MXLIMS model overview

Table of Contents

Version 0.6.0

After some discussion (with thanks to Ed Daniel) the basic structure of the model has now settled and is no longer expected to change in major ways (though input is of course always welcome). As of 23 April this version includes the complete model specification and document description in the form of a series of JSON schemas. The full model documentation can be fond in the docs/html/ directory A reference Python implementation of the model, with some associated example files will be produced in due course.

Introduction

MXLIMS aims to make a precise, defined data model that allows you to store and transfer relevant metadata to go with the actual data produced. The initial scope is for macromolecular crystallography, but the approach is general and can be expanded as far as someone is willing to take it. The approach is that of MongoDB or metadata catalogues like ICAT and SciCat: a minimum of linked core object, which contain metadata to accommodate the infinite variety of specific data one might want to store. The metadata are modelled separately, so that all data are precisely defined; the support for site-specific data, the lack of inter-object links in the metadata and the versioning of the metadata schemas, allow you to make local model changes without unpredictable consequences.

One requirement of the model is to support a LIMS system, including provenance tracking. Another is to define an API supporting information passing between programs and sites, from experiment planning and input parameters, through instructions for acquisition queues, metadata for results, and input to subsequent processing steps. A third requirement is maintainability. A fully precise model would require individual tables for each separate kind of data, but the experience of ISPyB shows

that such models become complex, rigid, and hard to maintain, particularly when multiple sites need to make changes to support their specific needs. To mitigate this problem it is a core principle in MXLIMS have very few core classes, and to cut down the number of classes by using the same schemas for similar use cases, and for all stages of the experimental process from the initial diffraction plans to the final result. Additionally, MXLIMS has specific slots where you can store extra keyword-value parameters without abusing or breaking the main model.

The modelling is based on work, discussions and use cases from various people in the world of synchrotron crystallography over a number of years. Of particular note is ICAT (the Job class is a core ICAT class), mmCIF, Ed Daniels (Icebear) and Karl Levik (Diamond) for modelling of samples and shipping, Global Phasing and Olof Svensson (ESRF) for handling workflows and multi-sweep experiments, and Kate Smith and May Sharpe (SLS) for discussions on SSX use cases. Most recently the model has been modified to reflect input from the MXCuBE AbstractLIMS working group.

Core MXLIMS model

The core model is illustrated in Figure 1 (below)

The model is organised around four core abstract classes:

- PreparedSample, describing sample content and composition
- LogisticalSample, describing the objects that you ship, mount and investigate in experiments: Plates, Pucks, Wells, Drops, Crystals, ...
- Dataset, describing the actual data produced
- Job, describing the experiments and calculations that produce the data.

These four classes, together with the links between them, allow you to track the provenance and history of all data produced. Links between objects are stored using (in essence) foreign keys as in a database (`sampleId, inputDataIds`, …), pointing to the UUIDs of other objects. All model objects are subclasses of the four core classes; the actual data for each class of model object are defined separately.

The **MxlimsObject** is the superclass for all model objects. The `uuid` attribute gives an identifier to each object; the `version` determines which schema versions are used for the parameters (metadata). There are two slots for site-specific extensions, in order to allow flexible use without breaking the model. The `extensions` attribute is an unconstrained keyword:value dictionary. The use of this field is discouraged, but allows for adding extra information quickly without breaking or abusing the model. The `namespacedExtensions` field is intended for site-specific extensions that are defined by published, schemas. Namespaces could be e.g. ‘ESRF’ or ‘GPhL’ and the relevant schemas are to be maintained by the owner of the namespace.

The **PreparedSample** class describes the material of the sample, including contents and components, creation date, identifiers and batch numbers. The same PreparedSample can be used in several different LogisticalSamples, at several different levels (well, drop, location, crystal).

LogisticalSamples are organised as nested containers, e.g. Shipments containing Plates. containing Drops, containing Wells. The lowest level of the hierarchy would be the Crystal. In practice a loop or drop location may contain multiple crystals that can only be identified during the experiment, so the Crystal object would often not be generated before the actual experiment. Jobs (experiments) can be applied to several types of LogisticalSample, provided they correspond to a single PreparedSample, e.g. to Pins, Wells, Drops, or Crystals.

Jobs describe an experiment or calculation that can generate Datasets. Jobs can have inputs such as template data (for diffraction plans), reference data (e.g. reference mtz files), or plain input data (for processing jobs), and produce Dataset results. Jobs can be nested, so that one job (e.g. a workflow run) can start other jobs (e.g. X-ray centring, characterisation, or acquisition) if desired.

Datasets can have either a source (the Job that created them) or a derivedFrom link (but not both). The `Dataset.role` specifies the role that the Dataset has relative to the job that created it; this allows you to distinguish different types of output. Information sufficient to identify the location of data files must be given in the type-specific metadata.

It should be noted that these four abstract classes delimit the kind of objects that can be modelled. There is for instance no class that can correspond to a sample component as such, be it Lysozyme, dithiothreitol, or fetal calf serum, so we cannot have a single object with a `uuid` that we can refer to for sample components. Sample components have to be specified as part of samples, with some resulting duplication in giving the various identifiers, names and Smiles strings together every time.

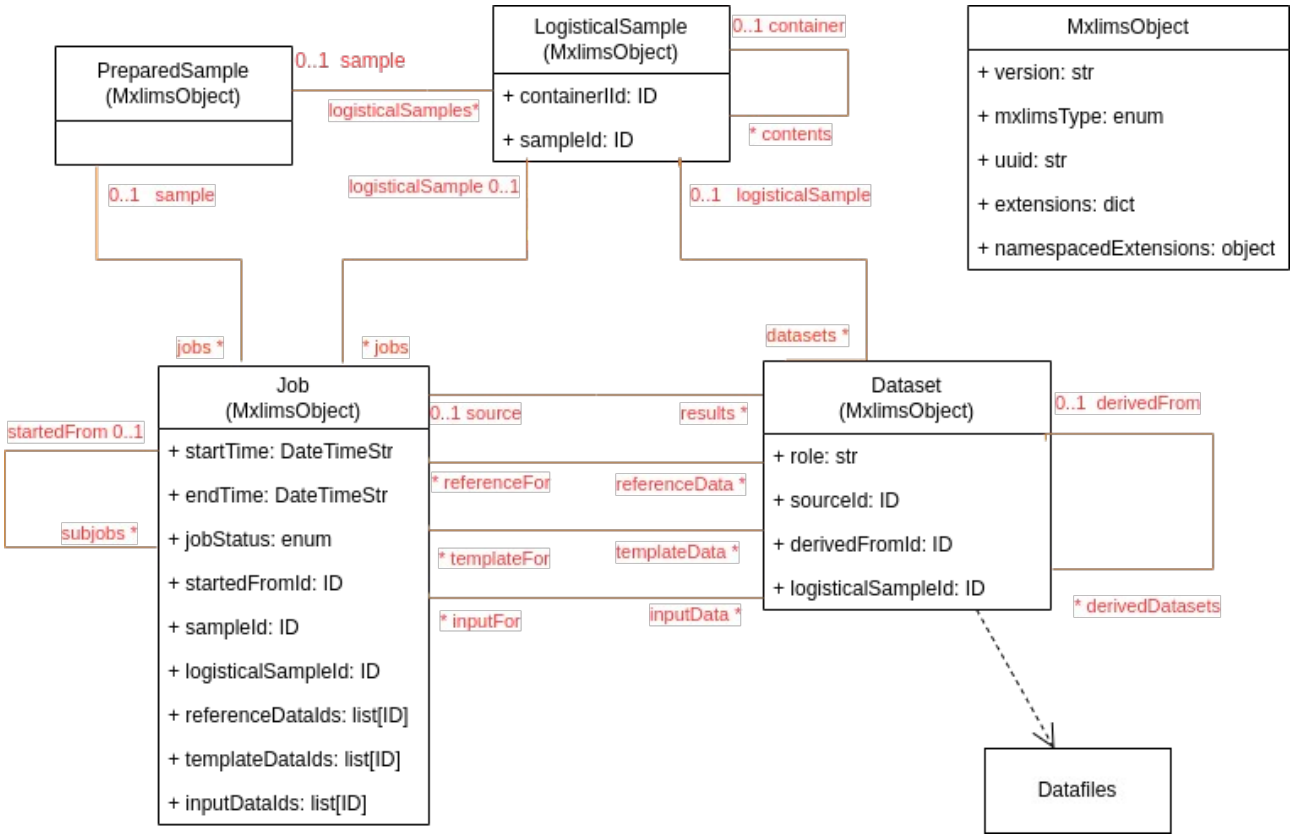

Figure 1 <u>The core model</u> The diagram shows the four core abstract classes, together with their attributes and the foreign key fields that are used to specify inter-object links. The orange lines show the inter-object links, all of which are two-way, together with the associated role names. These form part of the API defined by the model, but are not stored separately. Instead they must be derived by model implementations from the stored foreign keys.

Model Contents

The core abstract classes can be used to make an unlimited number of subclasses in order to model various scientific fields. At the moment the model is limited to macromolecular crystallography, including shipping, experiments, and processing. Each particular class has a defined set of attributes (‘metadata’), as well as limits on which types of objects it can be linked to, within the bounds of the inter-object links defined in the core. The model does not allow new inter-object links to be specified in the metadata. The classes currently supported in the model are shown in figures 2 and 3.

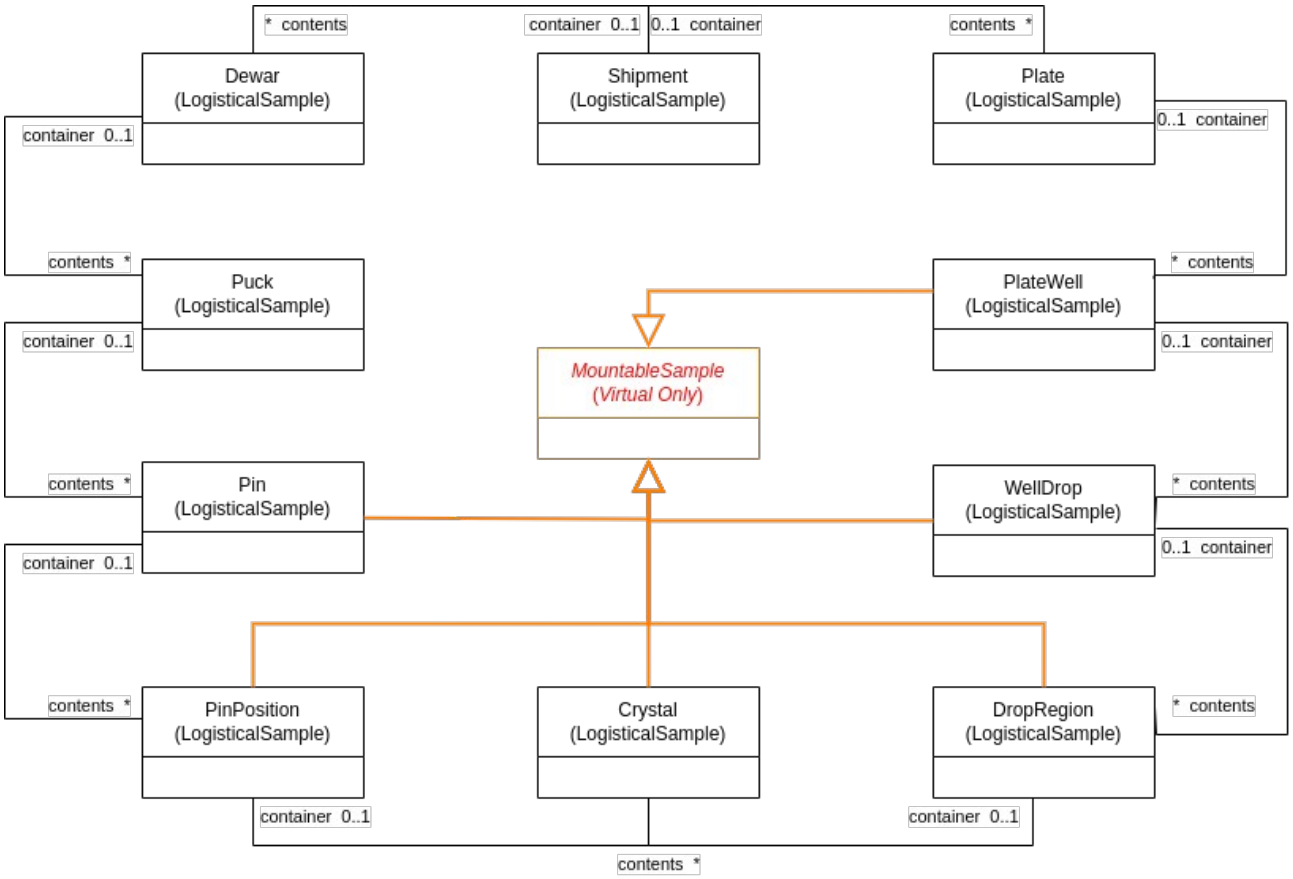

Figure 2. Logistical Samples. The figure shows the organisation of logistical samples (roughly: sample containers and crystals), with their container-content relationships. The '*MountableSample*' class is not actually a class in the model, but was added to make the diagrams easier to interpret. The classes shown (orange, broad-headed arrows) as extensions of 'MountableSample' (Pin, PinPosition, PlateWell, WellDrop, DropRegion and Crystal) are the logistical sample classes that can be used to perform a single experiment, and therefore those that can be linked to a PreparedSample, to Jobs, and to Datasets.

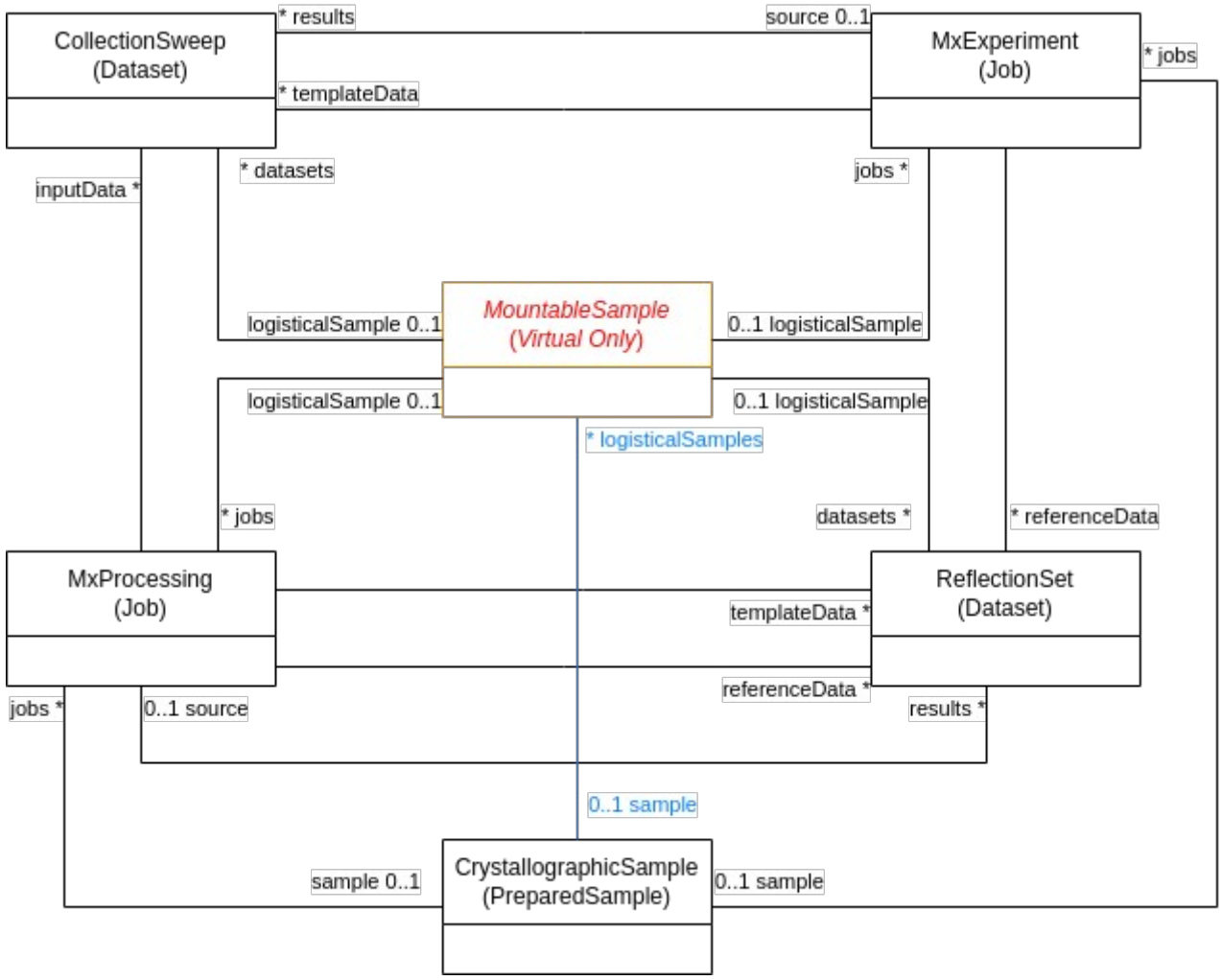

Figure 3, Model Details. This figure shows the model classes in the current model, apart from the logistical sample classes that were shown in Figure 2, as well as the links between them. The 'MountableSample' is not an actual class in the model but was added to the diagram to stand in for the logistical samples that share the links shown here, as described in Figure 2. Note that the MxExperiment class does not have any inputData link. The link between MountableSample and CrystallographicSample is no different from the other links in the diagram; it is shown in blue only to distinguish it graphically from the lines that cross it.

Model Implementation

Specification

The complete model specification is given as a series of JSON schemas. Each object (including the abstract classes) has one entry in the schemas/data/ directory, that specifies the data attributes. The schemas/datatypes/ directory contains enumerations (e.g. space group names) and data structures (e.g. Unit cells) that are used within actual objects. The schemas/references directory contains objects used to specify inter-object links in the model specification and within JSON messages. All

files in this directory can (and, in the future, will) be derived from the information in schemas/data/. The schemas/objects/ directory contains the completed model objects. This is a matter of combining the metadata specification (from schemas/data) and the core abstract class specification (from schemas/core) and adding constraints on the type of object that this object can be linked to.

Inter-object links

The handling of inter-object links in the specification is complex, since there have to be handled differently in the specification JSON schemas, the JSON data files, and the simple and database implementations. The diagrams above show the foreign-key IDs that are stored in implementations (with some differences between simple and database implementations), and the derived API fields that are used to access the links. The JSON schemas, on the other hand, follows the JSON files and represents links as a series of 'Ref' objects (e.g. DewarRef, MxExperimentRef, …). These have two fields, a fixed `mxlimsType`, and `ref`. The `mxlimsType` is optional in JSON files but defines the type of object linked to and so provides enough information to generate the implementations. The `ref` specifies a JSON reference composed of a regexp of the form "^/{mxlimsType}/{uuid}$". This matches the structure of the MxlimsMessage.json, which allows MxlimsMessage files to be validated against the schema so as to ensure that references are to objects defined within the document and of the correct type.

JSON files

The JSON representation of the model follows the JSON schemas (as indeed it should). This fulfils the key requirement that it should be possible to validate a JSON message against the model specification. The way multiple objects are combined into a single message is shown in the messages/MxlimsMessage.json file, that supports unlimited numbers of all currently defined objects.

It is a major problem for a JSON representation that JSON is limited to tree structures, whereas the current model describes an infinite object network. The same object could appear in multiple different contexts within the same file, forcing you to deal with either crosslinks or redundant copies of the data. Another problem is how to handle links to objects that are not part of a given message e.g. because you are referring to a sample composition that has been transferred previuosly and already resides within the destination LIMS system. The solution we have adopted is to avoid nesting JSON objects within each other and present them instead as a single top-level layer of objects, with a separate list for each object type. All inter-object links are treated as crosslinks, and are specified as JSON references, as described in the schemas/references/ directory. This makes the files somewhat less readable by humans (who are more at home with nested data structures than with networks of foreign keys) but allows you to validate that the object at the other end of a crosslink is indeed of the correct type and (depending on the message schema chosen) whether the

object must be included in the same message, or can be left as a `uuid` to be dereferenced at destination.

Simple python implementation

A prototype Python implementation to serve as a reference is still ‘in progress’. This would implement the core model and the API as shown in Figure 1. The current plan is to use a handwritten ‘database emulator’ with a few uuid:object dictionaries, combined with pydantic to store the modelled data. The major part of the model is specified in schemas/data and schemas/datatypes, and these parts can be converted directly into pydantic definitions using the pydantic datamodel_code_generator. The actual implementation would be put in the core abstract classes (schemas/core). These would have to be coded by hand, but, being independent of the model contents, that would require only a one-off effort. The final pydantic objects (as described in schemas/objects) do require some model-dependent code that is not supported by the pydantic code generation libraries, but it should be a simple matter to write code generation scripts that could take care of this. The final result will be a framework that can generated code for new parts of the model directly from the relevant JSON schema specification.

It should be noted that the behaviour of links upon deletion is implicit in the structure of the model. In the absence of mechanisms to ensure otherwise there are no cascading deletes, and foreign keys attributes (such as sourceId or sampleId) remain populated even if the objects they point to are deleted. The many-to-many links referenceData, inputData and templateData are stored on the Job side of the link, with the effect that these links are removed when the relevant Job is deleted.

Database implementation

Providing a full database implementation is outside the scope of the MXLIMS working group, but the MXLIMS model was designed specifically to work well with a database implementation using a system like MongoDB. The same API can be supported in a database implementation with only minimal modifications, as shown in figure 4.

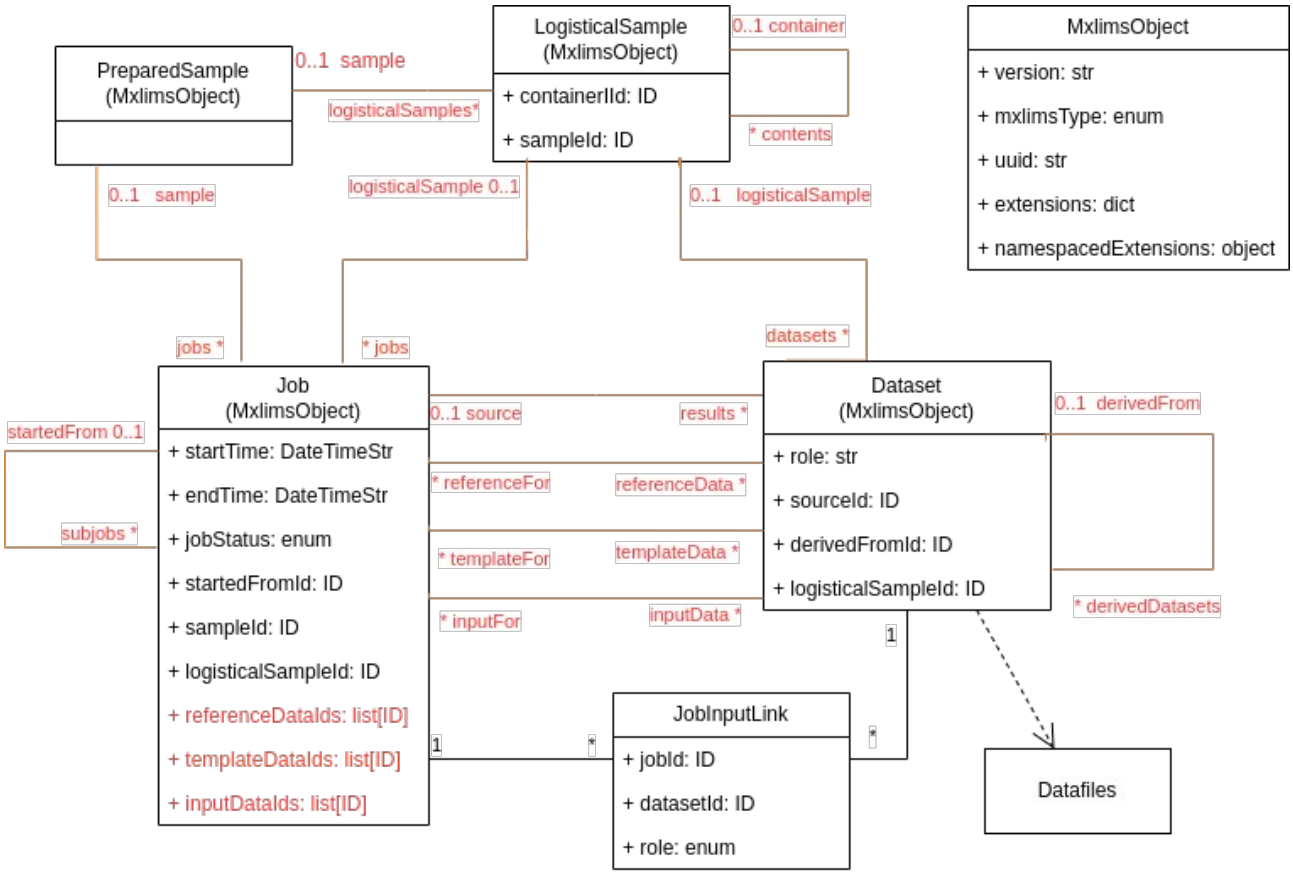

Figure 4, Database implementation The model can be implemented with one table for each core class, and with the same API as shown in figure 1, requiring only the additional JobInputLink table, in order to support the many-to-many links between the Job and Dataset classes. Element shown in orange and red are derived.

Since the model is already based on the use of foreign keys it translates seamlessly to a database implementation. The only difference regards the many-to-many links between the Job and Dataset classes (`inputData`, `templateData`, and `referenceData`), which are implemented using an additional table, JobInputLink. The attributes Job.referenceDataIds, Job.templateDataIds, and Job.inputDataIds are not part of the database implementation; to provide the same API as for other implementations these attributes must therefore be derived. The settings for cascading deletes (etc.) need to be be set appropriately in order to ensure the same behaviour as for different implementations (see preceding section).

Tools and dependencies

The model specification is written in JsonSchema. We use version 07 as this is the highest version supported by the documentation generator.

Html documentation is generated using json-schema-for-humans (https://coveooss.github.io/json-schema-for-humans/#/). The command used is (note the config file):

```
generate-schema-doc --config-file docs/schemadoc_config.json schemas docs/html
```

Conversion to pydantic is done in part with datamodel_code_generator (https://docs.pydantic.dev/latest/integrations/datamodel_code_generator/). In the longer run pydantic code generation is only relevant for the pydantic/data and pydantic/datatypes directories. The core classes must be implemented by hand, and the generation of code in pydantic/objects and pydantic/references (if required) should eventually be handled by bespoke code generation scripts/.

The command used is:

```
datamodel-codegen --input-file-type jsonschema --output-model-type pydantic_v2.BaseModel  --base-class mxlims.pydantic.MxBaseModel.BaseModel --use-schema-description --use-double-quotes --disable-timestamp --use-default --target-python-version 3.10 --snake-case-field --use-exact-imports --capitalise-enum-members --use-title-as-name --use-one-literal-as-default --input mxlims/schemas --output mxlims/pydantic
```

when run from the mxlims_data_model/ directory.

The resulting pydantic classes are subclassses of `mxlims.pydantic.MxBaseModel.BaseModel` which sets the necessary configueration

The JSON schema specification uses camelCase field names, which is the standard, while the pydantic classes are generated using snake_case field names, with the camelCase names as aliases. The correct export of pydantic data would therefore requires a command like

```
fp.write(mxrecord.model_dump_json(indent=4, by_alias=True, exclude_none=True)
```

In order to handle inter-object links correctly, it will be necessary to supplement the direct pydantic export commands with a certain amount of custom code.

The tools do impose some limitation on the modelling. Json-schema-for-humans seems to be able to handle the full JSON specification (up to version 07), but the pydantic generation does not support the full range of JSON schemas. There are some problems for complex logical constraints, which may indeed make more sense in the context of document validation than in the context of Python data storage classes. To obtain clearer pydantic, we have

- Modelled all enumerations as separate schemas rather than as part of field specifications.
- Modelled alternative variants as separate schemas and tried to avoid “oneOf” constraints, since these would anyway result in separate pydantic classes.

- Where “oneOf” constraints have been retained the pydantic model sometimes lack some of the model constraints, e.g. the constraint that a Dataset must contain *either* a `sourceId` *or* a `derivedFromId`. These cases must be handled with custom code.