems like the ones described above are some of the main sources of bugs in previous versions of GSRS code where a Relationship either couldn’t find its corresponding other side and become out of sync or created an extra other side so there were now 3 or more edges to a Relationship.

## Why Relationships have an Owner

Relationships have 2 points to Substances: an `owner` field and a `RelatedSubstance` field. The RelatedSubstance field points to the other Substance this Relationship relates to. The owner field was the Substance that originally created that relationship. Usually this means, that that was the Substance first loaded that mentions this reference, but other times one side of the relationship is more “important” and should be the Substance that “controls” the Relationship.

Early versions of GSRS only edits from the owning Substance side were allowed to edit the Relationship, but that has been relaxed in later versions as it was too confusing for users to keep track of who the owning Substance was or would make them open a new browser window to edit the Substance on the other side to make minor changes to a relationship.

Access Lists and GinasAccessContainer

**Which Access control Groups are intended to view a particular datum is stored using a GinasAccessContainer class where a more obvious design would have been some kind of Set of Group objects.**

### Description

GSRS allows pieces of data to have access flags, which are a set of named “groups”. These are currently informational flags and do not directly control data visibility. Those groups are stored in a special data type called **GinasAccessContainer** within the code.

### Rationale

Originally, in GSRS 1.0, access control was done using a List of Group objects. However, the initial automated database design by parsing the JPA annotations, this created extra join tables practically for every entity object – adding hundreds of tables.

The first attempt to fix it was to use a BitSet with the on bits being what Groups were being used. But this also caused issues if we were to ever try to import records from another GSRS system that had different Groups or even the same Group objects but saved in a different order. There wasn’t a way to tell what group with id 1 was to make sure it meant our own Group with id 1 versus a different Group 1.

The current model actually saves the Group access list as a serialized JSON stored as a String in a column in the database which is then uses a custom class to convert the serialized JSON into a List<Group> called GinasAccessConverter. In GSRS 2.x that uses EBean it uses a EBean Converter implementation and in GSRS 3 it extends Hibernate AttributeConverter interface.

Names

This section covers questions and design decisions regarding names.

## Why do Names have name and full name fields

**The Name entity has 3 fields that all have some form of the name as a String: name, fullName, stdName. Sometimes only 1 of those fields is set, other times 2 or 3 can be set and they are all different versions of the same name. Name and Fullname, in particular are worth describing.**

### Description

Name entity has a Name field that is only length 255 and a fullName that can be any length. Usually, only the name field is set, other times both the name field and fullName field is set where FullName is longer than name. When fullName is set, name is the first 255 characters or so.

### Rationale

The Name.name field maps to a database column that accidentally defaulted to 255 characters since there was no explicit JPA length annotation added to that field. This worked fine at first, so it went unnoticed until some exotic IUPAC names were added that went over that limit. While the field could have been extended to be up to 4000 characters (varchar), it was determined that some names really could be over even that limit and it should be allowed to be stored as a CLOB. Rather than convert or extend that default varchar name field, a new column Name.fullName was added, and would act for cases where the varchar column was not sufficient. FullName is a clob so it can be unlimited length, but it is slower to index database clob columns and we don’t usually need the long length anyway, so GSRS usually sets the name field unless the length is > 255, then it instead sets the fullName field and TRUNCATES the name to the first 255 bytes to set as the name field.

This setting the name vs fullName is done in prePersist and preUpdate hooks so the users of GSRS don’t have to worry about it and it is done automatically. While this could be simplified to instead *always* use a CLOB or *always* use a VARCHAR, the REST API responses do not reveal that there is a difference at all and no database migration/mutation/upgrade operations are needed as long as this structure is maintained.

## What is Name.name vs Name.stdName

**Name has a 3rd name field called stdName which is sometimes a differently formatted version of Name.name leading to confusion about what is the difference and which one should be used when reading the JSON record.**

### Description

Name.stdname is the standardized Name of the Name.name field. It should be the same name as Name.name except has a different character encoding or simpler formatting. For example, the FDA uses stdName to convert the name into all uppercase ASCII version of the name. So if there is a UTF8 name with an “α” character, the stdName will say “.ALPHA.” so that legacy systems that need to read in GSRS data as input that only understand ASCII can understand those kinds of names.

Some organizations also take advantage of this name standardization to input different kinds of marked-up in the name like HTML or LaTeX encoded names and let the stdName be the normal version of the name. As of this writing, there are several helper entity processors that can encode your names in different formats but it is hard to switch back and forth between encodings and it might be difficult to know what encoding each name used.

Other organizations have seen what the FDA does and there has been some confusion to which name field is the “real” name and which is the reformatted name. Sometimes they have been set “backwards” to the intent. This was confusion was largely due to deficits in the documentation. The REST API had not shown this field within the JSON by default until 2.7 when the “view=internal” version of the responses would show that field and the default UI would prioritize this view.

### Rationale

Legacy systems that don’t accept UTF8 needed a way to handle a simplified version of the name and at the same time we want modern systems that can handle UTF8 or other markup to be able to consume that instead. The UI looks much better being able to show HTML versions of the names with UTF8 characters especially for non-English characters.

# Molwitch

**GSRS uses an NCATS created library called Molwitch to abstract way which underlying cheminformatics library is actually used.**

### Description

GSRS computational software uses novel algorithms to analyze and process cheminformatics data, however the bedrock of those software tools needs to rely on an underlying cheminformatics library to manipulate chemical objects and read and write them out to various file formats. NCATS and the FDA have historically used a special commercially licensed version of software from Chemaxon. This is a problem for some users. NCATS software tools relying on commercial software is difficult to distribute since the commercial license cannot be shared, and other external groups are less likely to reuse our software, if it means they must also purchase a license from Chemaxon. Furthermore, since GSRS developers are funded by taxpayers, any work produced at NCATS or the FDA should be freely available and having such a dependency on Chemaxon’s JChem violates this principle.

There are other open-source cheminformatics libraries out there that perform similar tasks as JChem, but each one has its share of issues ranging from missing functionality, code verbosity, and memory and runtime performance differences. Switching to any of these other libraries would also break all GSRS code since the APIs are different and therefore would cause major portions of the codebase to be rewritten. Trying out several different third-party libraries would force even more software rewrites.

To address these problems, we developed an abstraction software layer, now known as Molwitch, that sits between the GSRS software and the cheminformatics library of choice. All cheminformatics calls are routed through this abstraction layer which will then delegate to the underlying cheminformatics library. This layer of abstraction removes any direct dependencies on commercial cheminformatics software like JChem and will allow the GSRS code to switch between different underlying cheminformatics libraries without recompiling any of the code.

### Rationale

Since GSRS software would have to be modified at least once to remove direct calls to Chemaxon’s JChem, it might as well switch to Molwitch so any future library changes wouldn’t require any additional code changes. Initially, only an old version JChem (3.17) from 2007 that NCATS used internally and CDK were supported. In 2018, Molwitch added support for the modern versions of JChem, (as of this writing 20.3.0). Preliminary work was also done to experiment with adding more cheminformatics libraries, but incompatible licensing issues and developer availability have put those modules on hold. The default version of GSRS now ships with CDK as the default cheminformatics library.

# Converting an Entity Object into a GSRS 3 Route

**GSRS JSON contain links back to the GSRS REST API to entities the way the route Strings are generated is done in an unusual way that can be quite confusing to follow and is radically different in GSRS 3 compared to GSRS 2. The Following sections describe some of the ways URLs are Generated**

## Why is there no code in Entity classes that directly link to Controller classes to get routes

**GSRS 3 entity classes do not directly reference Controller code or classes to the keep the entity Model free of any View or Controller code. However, GSRS 3 does have custom methods and annotations to give hints to the View and Controller code how to render itself.**

### Description

GSRS 3 Entity classes do not directly reference any GSRS Controller code. While Spring Boot does provide libraries such as HATEOS to generate URLs from Controller class methods we do not put any such code in the actual entity model classes. All that kind of code is kept in the View/Controller related classes and does not pollute the GSRS model classes.

GSRS entity classes may include multiple annotations including Jackson annotations for how to render entities and their child fields as JSON and some custom GSRS entities such as @GsrsApiAction and @EntityMapperOptions and some utility classes such as GsrsLinkUtil and FieldReference to provide additional hints and recipes for how to create the JSON record objects but there is no Spring View code mixed into the entity objects.

### Rationale

GSRS 2 did mix Controller reference code inside the entity classes and while that makes generating the JSON or creating routes and link outs to other entities easy it means you need the controller code to both compile or use the class elsewhere. This made sharing just our model code impossible because we had to lug around our entire JEE framework around with it just so you could write a class that could use the GSRS POJOs.

GSRS 3 is more modular with the entity classes completely separate in their own module entity-core. The Controller classes are in other modules that depend on the entity modules to help enforce to help enforce the rules that entities can’t link directly to Controllers.

However, this does lead to issues and complications to tell the code how to make the JSON for an entity and how to generate URLs for the entity JSON to link out to other entities or even itself. Spring provides lots of standard link generators and JSON generators out of the box but those do not match the home grown JSON that GSRS makes, and would not be backward compatible; so a lot of effort was put in to both make this unique GSRS JSON generation happen “auto-magically” yet be customizable so that particular classes that break the usual GSRS API mold can still match the API Contracts from GSRS 2.x so the UI and other API consumers don’t break.

## GsrsUnwrappedEntityModelProcessor and annotations @EntityMapperOptions and @GsrsApiAction

**GsrsUnwrappedEntityModelProcessor is the class that converts a GSRS entity into a HATEOS hypermedia object so that Spring can convert it to JSON with link-outs to other GSRS entities. GSRS provides special annotations to help tell the EntityModelProcessor how to convert the entity object into JSON.**

### Description

GSRSUnwrappedEntityModelProcessor is the glue code that converts GSRS entities into HATEOS RepresentationModelProcessor which is used by Spring to convert the returned Object into JSON with URLs. By default, the HATEOS object model would put links to entities in a “links” object which does not match what we need to keep backwards compatibility with older GSRS API Contracts. Therefore, we have to write a custom processor to generate link-outs the way GSRS needs them to be.

GSRS also has extra complications because we heavily rely on Jackson JSON View to include or exclude certain fields or methods and some fields or methods return URL links which we want serialized to JSON, sometimes in different ways!

@EntityMapperOptions has several fields that tell the EntityModelProcessor how to convert the Object into JSON including how to tell what the Id of an object is, what to do in the compact JSON view and how to render URLs.

@GsrsApiAction is a special annotation on methods that perform GSRS action functions that are meant for an API consumer to invoke. The GSRSApiAction annotation tells the EntityModelProcessor how to render this function as a link-out.

### Rationale

The GSRS API grew over several years and is not always consistent. Some entities have different ways the URLs link-outs are generated and if the link-outs include just the URL String or is an object with { “url” : “http//path”} or if they have additional fields inside that object and what they would be.

Some entity URLs are referenced by their IDs and some link out using a different field. Most entity classes have a corresponding Controller and some don’t but use another entity’s controller.