Schema annotations

Definitions in a schema can have annotations. These annotations provide additional information that NOMAD can use to alter its behavior around these definitions. Annotations are named blocks of key-value pairs:

```yaml
definitions:
  sections:
    MyAnnotatedSection:
      m_annotations:
        annotation_name:
          key1: value
          key2: value
```

Many annotations control the representation of data in the GUI. This can be for plots or data entry/editing capabilities.

{{ pydantic_model('nomad.datamodel.metainfo.annotations.ELNAnnotation', heading='## ELN annotations') }}

{{ pydantic_model('nomad.datamodel.metainfo.annotations.BrowserAnnotation', heading='## Browser') }}

Display annotations

{{ pydantic_model('nomad.datamodel.metainfo.annotations.QuantityDisplayAnnotation', heading='### Display annotation for quantities') }}

{{ pydantic_model('nomad.datamodel.metainfo.annotations.SectionDisplayAnnotation', heading='### Display annotation for sections') }}

`label_quantity`

This annotation goes in the section that we want to be filled with tabular data, not in the single quantities.

It is used to give a name to the instances that might be created by the parser. If it is not provided, the name of the section itself will be used as name.

Many times it is useful because, i. e., one might want to create a bundle of instances of, say, a "Substrate" class, each instance filename not being "Substrate_1", "Substrate_2", etc., but being named after a quantity contained in the class that is, for example, the specific ID of that sample.

```yaml
MySection:
  more:
    label_quantity: my_quantity
  quantities:
    my_quantity:
      type: np.float64
      shape: ['*']
      description: "my quantity to be filled from the tabular data file"
      unit: K
      m_annotations:
        tabular:
          name: "Sheet1/my header"
        plot:
          x: timestamp
          y: ./my_quantity
```

!!! important

The quantity designated as `label_quantity` should not be an array but a integer, float or string, to be set as the name of a file. If an array quantity is chosen, the parser would fall back to the use of the section as name.

Tabular data

{{ pydantic_model('nomad.datamodel.metainfo.annotations.TabularAnnotation', heading='### `tabular`') }}

Each and every quantity to be filled with data from tabular data files should be annotated as the following example.

A practical example is provided in How To (../howto/customization/tabular.md#preparing-the-tabular-data-file) section.

```yaml
my_quantity:
  type: np.float64
  shape: ['*']
  description: "my quantity to be filled from the tabular data file"
  unit: K
  m_annotations:
    tabular:
      name: "Sheet1/my header"
    plot:
      x: timestamp
      y: ./my_quantity
```

`tabular_parser`

One special quantity will be dedicated to host the tabular data file. In the following examples it is called `data_file`, it contains the `tabular_parser` annotation, as shown

below.

{{ pydantic_model('nomad.datamodel.metainfo.annotations.TabularParserAnnotation', heading = '') }}

Available Combinations

Tutorial ref.	`file_mode`	`mapping_mode`	`sections`	How to ref.
1	`current_entry`	`column`	`root`	HowTo (../howto/customization/tabular.md#1-column-mode-current-entry-parse-to-root)
2	`current_entry`	`column`	my path	HowTo (../howto/customization/tabular.md#2-column-mode-current-entry-parse-to-my-path)
np1	`current_entry`	`row`	`root`	Not possible
3	`current_entry`	`row`	my path	HowTo (../howto/customization/tabular.md#3-row-mode-current-entry-parse-to-my-path)
np2	`single_new_entry`	`column`	`root`	Not possible
4	`single_new_entry`	`column`	my path	HowTo (../howto/customization/tabular.md#4-column-mode-single-new-entry-parse-to-my-path)
np3	`single_new_entry`	`row`	`root`	Not possible
5	`single_new_entry`	`row`	my path	HowTo (../howto/customization/tabular.md#5-row-mode-single-new-entry-parse-to-my-path)
np4	`multiple_new_entries`	`column`	`root`	Not possible
np5	`multiple_new_entries`	`column`	my path	Not possible
6	`multiple_new_entries`	`row`	`root`	HowTo (../howto/customization/tabular.md#6-row-mode-multiple-new-entries-parse-to-root)
7	`multiple_new_entries`	`row`	my path	HowTo (../howto/customization/tabular.md#7-row-mode-multiple-new-entries-parse-to-my-path

)|

```yaml
data_file:
  type: str
  description: "the tabular data file containing data"
  m_annotations:
    tabular_parser:
      parsing_options:
        comment: '#'
      mapping_options:
      - mapping_mode: column
        file_mode: single_new_entry
        sections:
        - my_section/my_quantity
```

<!-- The available options are:

|**name**|**type**|**description**|

|---|---|---|

|`parsing_options`|group of options|some pandas `Dataframe` options.|

|`mapping_options`|list of groups of options|they allow to choose among all the possible modes of parsing data from the spreadsheet file to the NOMAD archive file. Each group of options can be repeated in a list. | -->

Plot

The PlotSection base section serves as an additional functionality to your sections. This base section is designed to simplify the process of creating various types of plots, making it easy to use Plotly Express, Plotly Subplot, and the general Plotly graph

objects.

Features:

- Plotly Express: Create simple and quick plots with a high-level, expressive API.

- Plotly Subplot: Organize multiple plots into subplots for more complex visualizations.

- General Plotly Graph Objects: Fine-tune your plots by working directly with Plotly's graph objects.

Usage:

- Inherit from this base section to leverage its plot functionality.

- Customize your plots using the annotations plotly-express, plotly-subplots, or/and plotly-graph-object.

The PlotSection class makes it possible to define plots that are shown alongside your data.

Underneath, we use the Plotly Open Source Graphing Libraries to control the creation of the plots,

and you can find many useful examples in their documentation.

In Python schemas, the PlotSection class gives you full freedom to define plots programmatically.

For example, you could use plotly.express and plotly.graph_objs to define plots like this:

```python
from nomad.datamodel.metainfo.plot import PlotSection, PlotlyFigure
from nomad.datamodel.data import EntryData
import plotly.express as px
import plotly.graph_objs as go
from plotly.subplots import make_subplots
class CustomSection(PlotSection, EntryData):
    m_def = Section()
```

```
                time = Quantity(type=float, shape=['*'], unit='s',
a_eln=dict(component='NumberEditQuantity'))
        substrate_temperature = Quantity(type=float, shape=['*'], unit='K',
a_eln=dict(component='NumberEditQuantity'))
        chamber_pressure = Quantity(type=float, shape=['*'], unit='Pa',
a_eln=dict(component='NumberEditQuantity'))
    def normalize(self, archive, logger):
        super(CustomSection, self).normalize(archive, logger)
        first_line = px.scatter(x=self.time, y=self.substrate_temperature)
        second_line = px.scatter(x=self.time, y=self.chamber_pressure)
        figure1 = make_subplots(rows=1, cols=2, shared_yaxes=True)
        figure1.add_trace(first_line.data[0], row=1, col=1)
        figure1.add_trace(second_line.data[0], row=1, col=2)
            figure1.update_layout(height=400, width=716, title_text="Creating Subplots in
Plotly")
        self.figures.append(PlotlyFigure(label='figure 1', figure=figure1.to_plotly_json()))
            figure2 = px.scatter(x=self.substrate_temperature, y=self.chamber_pressure,
color=self.chamber_pressure, title="Chamber as a function of Temperature")
                self.figures.append(PlotlyFigure(label='figure 2', index=1,
figure=figure2.to_plotly_json()))
        heatmap_data = [[None, None, None, 12, 13, 14, 15, 16],
            [None, 1, None, 11, None, None, None, 17],
            [None, 2, 6, 7, None, None, None, 18],
            [None, 3, None, 8, None, None, None, 19],
            [5, 4, 10, 9, None, None, None, 20],
            [None, None, None, 27, None, None, None, 21],
```

[None, None, None, 26, 25, 24, 23, 22]]

heatmap = go.Heatmap(z=heatmap_data, showscale=False, connectgaps=True, zsmooth='best')

figure3 = go.Figure(data=heatmap)

figure_json = figure3.to_plotly_json()

figure_json['config'] = {'staticPlot': True}

self.figures.append(PlotlyFigure(label='figure 3', index=0, figure=figure_json)

```

To customize the plot configuration in python one can add the config to the generated json by to_plotly_json().

```

figure_json['config'] = {'staticPlot': True}

```

In YAML schemas, plots can be defined by using the PlotSection as a base class,
and additionally utilizing different flavours of plot annotations. The different annotation options are described below.

{{ pydantic_model('nomad.datamodel.metainfo.annotations.PlotlyGraphObjectAnnotation', heading='### PlotlyGraphObjectAnnotation') }}

{{ pydantic_model('nomad.datamodel.metainfo.annotations.PlotlyExpressAnnotation', heading='### PlotlyExpressAnnotation') }}

{{ pydantic_model('nomad.datamodel.metainfo.annotations.PlotlySubplotsAnnotation', heading='### PlotlySubplotsAnnotation') }}

plot annotations in python

For simple plots in Python schema one could use the annotations without normalizer:

```python

```
from nomad.datamodel.metainfo.plot import PlotSection
from nomad.metainfo import Quantity, Section
from nomad.datamodel.data import EntryData
class CustomSection(PlotSection, EntryData):
    m_def = Section(
        a_plotly_graph_object=[
            {
                'label': 'graph object 1',
                'data': {'x': '#time', 'y': '#chamber_pressure'},
                'layout': {
                    'title': {
                        'text': 'Plot in section level'
                    },
                    'xaxis': {
                        'title': {
                            'text': 'x data'
                        }
                    },
                    'yaxis': {
                        'title': {
                            'text': 'y data'
                        }
                    }
                }
            }, {
                'label': 'graph object 2',
```

```
        'data': {'x': '#time', 'y': '#substrate_temperature'}
    }
],
a_plotly_express={
    'label': 'fig 2',
    'index': 2,
    'method': 'scatter',
    'x': '#substrate_temperature',
    'y': '#chamber_pressure',
    'color': '#chamber_pressure'
},
a_plotly_subplots={
    'label': 'fig 1',
    'index': 1,
    'parameters': {'rows': 2, 'cols': 2},
    'layout': {
        'title': {
            'text': 'All plots'
        }
    },
    'plotly_express': [
        {
            'method': 'scatter',
            'x': '#time',
            'y': '#chamber_pressure',
            'color': '#chamber_pressure'
```

```
            },
            {
                'method': 'scatter',
                'x': '#time',
                'y': '#substrate_temperature',
                'color': '#substrate_temperature'
            },
            {
                'method': 'scatter',
                'x': '#substrate_temperature',
                'y': '#chamber_pressure',
                'color': '#chamber_pressure'
            },
            {
                'method': 'scatter',
                'x': '#substrate_temperature',
                'y': '#chamber_pressure',
                'color': '#substrate_temperature'
            }
        ]
    }
)
```

```
time = Quantity(type=float, shape=['*'], unit='s', a_eln=dict(component='NumberEditQuantity'))
substrate_temperature = Quantity(type=float, shape=['*'], unit='K', a_eln=dict(component='NumberEditQuantity'))
```

chamber_pressure = Quantity(type=float, shape=['*'], unit='Pa', a_eln=dict(component='NumberEditQuantity'))

```

{{ pydantic_model('nomad.datamodel.metainfo.annotations.PlotAnnotation', heading='### PlotAnnotation (Deprecated)') }}
```