OSCAL Community Contribution Series:
A Developer’s view of OSCAL

Experiences and recommendations for implementing OSCAL Libraries

Credentive

Agenda

- User’s view vs. Developer’s view
- What is an OSCAL library for?
- Types of Objects in Oscal-Pydantic
- Oscal-Pydantic Class Diagram overview
- Recommendations for implementing OSCAL in any (Object Oriented) language

User’s view versus Developer’s view

UX (User Experience) creates a pleasant experience for non-technical folks trying to solve business problems.

DX (Developer Experience) creates a pleasant experience for technical folks trying to develop tooling to solve user problems.

the the

User

problem interface

fights not

the the

Developer

problem API

What are the features of a good API?

- Easy to do the most common things
- The right way is the easiest way
- Idiomatic
- Features are discoverable
- Features are implemented consistently
- Minimize boilerplate (DRY)

What is an OSCAL API for?

What problem is an OSCAL API solving?

- Reduce cognitive burden of maintaining mental copies of the OSCAL specification
- Easily focus on the part of the specification that you need for the problem at hand
- Produces OSCAL data that is valid and well formed.
- OSCAL imported from external sources is presented idiomatically
- Users can easily extend the specification with minimal impact on compatibility.

Machine Generated Code: Pros and Cons

Machine Generated Code	Handwritten Code
✓ Tracks the specification closely	X Implements as much of the spec as the author needs
✓ Can always be up to date	X Updated occasionally
✓ Can implement multiple versions of the specification	X Implements the versions the author is interested in
X Limited to Core Specification	✓ Can be designed for extensibility
X Limited by the quality of the inputs	✓ Not dependent on limitations or structure of underlying specifications
X Produces verbose, difficult code	✓ Code can be carefully tuned for readability

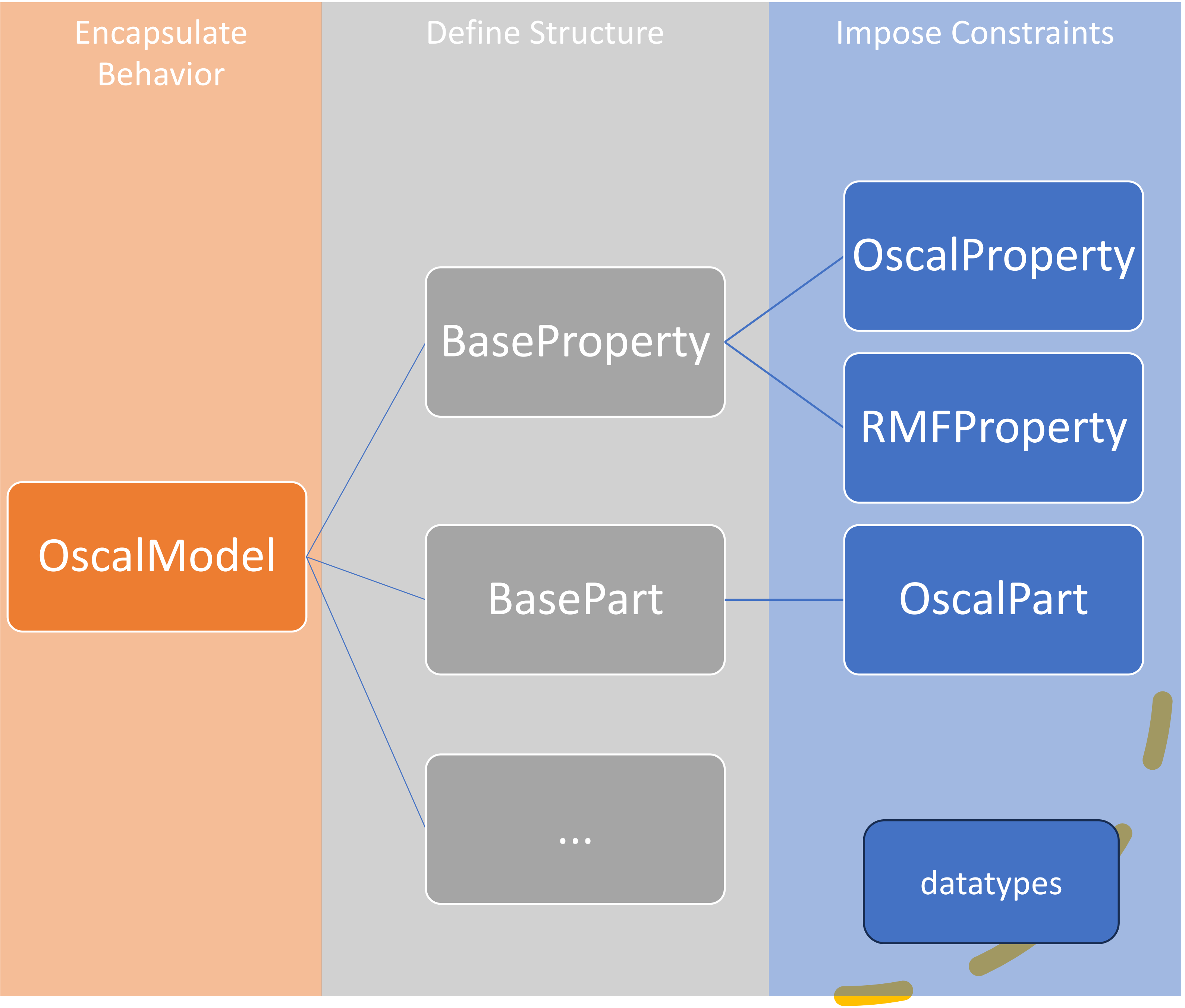
Oscal-Pydantic

What is it?

A pythonic, type-hinted library for importing or generating OSCAL documents.

- Implemented in a “dataclass” like format, using Pydantic
- Hand-written to optimize the developer experience for encoding and extending OSCAL.

Oscal-Pydantic Overview



OscalModel: Encapsulate Shared Behavior

- Encapsulates core validation Logic
- Defines basic behaviors common to all models
  - How to export JSON
  - How to translate attribute value names
- Most objects extend OscalModel directly

Datatypes

- Defines basic constraints for metaschema datatypes

Structure Models: Define Common Structures

- List Attributes of a Model
- Define basic type information

```
class BaseProperty(base.OscalModel):
# NOTE: This generic Property class should be extended to
provide constraints on value name
  name: datatypes.OscalToken = Field()
  uuid: datatypes.OscalUUID | None = Field(default=None)
  ns: datatypes.OscalUri = Field(
    default=datatypes.OscalUri(
      "http://csrc.nist.gov/ns/oscal"
    )
  )
  value: datatypes.OscalString = Field()
  prop_class: datatypes.OscalToken | None = Field(default=None)
  remarks: datatypes.OscalMarkupMultiline | None =
Field(default=None)
```

Impose Constraints

- Cardinality Constrainst
- Value Constraints
- “Type” constraints
- Deprecation Constraints
- Uniqueness Constraints

Cardinality Constraints

- How many attributes can appear
  - Usually 0..1, 1, or 0..inf

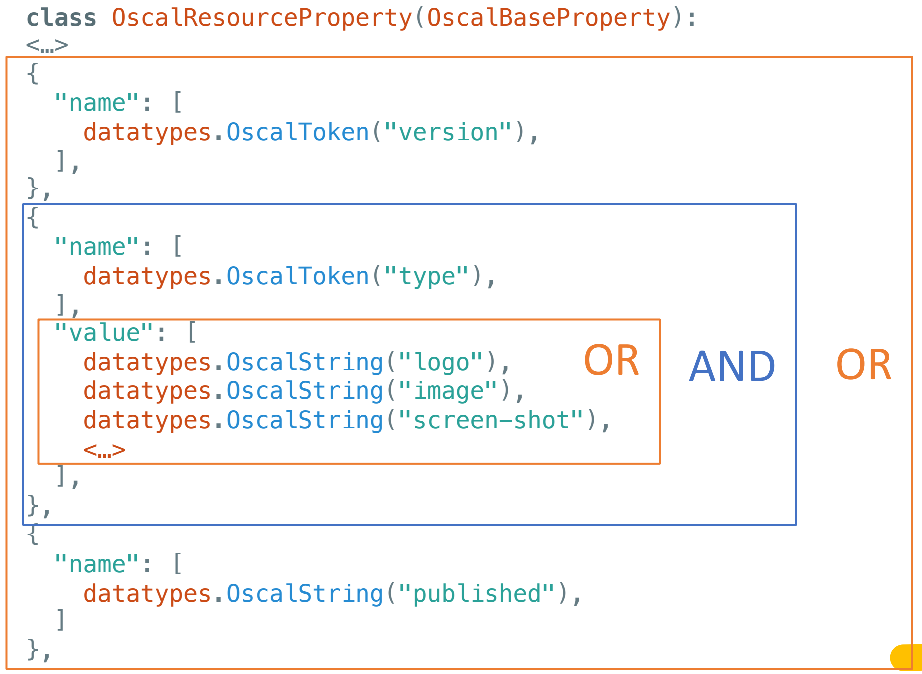
```
# Cardinality 1
uuid: datatypes.OscalUUID

# Cardinality 0..1
description: datatypes.OscalMarkupMultiline | None

# Cardinality 0..inf
props: list[properties.BaseProperty] | None
```

Value Constraints

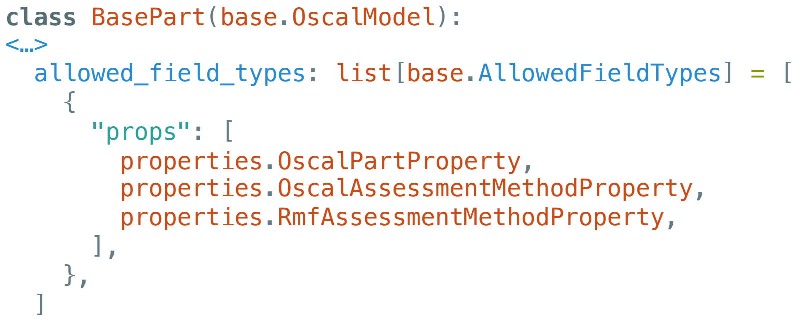
- Possible values of attributes
- Expressed as single values or sets of values
- Multiple Constraints can be expressed
- Within a constraint - AND
- Between constraints – OR
- Watch for “may be locally defined”, or allow-other="yes"



```
class OscalResourceProperty(OscalBaseProperty):
<…>
{
  "name": [
    datatypes.OscalToken("version"),
  ],
},
{
  "name": [
    datatypes.OscalToken("type"),
  ],
  "value": [
    datatypes.OscalString("logo"),
    datatypes.OscalString("image"),
    datatypes.OscalString("screen-shot"),
    <…>
  ],
},
{
  "name": [
    datatypes.OscalString("published"),
  ]
},
```

“Type” Constraints

- NOTE: Types do not exist in the Metaschema specification
  - A “type” encapsulates a specific set of value constraints
- Define Possible types of attributes



```
class BasePart(base.OscalModel):
<…>
  allowed_field_types: list[base.AllowedFieldTypes] = [
    {
      "props": [
        properties.OscalPartProperty,
        properties.OscalAssessmentMethodProperty,
        properties.RmfAssessmentMethodProperty,
      ],
    },
  ]
```

Deprecation Constraint

- Raise a warning in the event of a valid but deprecated value

```
class OscalControlProperty(OscalBaseProperty):
<…>
  def capitalized_withdrawn_deprecated(cls, value: str) -> datatypes.OscalToken:
    # raise a deprecationwarning if value is capitalized
    if type(value) == str and value == ”Withdrawn":
      warnings.warn(
        ”’Withdrawn' is a deprecated property value for Control. Use ‘withdrawn' instead",
     DeprecationWarning,
    )
    return value
```

Uniqueness Constraint

- Prevent duplicate values in some cases

```
class Resource(base.OscalModel):
<…>
  def unique_rlink(self) -> Resource:
    if self.rlinks is not None:
      links_counter = Counter([rlink.href for rlink in self.rlinks])
      duplicates = [item for item, count in links_counter.items() if count > 1]
      if len(duplicates) > 0:
        raise ValueError("Duplicate rlinks in %s: %s", self.uuid, duplicates)
    return self
```

Special Constraints

- Some kinds of data have unique constraints
  - Example: Hash
    - algorithm + value
    - value must be valid for the algorithm

```
if self.algorithm == "SHA-224" or self.algorithm == "SHA3-224":
    if len(self.value) == 28 and self.value_is_hex():
        # value is okay
    else:
        raise ValueError("Hash value length or contents do not match algorithm")
```

Summary

Recommendations

- Identify the key features of the API
  - Extensibility?
- Look at the Metaschema or official documentation
  - JSON Schema is incomplete!
- Don’t be afraid to start by hand coding