### Ontology Builder: from OpenAPI to OWL

Below we describe the mapping from OpenAPI constructs to their OWL counterparts.

It is important to note that the expressive power of both languages differs, and some concepts from OpenAPI cannot be expressed in OWL. Such concepts are translated into OWL annotations. These do not have semantical interpretation in OWL, and they serve as descriptions.

**Primitive Data Types.** OpenAPI[[1](https://swagger.io/specification/#data-type-format)] defines a list of supported primitive, as most appear in the OWL[[2](https://www.w3.org/TR/xmlschema-2/#schema)] specification the mapping is immediate. The only exception is the password format which is used as hint to UI that the value should be obscured. We encode this information using an ontology annotation.

|  |  |  |  |
| --- | --- | --- | --- |
| OpenAPI | OpenAPI - Format | OWL2 (note: below xsd types) | comments |
| integer | signed 32 bits | xsd: short |  |
| integer | signed 64 bits | xsd:long |  |
| number | float | xsd:float |  |
| number | double | xsd:double |  |
| string |  | xsd:string |  |
| string | byte | xsd:string | add rdfs:comment “byte” |
| string | binary | xsd:string | add rdfs:comment “binary” |
| boolean |  | xsd:boolean |  |
| string | date - RFC3339 | xsd:date | Simple conversion as both are based on [ISO8601](https://xml2rfc.tools.ietf.org/public/rfc/html/rfc3339.html#ISO8601) |
| string | date-time - RFC3339 | xsd:dateTime | Simple conversion as both are based on [ISO8601](https://xml2rfc.tools.ietf.org/public/rfc/html/rfc3339.html#ISO8601) |
| string | password | string | add rdfs:comment “password” |

**Schema Object.** A schema object allows the definition of data types. These types can be objects, but also primitives and arrays. As already stated, we focus on objects that are explicitly defined under the [definitions](https://json-schema.org/understanding-json-schema/structuring.html#defs) block of the specification. The URI of these objects is derived from the absolute-relative path of the file that contains them, and their name is defined as a key in the definitions block. In addition, the definition of each schema object may include metadata (such as, title and description), its type, its properties, and relations to other schema objects. A property definition must be a schema object (or a reference to such).

In addition, OpenAPI supports polymorphism, which can be expressed using allOf [[1](https://datatracker.ietf.org/doc/html/draft-fge-json-schema-validation-00#page-15)]. While the first is expressible in OWL, the other cannot be, and are therefore encoded using annotations. Additionally, all [metadata keywords](https://datatracker.ietf.org/doc/html/draft-fge-json-schema-validation-00#page-17) are all encoded as annotations.

The following tables details the translation of Schema Objects to OWL. We focus on explicit types, i.e., types that appear directly under the definition block.

The encoding of schema objects to OWL depends on the schema type. Schemas of types: null, number, string, and boolean are encoded as OWL datatypes; while schemas of type object and array are encoded as OWL classes. From here on, we refer to array and object as complex types, and to all other types as simple types.

The encoding of a simple type is as follows. First, a new OWL datatype is created, with a similar name and URI. Then, the corresponding OWL primitive type (see table $above) is defined, and range restrictions are added according to the table below. Finally, metadata is added as annotations. The table below provides the complete description:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | OpenAPI properties | OpenAPI Example | OWL2 | OWL 2  Example |
| Metadata | description, title, format, default |  | xsd:annotation |  |
| Numerical constraint | multipleOf,  maximum,  exclusiveMaximum,  minimum,  exclusiveMinimum | { ‘maximum’ : 5, ‘exclusiveMaximum’: true } | Numerical datatype range restrictions | xsd:int[>=5] |
| String constraints | maxLength,  minLength,  pattern[regular expression, according to the Ecma-262 Edition 5.1 regular expression dialect],  enum | { ‘maxLength’: 3 } | String datatype range restrictions; and enumeration of literals | xsd:string[length>5]; |

Below, we describe the encoding of schema object and array types. These objects are recursive in nature, as they hold schema objects. For simplicity, we assume bounded nesting, i.e., we restrict the object to either hold JSON schemas of simple types (i.e., inline), or to reference other schemas. In other words, we restrict implicit definitions of non-simple types within these objects. While may seem restrictive, it is possible to mitigate this. By defining the

The encoding of complex type is as follows. First, a new OWL class is created, with a similar name and URI. For array types, we create an ‘items’ property, while for object types, we use the names of the properties defined in the objects. A property is defined an OWL Data Property, iff, includes an inline definition of a simple type, or defined as an OWL Object Property otherwise. In addition, we encode size restrictions of these as OWL cardinality constraint.

Finally, metadata is added as annotations. The table below provides the complete description.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | OpenAPI properties | OpenAPI Example | OWL2 | OWL 2  Example |
| Metadata | description, title, format, default | {  "type": "array",  "description": "The name of the feature." } | annotation rdfs:comment, rdfs:label, rdfs:isDefinedBy, rdfs:comment resp. | AnnotationAssertion( rdfs:comment Type\_URI “The name of the feature." ) |
| Properties | properties | {  "type": "object",  "properties": {  "name": {  "type": "string",  "description": "The name of the feature."  },  "value": {  "type": "string",  "description": "The corresponding value for the feature."  }  },  "description": "Specifies additional capabilities supported by the image" } | Encoded references as object properties with corresponding names, and as data otherwise | TODO: either protégé fig or xml format |
| Items | items | {  "type": "array",  "items": {  "$ref": "#/definitions/VirtualMachineImageFeature"  } } | Encoded references as object properties,and as data otherwise | TODO: either protégé fig or xml format |
| Properties | maxProperties,  minProperties , AdditionalProperties | {  "type": "object",  "properties": {  "name": {},  "gender": {},  }  "maxProperties": 2,  "required": [‘gender']  } | No equivalent concepts, encoded as annotation  rdfs:comment |  |
| required | Encode all properties in the required set using some cardinality constraint |  |
| Array size constraints | minItems,  maxItems,  uniqueItems | {  "type": "array",  "maxItems": 3  } | min, max, rdfs:comment resp. |  |
|  |  |  |  |  |

properties, their description, their encoding to OWL, and a comment when the translation is not immediate. We use *so* to denote the schema object, *pn* to denote a property name under the ‘properties’ property, and *t* to denote its type.

We assume that properties that reference a JSON Schema object of type ‘object’ use a reference to reference it. In other words, we assume that no nesting exists. If this is not true, it could easily be resolved by redefining the inline object outside of the properties block.