How to write a YAML schema package

This guide explains how to write and upload NOMAD schema packages in the YAML format that can be uploaded as part of your data. This is a good way to start out experimenting with custom data structures in NOMAD, but for more advanced use cases you may need to use Python schema packages (../plugins/schema_packages.md). For more information on how an archive file is composed, visit Explanation > Data structure (../../explanation/data.md).

Example data

Let's assume we want to describe chemical compositions using the elements they contain.

The following structured data (in this example as a `.yaml` document) could describe the composition of water.

```yaml
{{ yaml_snippet('examples/docs/basic_schema/data.archive.yaml:data', '', 'm_def') }}
```

In structured data formats (such as `.yaml` or `.json`), data is put into combinations of *primitive values* (e.g. `'H2O'`, `1.141`), *objects* (a set of *keys* and *value* pairs, where *values* can be *objects*, *lists*, or *primitive values*), and *lists* of *values*.

Sections

In a schema package, we want to describe the structure of data, i.e. what are the allowed combinations of *objects*, *lists*, and *primitive values*.

The crucial task here is to define what *keys* certain *types of objects* can have and what possible *values* might exist for each of these keys.

In NOMAD, we call *objects* **sections** and we define *types of objects* with **section

definitions**. Since *objects* can be nested, **sections** become like the sections and subsections of a book or paper. Sections are a representation of data and they are the building blocks for **archives** (../../reference/glossary.md#archive). Section definitions form a schema package and they are
the building blocks for the **metainfo** (../../reference/glossary.md#metainfo).

In the above example, we have two *types* of *objects*: an overaching object for the entire structure
(with *keys* for `composition` and `elements`), and an additional object which describes the internal structure of
`elements` (with *keys* for `label`, `density`, and `isotopes`). Let's start with
the *definition* for elements. This is what the *section definition* looks like in NOMAD's yaml-based schema package format:

```yaml
Element:
{{
yaml_snippet('examples/docs/basic_schema/package.archive.yaml:definitions/sections/
Element', '  ') }}
```

A *section definition* provides all the available *keys* for a *section* that instantiates this *definition*. For each *key*, e.g. `label`, `density`, `isotopes`, it provides
more information on the possible values.

Let's have a look at the overall definition for our chemical composition:

```yaml
Composition:
{{
yaml_snippet('examples/docs/basic_schema/package.archive.yaml:definitions/sections/
```

Composition', ' ') }}

```

Again, all possible *keys* (`composition` and `elements`) are defined. But now we see that there are two different types of *keys*, **quantities** and **subsections**. We say that *section definitions* can have **properties** (e.g. the *keys* they define) and there are two distinct types of *properties*.

Quantities

*Quantities* define possible *primitive values*. The basic properties that go into a *quantity definition* are:

- **type**: what kind of *primitive value* can be used, e.g. `str` or `np.float64`
- **shape**: what is the shape of the value, e.g. scalar or list (`['*']`)
- **unit**: what is the physical meaning of the value

The *names* of *quantity definitions* serve as the *key*, used in respective *section objects*.

Type

This is a list of supported quantity types.

|type|description|
|-|-|
|`string`||
|`str`||
|`float`||
|`integer`||
|`int`||
|`boolean`||
|`bool`||
|`np.int32`|Numpy based integer with 32 bits.|
```

|`np.int64`|Numpy based integer with 64 bits.|

|`np.float32`|Numpy based float with 32 bits.|

|`np.float64`|Numpy based float with 64 bits.|

|`Datetime`||

|`User`|A type for NOMAD users as values.|

|`Author`|A complex type for author information.|

|`{type_kind: Enum, type_data: []}`|Use `type_data` to specify enum values as list of strings.|

|**|To define a quantity that is a reference to a specific section.|

Shape

The shape of a quantity is a list of *dimensions*, where each *dimension* defines the possible size of that *dimension*. The empty list (or no shape) describes a scalar value, a list with one *dimension* a list or vector, a list with two *dimensions* a matrix, etc.

Dimensions can be given as:

- an integer number to define a fixed size, e.g. a 3x3 matrix would have shape `[3, 3]`.

- the string `'*'` to denote am arbitrary sized dimension, e.g. a list quantity would have shape `['*']`.

- A string that describes the name of a sibling quantity with an integer type, e.g. `['number_of_atoms', 3]`

Unit

NOMAD manages units and data with units via the Pint (https://pint.readthedocs.io/en/stable/) Python package. A unit is given as a string that is parsed by pint. These strings can

be simple units (or their aliases) or complex expressions. Here are a few examples: `m`, `meter`, `mm`, `millimeter`, `m/s`, `m/s**2`.

While you can use all kinds of units in your uploaded schema packages, the built-in

NOMAD schema (Metainfo) uses only SI units.

Subsections

Subsections define a *part-of-relationship* between two *sections*. *Subsection definitions* are *properties* of the parent *section definition* and name a child *section definition*. In the data, we can now contain instances of the target (e.g. `Element`) in instances of the source (e.g. `Composition`). A *subsection* can be defined as *repeating* to allow many child *sections* of the same *type*. In our example,

one `Composition` can contain many `Elements`.

The *names* of *subsection definitions* serve as the *key*, used in respective *section objects*.

Uploading schema packages

NOMAD archive files allow you to upload data in NOMAD's native file format. An archive file can be a .yaml or .json file. It ends with `.archive.json` or `.archive.yaml`.

Archive files are mainly used to convey data. Since YAML schema packages are also "just" data, archive

files can also be used to convey a schema package.

You can upload schema packages and data in separate files.

`schema_package.archive.yaml`

```yaml
--8<-- "examples/docs/basic_schema/package.archive.yaml"
```

and `data.archive.yaml`

```yaml
--8<-- "examples/docs/basic_schema/data.archive.yaml"
```

Or, you can upload the schema package and data in the same file:

```yaml
--8<-- "examples/docs/basic_schema/package.archive.yaml"
data:
  m_def: Composition
{{ yaml_snippet('examples/docs/basic_schema/data.archive.yaml:data', '  ', 'm_def') }}
```

References

Reference quantities

We already saw that we can define a *part-of* relationship between sections. When we want to represent highly inter-linked data, this is often insufficient. *References* allow us to create a more lose relationship between sections.

A reference is a uni-directional link between a *source* section and a *target* section. References can be defined in a schema package as a quantity in the *source* section definition that uses the *target* section definition as a type.

Instead of connecting the elements in a composition with subsections, we can also connect a composition section to elements with a quantity:

```yaml
Composition:
{{
yaml_snippet('examples/docs/references/single.archive.yaml:/definitions/sections/Composition', '  ') }}
```

Here, `type: Element` refers to the section definition `Element`, very similar to `section: Element` in a subsection definition.

We saw above that subsections are represented as nested *objects* in data (forcing a *part-of* relationship). References are represented as string-typed *primitive values* in serialized data. Here is an example `Composition` with references to elements:

```yaml
{{   yaml_snippet('examples/docs/references/single.archive.yaml:/data/compositions/0', '') }}
```

These string-references determine the *target* section's place in the same archive. Each `/`-separated segment represents a *key*. A reference starts from the root *object* and following the sequence of *keys* to a specific *object* (i.e. section).

Here is the full archive data:

```yaml
data:
{{ yaml_snippet('examples/docs/references/single.archive.yaml:/data', ' ', 'm_def') }}
```

If you follow the *keys* `data`, `periodic_table`, `elements`, `0`, you reach the section that represent hydrogen. Keep in mind that *lists* use index-numbers as *keys*.

Schema package references

References can look different depending on the context. Above we saw simple references

that point from one data section to another. But, you also already a saw a different type of reference. Schema packages themselves contain references: when we used `type: Element` or `section: Element` to refer to a *section definition*, we were writing down references that point to a *section definition*. Here we can use a convenience representation: `Element` simply replaces the otherwise cryptic

`#/definitions/sections/0`.

So far, we never discussed the use of `m_def`. In the examples you might have seen this

as a special *key* in some objects. Whenever we cannot determine the *section definition*

for a *section* by its context (e.g. the *key*/*subsection* used to contain it in a *parent section*), we use `m_def` to provide a reference to the *section definition*.

Different forms of references

Depending on where references are used, they might take a different serialized form. Here are a few examples for different reference syntax:

|Example reference|Comments|

|---|---|

|`#/data/periodic_table/elements/0`|Reference to a section within the subsection hierarchy of the same archive.|

|`Element`|Reference to a *section definition* in the same archive. Can only be used to target *section definitions*.|

|`nomad.datamodel.metainfo.workflow`|Reference to a *section definition* that was written in Python and is part of the NOMAD code. Can only be used to target *section definitions*.|

|`../upload/raw/data.archive.yaml#/data`|Reference to a section in a different `.archive.yaml` file of the same upload.|

|`../upload/archive/mainfile/data.archive.yaml#/data`|Reference to a section in a processed archive given by entry *mainfile*.|

|`../upload/archive/zxhS43h2kqHsVDqMboiP9cULrS_v#/data`|Reference to a section in a processed archive given by entry-id.|

|`../uploads/zxhS43h2kqHsVDqMboiP9cULrS_v/raw/data.archive.yaml#/data`|Reference

to a section in an entry of a different upload.|

|`https://mylab.eu/oasis/api/v1/uploads/zxhS43h2kqHsVDqMboiP9cULrS_v/raw/data.archive.yaml#/data`|Reference to a section in an entry in a different NOMAD installation.|

References across entries

A references in the archive of one entry can point to a section in a different entry's archive. The following two example files, exemplify this use of reference between two NOMAD entries.

periodic_table.archive.yaml

```yaml
--8<-- "examples/docs/references/periodic_table.archive.yaml"
```

composition.archive.yaml

```yaml
--8<-- "examples/docs/references/composition.archive.yaml"
```

These inter-entry references have two parts: `#`, where *entry* is a path or URL denoting the *target* entry and *section* a path within the *target* entry's subsection containment hierarchy.

Please note that also schema packages can be spread over multiple files. In the above example,

one file contained the schema package and data for a periodic table and another file contained

schema package and data for the composition of water (using the periodic table).

Base sections and inheritance

We add a relationship between *section definitions* that allows us to create more *specialized* definitions from more *abstract* definitions. Here the *properties* of the *abstract* definition are inherited by the more *specialized definitions*

Base sections

Here is a simple schema package with two *specialization* of the same *abstract* section definition:

```yaml
definitions:
{{ yaml_snippet('examples/docs/inheritance/basic.archive.yaml:/definitions', '  ') }}
```

The two *specialized* definitions `Annealing` and `Evaporation` define the *abstract* definition `Process` via the `base_section` property. With this `Annealing` and `Evaporation` inherit the quantity `time`. We do not need to repeat quantities from the base section, and we can add more properties. Here is an example `Evaporation` using both the inherited and added quantity:

```yaml
{{ yaml_snippet('examples/docs/inheritance/basic.archive.yaml', '', 'definitions') }}
```

Polymorphy

What happens if we reference *abstract* definitions in subsections or reference quantities?

Here is an subsection example. In one schema, we define the relationship between `Sample` and `Process`. In another schema, we want to add more *specializations* to what a process is.

abstract.archive.yaml

```yaml
--8<-- "examples/docs/inheritance/abstract.archive.yaml"
```

specialized.archive.yaml

```yaml
{{ yaml_snippet('examples/docs/inheritance/specialized.archive.yaml', '', 'data') }}
```

The *section definition* use in the subsection `processes` defines what a contained section has to be "at least". Meaning that any section based on a *specialization* of `Process` would be a valid `processes` subsection.

specialized.archive.yaml

```yaml
definitions:
  # see above
data:
{{ yaml_snippet('examples/docs/inheritance/specialized.archive.yaml:data', '  ') }}
```

The fact that a subsection or reference target can have different "forms" (i.e. based on different *specializations*) is called *polymorphism* in object-oriented data modelling.

Pre-defined sections

NOMAD provides a series of built-in *section definitions*. For example, there is `EntryArchive`, a definition for the top-level object in all NOMAD archives (e.g. `.archive.yaml` files). Here is a simplified except of the *main* NOMAD schema `nomad.datamodel`:

```yaml
EntryArchive:
  sub_sections:
    metadata:
      section: EntryMetadata
    definitions:
      section: nomad.metainfo.Package
    data:
      section: EntryData
    # ... many more
EntryData:
  # empty
```

Compare this to the previous examples: we used the top-level *keys* `definitions` and `data` without really explaining why. Here you can see why. The `EntryArchive` *property* `definitions` allows us to put a *schema package* into our archives. And the `EntryArchive` *property* `data` allows us to put *data* into archives that is a *specialization* of `Schema`. The `Schema` definition is empty. It is merely an *abstract* placeholder that allows you to add *specialized* data sections to your archive. Therefore, all *section definitions* that define a top-level data section, should correctly use `nomad.datamodel.Schema` as a base section. This would be the first "correct" example:

```yaml
--8<-- "examples/docs/inheritance/hello.archive.yaml"
```

Here are a few other built-in section definitions and packages of definitions:

Section definition or package	Purpose
nomad.datamodel.EntryArchive	Used for the root object of all NOMAD entries
nomad.datamodel.EntryMetadata	Used to add standard NOMAD metadata such as ids, upload, processing, or author information to entries.
nomad.datamodel.EntryData	An abstract section definition for the `data` section.
nomad.datamodel.ArchiveSection	Allows to put `normalize` functions into your section definitions.
nomad.datamodel.metainfo.eln.*	A package of section definitions to inherit commonly used quantities for ELNs. These quantities are indexed and allow specialization to utilize the NOMAD search.
nomad.parsing.tabular.TableData	Allows to inherit parsing of references .csv and .xls files.
nomad.datamodel.metainfo.workflow.*	A package of section definitions use by NOMAD to define workflows
nomad.metainfo.*	A package that contains all *definitions* of *definitions*, e.g. NOMAD's "schema language". Here you find *definitions* for what a sections, quantity, subsections, etc. is.

Separating data and schema package

As we saw above, a NOMAD entry can contain schema package `definitions` and `data` at the
same time. To organize your schema package and data efficiently, it is often necessary to re-use
schema packages and certain data in other entries. You can use *references* to spread your
schema packages and data over multiple entries and connect the pieces via

references.

Here is a simple schema package, stored in a NOMAD entry with mainfile name `package.archive.yaml`:

```yaml
--8<-- "examples/docs/references/multiple_files/package.archive.yaml"
```

Now, we can re-use this schema package in many entries via *references*. Here, we extend

a schema contained in the package and instantiate definitions is a separate mainfile `data-and-package.archive.yaml`:

```yaml
--8<-- "examples/docs/references/multiple_files/data-and-package.archive.yaml"
```

Here is a last example that re-uses the schema and references data from the two entries

above:

```yaml
--8<-- "examples/docs/references/multiple_files/package.archive.yaml"
```

!!! warning "Attention"

    You cannot create definitions that lead to circular loading of `*.archive.yaml` files.

    Each `definitions` section in an NOMAD entry represents a *schema package*. Each *schema package* needs to be fully loaded and analyzed before it can be used by other *schema packages* in other entries. Therefore, two *schema packages* in two entries cannot reference each other.

Conventions

Conventions for labels

When assigning labels within your codebase, it's essential to follow consistent naming conventions for clarity and maintainability. The following guidelines outline the conventions for labeling different elements:

- **Sections**: Labels for sections should adhere to Python convention of CapitalizedCamelCase.

This means that each word in the label should begin with a capital letter, and there should be

no spaces between words. For example: `SectionLabelOne`, `SectionLabelTwo`.

- **Quantities and Subsections**: Labels for quantities and subsections should be in lower_case. This convention involves writing all lowercase letters and separating words with whitespace. Abbreviations within these labels may be capitalized to enhance scientific readability. For example: `quantity label`, `subsection label`, `IV label`.