ezEML Developer Notes- Metapype

ezEML uses the Metapype library, also developed by EDI, to manage the EML metadata model being worked on. A Metapype model is a directed acyclic graph with nodes corresponding to EML elements. Nodes are represented by Metapype's `Node` class. A node object has attributes:

```
self._id = str(uuid.uuid1()) if id is None else id
self._name = name
self._parent = parent
self._content = None if content is None else str(content)
self._tail = None
self._attributes = {}
self._nsmap = {}
self._prefix = None
self._extras = {}
self._children = []
```

`_name` here refers to the name of the element in the EML schema, e.g., "title", "dataTable", etc. I.e., it's the type of element, not a unique identifier for the individual node in question. The `_id` attribute provides the unique identifier.

Nodes are stored in a class-level dictionary called `store`, indexed by `_id`.

Metapype provides a variety of methods for finding nodes. Of particular interest are these:

```
get_node_instance(cls, id: str)

find_child(self, child_name)

find_all_children(self, child_name)

find_descendant(self, descendant_name)

find_single_node_by_path(self, path: list)

find_all_nodes_by_path(self, path: list)
```

The Metapype library also provides methods for reading and writing JSON and XML files representing an EML model and methods to evaluate/validate a model for its adherence to the portion of the EML standard implemented in Metapype.

Metapype node-handling methods are generic. I.e., they don't assume that the model represents EML metadata.

Metapype also contains code that is specific to EML. The relevant source code is in the `/src/metapype/eml` directory.

rules.json and rule.py

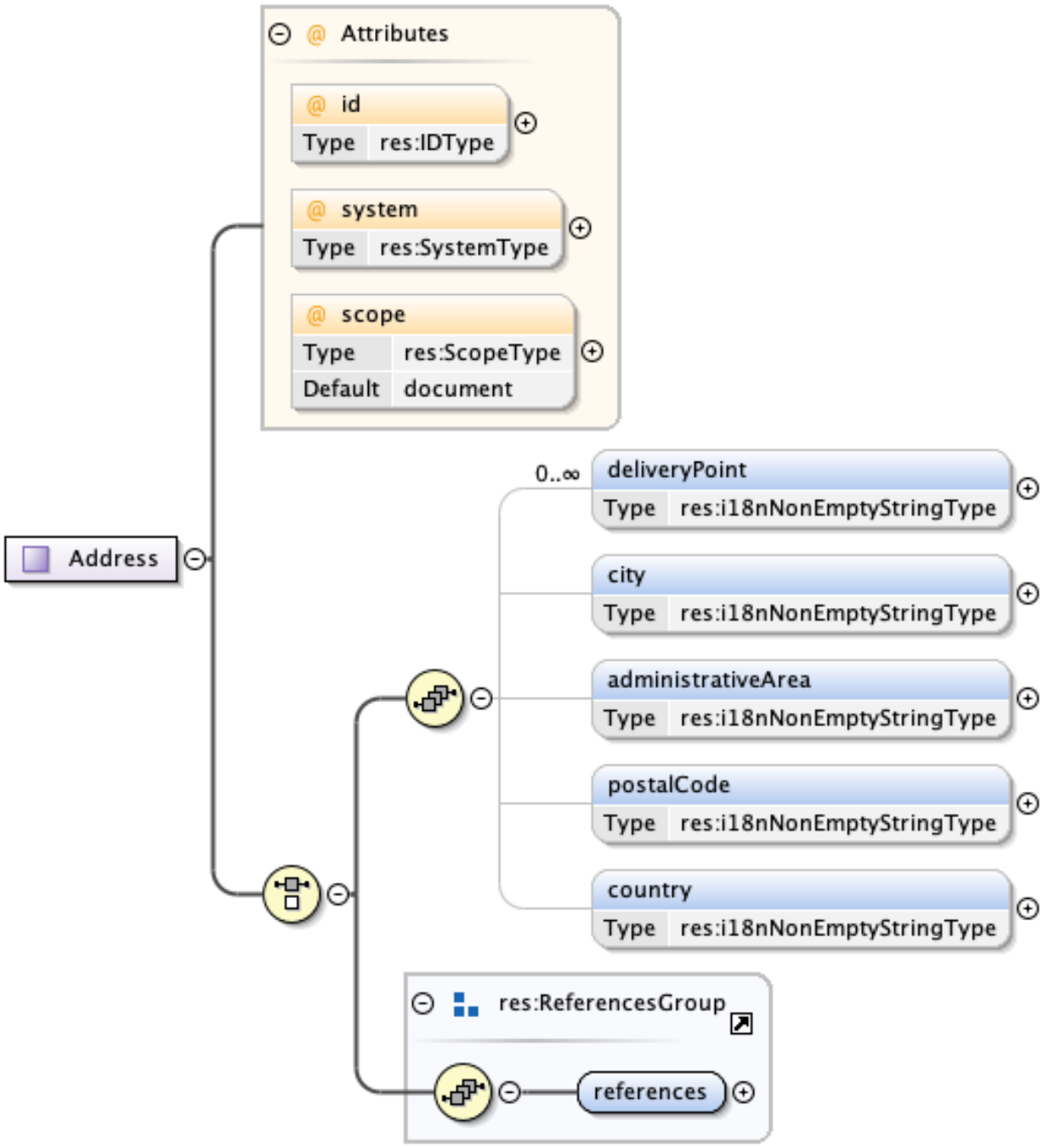
The EML schema is represented in JSON form in the file rules.json. EML schema rules are represented by the `class Rule`, defined in rule.py. The rules are loaded from the rules.json file via function `load_rules()` in rule.py.

Here is a typical rule in rules.json:

```
"addressRule" : [
    {
        "id": [false],
        "system": [false],
        "scope": [false, "document", "system"]
    },
    [
        ["deliveryPoint", 0, null],
        ["city", 0, 1],
        ["administrativeArea", 0, 1],
        ["postalCode", 0, 1],
        ["country", 0, 1]
    ],
    {
        "content_rules" : ["emptyContent"]
    }
],
```

`addressRule` here is the name by which the rule is indexed in the rules dictionary. This particular example is the rule for the Address type in the EML schema.

The EML schema for Address is shown below:



The three items in the JSON list for a rule correspond to its [attributes, children, content].

attributes is a dictionary that lists the attributes of the EML element. For each attribute, the first entry is a Boolean that indicates if the attribute is required. Remaining entries, if any, list the allowed values for the attribute. In the example above, `scope` has allowed values `document` and `system`.

children lists the rules that are children of the rule. For each child, its minimum and maximum cardinality is shown. `deliveryPoint`, for example, has minimum cardinality 0 (i.e., it is not required to be present) and maximum cardinality `null`, which stands for “infinity”.

content contains information about what type of content is allowed. In the `addressRule` example we see `"content_rules" : ["emptyContent"]`, meaning that an Address node has empty content (its content is contained in its children).

How ezEml Uses Metapype

The Metapype model for a data package is stored in the file system as a JSON file. Metapype provides methods for reading/writing a model in JSON or XML form.

The HTTP protocol is stateless, but ezEML needs to preserve state for a given user and data package across web requests. It does so via the JSON file produced by Metapype. The JSON file is read in at the start of a web access and saved at the end if it has changed. It is then available to be loaded in by the next web access that comes along referencing that data package.

Metapype gives each EML element (node, in Metapype parlance) a unique ID. These are saved and restored as the JSON file is saved and restored. This has the effect of providing a persistent storage with persistent node IDs, so the node IDs can be used to reference particular nodes across web accesses. The various ezEML pages for selecting items from a list, for example, identify the selected item via its node ID, and because the model will be read in when the next web access is handled, the selected node can be found via its ID in the node store.