CEDAR Template Model V1.6.0

Martin J O'Connor, Marcos Martínez-Romero,
and the CEDAR Team

August 10th, 2020

Table of Contents

Introduction

One of the main goals of the CEDAR project is to build an infrastructure for the creation and storage of machine-readable *metadata templates*. Metadata templates provide detailed definitions of the metadata that describes a particular data resource.

In CEDAR, the metadata template describes both the structure and the semantics of that metadata. The CEDAR system uses metadata templates to create *metadata instances*, which describe specific instances of data resources. Users typically generate these metadata instances to annotate their data.

This document describes the metadata template model developed for the CEDAR project. This model provides a detailed specification for the representation of metadata template and metadata template instances.

We first developed a template model to specify the key aspects of template construction [EKAW2016]. This model represents the core structural characteristics of templates—the common entities and compositional patterns that define a template. We then produced a concrete representation of the template model, emphasizing the addition of semantic markup and constraints. The concrete template model provides a consistent, interoperable information framework for defining templates and for creating and filling out metadata instances that correspond to those templates. Finally, we developed a set of tools for creating metadata templates and for acquiring metadata to generate metadata instances.

Template Model

Our system aims to recursively compose templates from existing, more granular templates. In our model, we term these sub-templates *template elements*. Template elements constitute the building blocks of metadata templates. Template elements may contain one or more atomic pieces of information, such a text or date field, or may be recursively composed from other template elements. *Template fields* are used to represent these atomic pieces of metadata. For example, a template field could be used to indicate the date at which a measurement was made for a particular scientific experiment. Template elements are used to recursively combine template fields or template elements to create more complex descriptions. For example, template fields “Phone” and “Email” could be contained in a template element called “Contact Information”, which could itself be contained in a template element called “Person”.

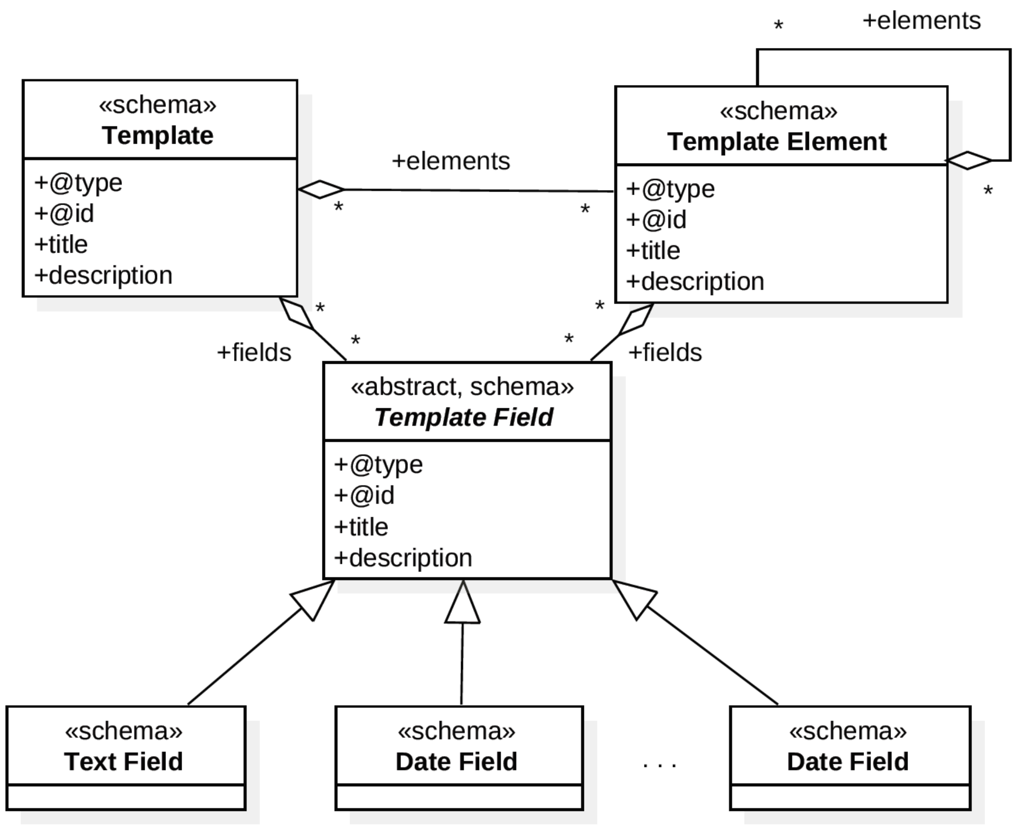

Figure 1. Schema Level of CEDAR's Template Model

Figure 1 presents a basic overview of the schema level of the CEDAR template model. The `Template`, `Template Element`, and `Template Field` entities represent their namesake concepts. All entities have an `@type` field and are uniquely identifiable via an `@id` field. They also contain `title` and `description` fields. A variety of built-in template field types are provided. These include a `Text Field`, which represents a free text field, and a `List Field`, which represent a multiple-choice element. This set can be extended to incorporate additional field types. Both templates and template elements can optionally have fields or elements nested inside them. Template elements and fields can be grouped together in a `Template` to provide an overall description of a collection of metadata.

The template model also defines instances derived from templates, which we refer to as *template instances*. Template instances are created from template specifications. A template effectively serves as a schema specification for metadata instances conforming to that template. Figure 2 presents the instance level of the CEDAR template model. As with schema-level templates and elements, all template and element instances have an `@id` field, which uniquely identifies the instance. Template and element instances can also contain an optional `@type` field, which can contain one or more URIs that provide type information for the associated instance. Template field instances can have three subtypes: (1) literal fields, which contain literal values;

(2) IRI fields, which contain IRI values, and (3) multiple value fields, which contain arrays of either literals or IRIs. Literal fields contain a literal value object that contains an `@value` field and an optional `@type` field. The `@value` field contains the raw literal value and the `@type` field contains one more datatypes for that literal (e,g., `http://www.w3.org/2001/XMLSchema#integer`). IRI value fields contain an IRI value object. In this case, an `@id` field is used to store the IRI value; the `@type` field can be used to optionally provide one or more types for that IRI.

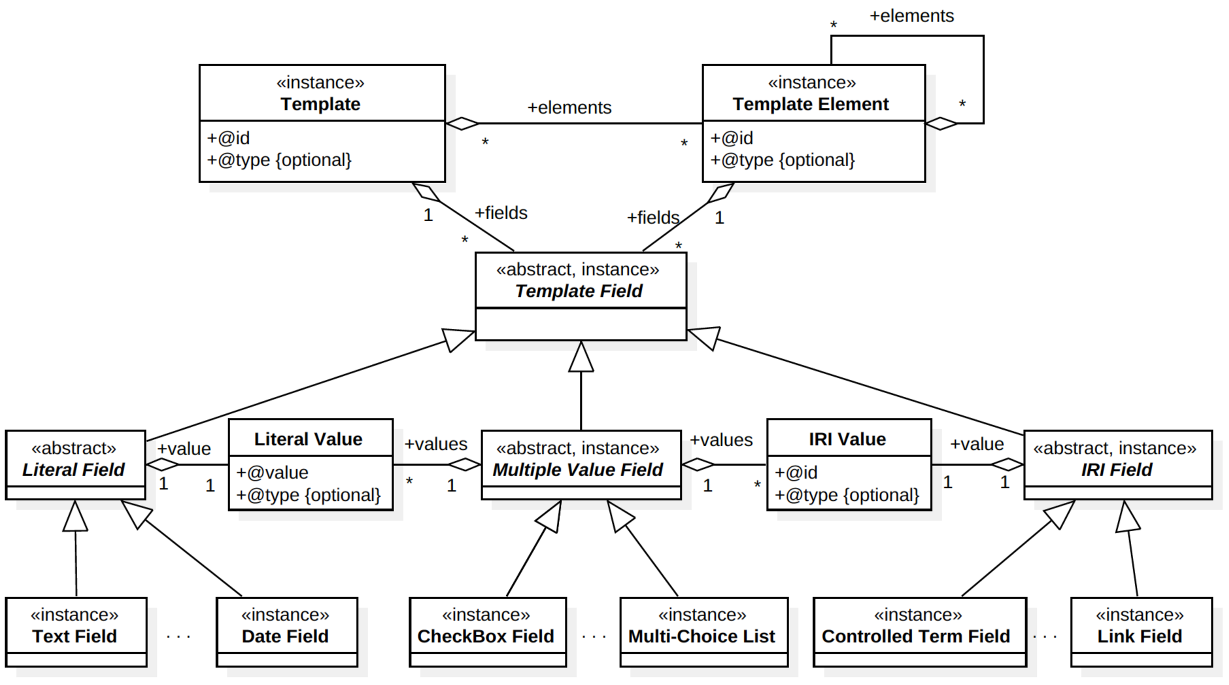

Figure 2. Instance Level of CEDAR's Template Model

The overall model provides an abstract structural specification of templates and instances. In the next section we will outline the development of a concrete representation of this model.

Template Model Concrete Representation

The template model requires a machine-interpretable representation for software systems to work with the model programmatically[1]. This representation must meet a variety of goals. Primarily, it must describe the structure of templates and the instances generated from these templates. It must also describe and constrain the various relationships between the entities in the model. Template representations must be conveniently serializable so that they can be provided via REST APIs and persisted to storage media. Ideally, the representation should be based on standard formats so that existing tools can be used to manage model entities. The representation should also permit easy validation, and easy indexing to support search. To

[1] The primary CEDAR REST APIs can be used to work with resources described using this model. An introduction to these APIs can be found here.

enable interoperation with controlled terms, a standardized means to annotate templates with controlled terms is key. Finally, the template format must interoperate with Linked Open Data technologies such as RDF and OWL, and allow metadata to be represented as RDF graphs.

We identified two key JSON-based technologies can be combined to meet many of the goals outlined above—while retaining full interoperation with semantic resources: JSON Schema[2] [JSON SCHEMA], and JSON-LD [JSON-LD][3]. Both are supported by a large variety of Web-centric tools.

JSON Schema is a technology for describing and validating the structure of JSON data. Its directives—themselves represented as standard JSON elements—can be used to provide a structural description of any JSON document. JSON documents that are specified with JSON Schema can be structurally validated against their associated schemas via off-the-shelf tools. JSON Schema provides a structural specification only—it does not describe the semantics of JSON documents. A recent technology called JSON-LD ("Linked Data") was developed to meet this goal. JSON-LD provides a lightweight syntax to add semantic annotations to JSON documents. The key goals of JSON-LD are to support the use of Linked Data in Web-based programming environments, to build interoperable Web services, and to store Linked Data in JSON-based storage engines. JSON-LD effectively allows JSON documents and their contents to be made available as Linked Data, offering the potential for machine-interpretable RDF semantics.

We first outline how we use JSON Schema to describe the structure of templates, template elements, and template fields and to constrain and validate the instances generated from those templates. We then show how we use JSON-LD to mark up these specifications, adding semantic content to the generated instances. We show how this combination of JSON Schema and JSON-LD provides the capabilities to fully represent the template model and to build a strong bridge to semantic technologies.

Throughout this document we will use the term *artifact* to refer generically to model entities—templates, elements, fields, and template instances. We will use the term *schema artifact* to refer to artifacts that contain structural specifications—templates, elements, and fields—and instance artifact to refer to entities generated from these specifications—currently, just template instances, though standalone element and field instances are not precluded by the model.

[2] See Appendix A for an overview of JSON Schema.
[3] See Appendix B for an overview of JSON-LD.

Representing Artifact Provenance

The CEDAR model defines provenance fields for artifacts. At present, seven core provenance fields are specified for artifacts. These are:

schema:name	This is a Schema.org field that is used to hold the user-supplied name of the artifact
schema:description	This is a Schema.org field that is used to hold the user-supplied description of the artifact
schema:schemaVersion	This is a Schema.org field that is used to hold the model version used when creating the artifact
pav:createdOn	This is a Provenance and Versioning Ontology (PAV) field that specifies a datetime-encoded value indicating when the artifact was created.
pav:createdBy	This is a PAV field that specifies a IRI-encoded value indicating who created the artifact.
pav:lastUpdatedOn	This is a PAV field that specifies a datetime-encoded value indicating when the artifact was last updated.
oslc:modifiedBy	This is an IRI-encoded field using an Open Services for Lifecycle Collaboration (OSLC) ontology term that specifies who updated the artifact last.

The `schema` prefix identifies the Schema.org namespace https://schema.org/, the `pav` prefix identifies the Provenance and Versioning Ontology namespace http://purl.org/pav/, and the `oslc` prefix identifies the Open Services ontology namespace http://open-services.net/ns/core#[4].

[4] Later we will formally map these prefixes to their associated namespace IRIs.

Two provenance fields that may optionally be present in artifacts are:

schema:identifier	A user-specified identifier for an artifact
pav:derivedFrom	If an artifact was copied from another artifact this field identifies the URI of that artifact

Note that the JSON Schema `title` and `description` fields are also included in an artifact specification in addition to the Schema.org-based `schema:name` and `schema:description` fields. The JSON Schema field generally holds tool-generated information whereas the Schema.org-based fields hold user-supplied information.

Representing Schema Artifact Version Information

CEDAR schema artifacts (i.e., templates, elements, and fields) can also be versioned. At present, instance artifacts are not versioned. The following fields are used to store version information:

pav:version	This is a Provenance and Versioning Ontology (PAV) field that holds the version of the artifact. Follows Semantic Versioning best practices (https://semver.org/)
bibo:status	This is a Bibliographic Ontology publication status of the artifact. Currently, valid values are bibo:draft and bibo:published.
pav:previousVersion	This field identifies the artifact that this artifact was originally copied from, if any

The `bibo` prefix identifies the Bibliographic Ontology namespace http://purl.org/ontology/bibo/.

CEDAR artifacts are versioned following standard software artifact versioning practices. The `bibo:status` status field is used to indicate whether an artifact is in draft state (`bibo:draft`) or is published (`bibo:published`). Artifacts begin in a draft state and when finialized become published. A single new version can be derived from a published artifact, using the `pav:previousVersion` field to point to the previous version. Only a single new version can be derived from a published artifact so no version branching is allowed. A new artifact's version number must be later than the artifacts it is derived from and will begin in a draft state.

Representing Artifact Structure Using JSON Schema

With JSON Schema we define the structure of the primary artifacts in the CEDAR template model. We first outline its use to define the three core artifacts in the model: template fields, template elements, and templates. We then describe the structure of template instances that conform to the schema specification provided by templates.

Representing Template Fields

Template fields are used to describe an atomic piece of metadata. Informally, they correspond to a single field in a form, which when filled out contains a single value. In principle, a template field could be stored as a simple JSON property value, such as string or number. However, in many cases we would like the option to add additional metadata to describe template fields. At a minimum, we want users to record a name and description of each field. Hence, we use a JSON object to describe template fields.

The template field representation includes a value field, in addition to the other descriptive information.

We use the JSON-LD `@value` field of type string to hold raw literal values. We also use the standard JSON Schema `title` and `description` fields to hold a name and description for the field.

For example, here is the definition of a Full Name template field, which contains the full name of a person as a single string[5]:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Full Name", "description": "Full name template field",
  "properties": { "@value": { "type": "string" } },
  "required": [ "@value" ], "additionalProperties": false,
  "schema:name": "Full Name",
  "schema:description": "A person's full name",
  "schema:schemaVersion": "1.6.0",
  "pav:version": "1.1.0",
  "bibo:status": "bibo:released",
  "pav:createdOn": "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

[5] A useful online JSON Schema validator can be found at `www.jsonschemavalidator.net`.

A conforming instance of this template field could look as follows:

```
{ "@value": "John Smith" }
```

In some cases, we may add further type restrictions to literals. For example, if we know that the literal is an email address we can use the JSON Schema `format` keyword with type `email` to restrict the value.

For example, the specification for an email field could look as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Email", "description": "Email template field",
  "properties": { "@value": { "type": "string", "format": "email" } },
  "required": [ "@value" ], "additionalProperties": false,
  "schema:name": "Email",
  "schema:description": "An email address",
  "schema:schemaVersion": "1.6.0",
  "pav:version": "1.1.0",
  "bibo:status": "bibo:released",
  "pav:createdOn″: "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

Similar format restrictions can be used for the `number`, `date-time`, `ipv4`, and `ipv6` JSON Schema formats.

The CEDAR model also distinguishes literals values from IRI values. We use the JSON-LD `@id` field in place of the `@value` field to make this distinction.

For example, here is the definition of a Home Page template field, which contains the URL of a page:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Home Page", "description": "Home page template field",
  "properties": { "@id": { "type": "string", "format": "uri" } },
  "required": [ "@id" ], "additionalProperties": false,
  "schema:name": "Home Page",
  "schema:description": "Enter a home page URL",
  "schema:schemaVersion": "1.6.0",
  "pav:version": "1.1.0",
  "bibo:status": "bibo:released",
  "pav:createdOn": "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

A conforming instance of this template field could look as follows:

```
{ "@id": "https://example.com/home/JohnSmith.html" }
```

By default the `schema:name` field can effectively be treated as the question but in some cases customized questions are desirable. Template fields also support two additional optional fields that can be used to store question text that can be presented to users. To support this we allow template fields to contain the `skos:prefLabel` and `skos:altLabel` fields. The `skos:prefLabel` field can be used to store the default question text for a field that is presented to users; the `skos:altLabel` field can contain alternate question text that may optionally be presented.

For fields with an IRI value (i.e., those using an `@id` instead of a `@value` field to hold values) we support the presence of a field to hold a user-friend label for the IRI. We use the `rdfs:label` field to store this label. This field is optional and we also allow its value to be null. We also allow a field called `skos:notation`. This field can store values that may be intended for database storage. The JSON Schema field specification must indicate the requirement for these additional fields.

Here is the previous Home Page template field specification extended to allow field instances to contain the `rdfs:label` and `skos:notation` fields:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Home Page", "description": "Home page template field",
  "type": "object",
  "properties": {
   "@id": { "type": "string", "format": "uri" },
   "rdfs:label": { "type": [ "string", null ] },
   "skos:notation": { "type": [ "string", null ] }
  },
  "required": [ "@id" ], "additionalProperties": false,
  "schema:name": "Home Page",
  "schema:description": "Enter a home page URL",
  "schema:schemaVersion": "1.6.0",
  "pav:version": "1.1.0",
  "bibo:status": "bibo:released",
  "pav:createdOn”: "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

Representing Template Fields with Multiple Values

Some fields can also contain multiple values. In the CEDAR model these multiple values are stored as an array of objects containing `@value` or `@id` fields.

For example, the content of a multi-value literal field that contains the strings "O1" and "O2" could be represented as follows:

```
[ { "@value": "O1" }, { "@value": "O2" } ]
```

Similarly, the content of a multi-value IRI field that contains, say, the IRIs `http://example.com/A1` and `http://example.com/A2` could be represented as follows:

```
[ { "@id": "http://example.com/A1", "rdfs:label": "A1" },
  { "@id": "http://example.com/A2", "rdfs:label": "A2" } ]
```

JSON Schema has inbuilt support for indicating that the value of a JSON field can be an array (see Appendix A and Section 4.1.4 of [JSON SCHEMA]).

The template field schema to capture this representation could look as follows:

```
{
 "$schema": "http://json-schema.org/draft-04/schema#"
 "title": "Home Pages", "description": "Home pages template field",
 "type": "array", "minItems": 1,
 "items": {
  "type": "object",
  "properties": {
   "@id": { "type": "string", "format": "uri" },
   "rdfs:label": { "type": [ "string", null ] },
   "skos:notation": { "type": [ "string", null ] }
  },
  "required": [ "@id" ], "additionalProperties": false,
  "schema:name": "Home Page",
  "schema:description": "Enter a home page URL",
  "schema:schemaVersion": "1.6.0",
  "pav:version": "1.1.0",
  "bibo:status": "bibo:released",
  "pav:createdOn”: "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
 }
}
```

As can be seen, we use the JSON Schema `array` directive to indicate that the field values are stored in an array. We also indicate that the array must contain at least one item. Note that this approach allows each value object to contain provenance information.

Representing Template Elements

Template elements offer composition—they can include multiple template fields and/or template elements. Template elements are represented using an approach equivalent to the one used to represent template fields. Again, we specify that a template element must be represented as a JSON object. We can then restrict each nested template field or template element using nested JSON Schema specifications.

For example, the definition of an Investigator template element is shown below. It contains one nested template field called `fullName`.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "title": "Investigator", "description": "Investigator element",
 "type": "object",
 "properties": {
  "fullName": {
   "type": "object",
   "title": "Full Name", "description": "Full name template field",
   "properties": { "@value": { "type": "string" } },
   "required": [ "@value" ], "additionalProperties": false,
   "schema:name": "Full Name",
   "schema:description": "A person's full name",
   "schema:schemaVersion": "1.6.0",
   "pav:version": "1.1.0",
   "bibo:status": "bibo:released",
   "pav:createdOn”: "2017-05-03T09:00:52-0700",
   "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
   "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
   "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
  }
 },
 "required": [ "fullName" ],
 "additionalProperties": false,
 "schema:name": "Investigator",
 "schema:description": "The lead investigator of a project",
 "schema:schemaVersion": "1.6.0",
 "pav:version": "1.1.0",
 "bibo:status": "bibo:released",
 "pav:createdOn": "2017-05-03T09:00:52-0700",
 "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
 "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
 "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

As can be seen above, the template element specification requires that the nested `fullName` field is present in instances.

A conforming *template element instance* could look like the following:

```
{
  "fullName": { "@value": "Dr. P.I." },
}
```

As with template fields, we do not require that element instances contain provenance fields.

Representing Templates

The representation of templates follows the same principles as template elements. Like template elements, templates can have nested element values and template elements.

We require that conforming template instances contain a `schema:isBasedOn` field to identify the template to which they conform (where the schema `prefix` identifies the Schema.org namespace https://schema.org/). Instances of template must also include provenance fields. The schema specification for templates must include this requirement. A template's JSON Schema `properties` fields can be used to express these requirements as follows:

```
"properties": {
  "schema:isBasedOn": { "type": "string", "format": "uri" },
  "schema:name": { "type": "string", "minLength": 1 },
  "schema:description": { "type": "string", },
  "pav:createdOn": { "type": ["string", "null"], "format": "date-time" },
  "pav:createdBy": { "type": ["string", "null"], "format": "uri" },
  "pav:lastUpdatedOn": { "type": ["string", "null"], "format": "date-time" },
  "oslc:modifiedBy": { "type": ["string", "null"], "format": "date-time" }
},
"required": [ "schema:isBasedOn", "schema:name", "schema:description",
              "pav:createdOn",  "pav:createdBy" "pav:lastUpdatedOn",
              "oslc:modifiedBy" ],
"additionalProperties": false
```

Note that with the exception of the `schema:schemaVersion` field we make all provenance fields required for template instances. With the exception of `schema:isBasedOn`, `schema:name` and `schema:description` fields, we also allow these provenance fields to be present with null values. Typically, these provenance fields are generated by server components so allowing nulls lets clients generate instances without values for these fields and still pass validation.

A complete template specification that contains a nested study title field and a nested principal investigator element could then look as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Investigation", "description": "Investigation template",
  "type": "object",
  "properties": {
    "schema:isBasedOn": { "type": "string", "format": "uri" },
    "schema:name": { "type": "string", "minLength": 1 },
    "schema:description": { "type": "string" }
    "pav:createdOn": { "type":["string", "null"], "format": "date-time" },
    "pav:createdBy": { "type":["string", "null"], "format": "uri" },
    "pav:lastUpdatedOn": { "type":["string", "null"], "format": "date-time" },
    "oslc:modifiedBy": { "type":["string", "null"], "format": "date-time" },
    "studyTitle": { ... },
    "pi": {... }
  },
  "required":
    [ "schema:isBasedOn", "schema:name", "schema:description",
      "pav:createdOn", "pav:createdBy", "pav:lastUpdatedOn",
      "oslc:modifiedBy", "studyTitle", "pi" ],
  "additionalProperties": false,
  "schema:name": "Investigation",
  "schema:description": "An investigation",
  "schema:schemaVersion": "1.6.0",
  "pav:version": "1.1.0",
  "bibo:status": "bibo:published",
  "pav:createdOn": "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

Here is an example of a template instance conforming to the above template:

```
{
  "schema:isBasedOn": "https://repo.metadatacenter.org/templates/43453",
  "schema:name": "Study",
  "schema:description": "Study template instance",
  "pav:createdOn": "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98",
  "studyTitle": { "@value": "Immune Biomarkers"  },
  "pi": { "fullName": { "@value": "Dr. P.I." } }
}
```

Representing Multiple Instances of Template Elements and Template Fields

The above specifications support the definition of nested template elements or element fields that contain exactly one instance of each. For example, only one principal investigator template element instance is allowed inside an investigation template instance. In many cases we would like items or elements to be capable of acquiring multiple instances at runtime. As mentioned earlier, JSON Schema has inbuilt support for indicating that the value of a JSON field can be an array (see Appendix A and Section 4.1.4 of [JSON SCHEMA]).

In the CEDAR template model, this approach can be used to indicate that that a template instance may contain, or must contain, multiple instances of nested template elements or template fields.

For example, we can extend the earlier investigation template to indicate that an investigation can have between 1 and 4 principal investigators as follows:

```
{
  "title": "Investigation", "description": "Investigation template",
  "type": "object",
  "properties": {
    "schema:isBasedOn": { "type": "string", "format": "uri" },
    "schema:name": { "type": "string", "minLength": 1 },
    "schema:description": { "type": "string" },
    "pav:createdOn": { "type":["string", "null"], "format": "date-time" },
    "pav:createdBy": { "type":["string", "null"], "format": "uri" },
    "pav:lastUpdatedOn": { "type":["string", "null"], "format": "date-time" },
    "oslc:modifiedBy": { "type":["string", "null"], "format": "date-time" },
    "studyTitle": { ... },
    "pi": {
      "type": array, "minItems" : 1, "maxItems" : 4,
      "items" : {
        "type": "object",
        "title": "Principal Investigator",
        "description": "Principal investigator element",
        ...
      }
    }
  },
  "required": [ "schema:isBasedOn", "schema:name", "schema:description",
      "pav:createdOn", "pav:createdBy", "pav:lastUpdatedOn",
      "oslc:modifiedBy", "studyName", "pis" ],
  "additionalProperties": false,
  "schema:name": "Investigation",
  "schema:description": "An investigation",
  "schema:schemaVersion": "1.6.0",
  "pav:version": "1.1.0",
  "bibo:status": "bibo:released",
  "pav:createdOn": "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98",
  "$schema": "http://json-schema.org/draft-04/schema#"
}
```

Representing Artifact Semantics using JSON-LD

JSON Schema is useful for defining structural restrictions on JSON documents. It can also be used to specify basic type restrictions on field values. However, it provides a very basic set of built-in type restrictions. It also does not provide a way to add additional types or to interoperate with types defined in external sources, such as RDF- or OWL-based ontologies.

As mentioned, JSON-LD [JSON-LD][6] was developed to meet this goal. JSON-LD provides a lightweight syntax to add semantic annotations to JSON documents that can restrict the types and values of fields using terms from external vocabularies. Like JSON Schema, it adds some custom fields with well-known names to a JSON document to provide additional markup information.

JSON-LD provides three core fields to add semantic markup to JSON documents: `@context`, `@type`, and `@id`. The `@context` field is used to define prefixes for controlled vocabularies and to map JSON properties to controlled vocabularies; the `@type` field indicates the semantic type of a JSON object; the `@id` field gives a unique identifier to a JSON object instance. JSON-LD is used to mark up the structural specification to add semantic content to the CEDAR templates and instances. Essentially, JSON-LD is used to add type information to JSON-described content.

Here, for example, is a JSON-LD–enhanced template instance representing a study (with JSON-LD clauses in bold):

```
{
  "@type": "http://semantic-dicom.org/dcm#Study",
  "@id": "https://repo.metadatacenter.org/template_instances/55417",
  "@context": {
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "pav": "http://purl.org/pav/",
```

[6] See Appendix B for an introduction to JSON-LD; a good introduction can also be found at `json-ld.org`.

```
    "schema": "http://schema.org/",
    "oslc": "http://open-services.net/ns/core#",
    "schema:isBasedOn": { "@type": "@id" },
    "schema:name": { "@type": "xsd:string", "minLength": 1 },
    "schema:description": { "@type": "xsd:string" },
    "pav:createdOn": { "@type": "xsd:dateTime" },
    "pav:createdBy": { "@type": "@id" },
    "pav:lastUpdatedOn": { "@type": "xsd:dateTime" },
    "oslc:modifiedBy": { "@type": "@id" },
    "studyTitle": "https://schema.org/title",
    "pi": "https://mycompany.org/property/hasPI"
  },
  "studyTitle": { "@value": "Immune biomarkers study" },
  "pi": {
    "@type": "https://schema.org/Person",
    "@id": "https://repo.metadatacenter.org/element_instances/88417",
    "@context": { "fullName": "https://schema.org/name" },
    "fullName": { "@value": "Dr. P.I" }
  },
  "schema:isBasedOn": "https://repo.metadatacenter.org/templates/4353",
  "schema:name": "Study",
  "schema:description": "Study template instance",
  "pav:createdOn": "2017-05-03T09:00:52-0700",
  "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
  "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
  "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

Note that we have added JSON-LD `@context`, `@type`, and `@id` fields to provide semantic markup. The `@context` field ensures that properties are mapped to properties in controlled vocabularies. The `@context` specification does a full mapping of prefixes to namespaces for all CEDAR model prefixes and also specifies their datatypes. The `@type` field indicates the semantic type of the instance, which in the case above is the Study class in the Radiation Oncology Ontology. Finally, the `@id` field gives a unique identifier to the instance.

The JSON Schema specification can ensure that conforming instances are marked up with JSON-LD, both by demanding that specific fields are present and by restricting the content of those fields.

If we want to enforce that the `@type` field is contained in the instance and uses a specific IRI we can do so as follows:

```
"properties": { "@type": { "enum": ["http://semantic-dicom.org/dcm#Study"] }}
```

If we want to enforce that the `@id` field is contained in the instance and contains an IRI we can do so as follows:

```
"properties": { "@id": { "type": "string", "format": "uri" }}
```

The `@context` field is far more complex but can be declaratively specified as follows:

```
"properties": {
 "@context": {
  "type": "object",
  "properties": {
   "rdfs": { "type": "string", "format": "uri",
            "enum": [ "http://www.w3.org/2000/01/rdf-schema#" ] },
   "xsd": { "type": "string", "format": "uri",
            "enum": [ "http://www.w3.org/2001/XMLSchema#" ] },
   "pav": { "type": "string", "format": "uri",
            "enum": [ "http://purl.org/pav/" ] },
   "schema": { "type": "string", "format": "uri",
               "enum": [ "http://schema.org/" ] },
   "oslc": { "type": "string", "format": "uri",
             "enum": [ "http://open-services.net/ns/core#" ] },
   "skos": { "type": "string", "format": "uri",
             "enum": [ "http://www.w3.org/2004/02/skos/core#" ] },
   "schema:isBasedOn": { "type": "object",
     "properties": { "@type": { "type": "string", "enum": ["@id"] }}},
   "schema:name": { "type": "object",
     "properties": { "@type": { "type": "string", "enum": ["xsd:string"] }}},
   "schema:description": { "type": "object",
     "properties": {"@type": { "type": "string", "enum": ["xsd:string"] }}},
   "pav:createdOn": { "type": "object",
     "properties": {"@type": { "type": "string", "enum": ["xsd:dateTime"] }}},
   "pav:createdBy": { "type": "object",
     "properties": {"@type": { "type": "string", "enum": [ "@id" ] }}},
   "pav:lastUpdatedOn": { "type": "object",
    "properties": {"@type": { "type": "string", "enum": ["xsd:dateTime"] }}},
   "oslc:modifiedBy": { "type": "object",
     "properties": { "@type": { "type": "string", "enum": ["@id"] }}},
   // Nested element and field IRI mappings here
  },
  "required": [ "rdf", "xsd", "pav", "schema", "oslc", "schema:isBasedOn",
    "schema:name", "schema:description", "pav:createdOn", "pav:createdBy",
    "pav:lastUpdatedOn", "oslc:modifiedBy"
  ], "additionalProperties": false
 }
}
```

The above schema specification is basically requiring that a template instance contains the following context definition:

```
"@context": {
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "pav": "http://purl.org/pav/",
    "schema": "http://schema.org/",
    "oslc": "http://open-services.net/ns/core#",
    "schema:isBasedOn": { "@type": "@id" },
    "schema:name": { "@type": "xsd:string" },
    "schema:description": { "@type": "xsd:string" },
    "pav:createdOn": { "@type": "xsd:dateTime" },
    "pav:createdBy": { "@type": "@id" },
    "pav:lastUpdatedOn": { "@type": "xsd:dateTime" },
    "oslc:modifiedBy": { "@type": "@id" }
}
```

Note that nested elements and fields will imply additional context specifications to ensure property assignments are made. For example, if a template contains a `studyTitle` field and we would like to map that name to the IRI `https://schema.org/title` we can add it to the template context definition as follows:

```
"@context": {
  "properties": {
     ...
     "studyTitle": { "enum": [ "https://schema.org/title" ]
  }
}
```

This restriction forces instances conforming to the template to contain the following property assignment in their context definition:

```
    "studyTitle": "https://schema.org/title"
```

Coupled with the type assignment to templates, elements and fields, this property assignment allows relationships between nested elements and fields to be mapped to controlled terms.

The overall template specification also makes the `@context`, `@type`, and `@id` fields required to ensure that instances are self descriptive.

For example, here is a JSON Schema template specification for the above study instance with clauses (marked in bold) ensuring that conforming instances carry appropriate JSON-LD markup (we elide the full context definition for brevity):

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "title": "Study", "description": "Study template",
 "@type": "https://repo.metadatacenter.org/core/Template",
 "@id": "https://repo.metadatacenter.org/templates/4353",
 "properties": {
   "schema:isBasedOn": { "type": "string", "format": "uri" },
   "schema:name": { "type": "string", "minLength": 1 },
   "schema:description": { "type": "string", "minLength": 1 }
   "pav:createdOn": { "type":["string", "null"], "format": "date-time" },
   "pav:createdBy": { "type":["string", "null"], "format": "uri" },
   "pav:lastUpdatedOn": { "type":["string", "null"], "format": "date-time" },
   "oslc:modifiedBy": { "type":["string", "null"], "format": "date-time" },
   "@type": { "enum": [ "http://semantic-dicom.org/dcm#Study" ] },
   "@id": { "type": "string", "format": "uri" },
   "@context": {
     "properties": {
       ...
       "studyTitle": { "enum": [ "https://schema.org/title" ] },
       "pi": { "enum": [ "https://mycompany.org/property/hasPI" ] }
     },
     "required":  [ ..., "studyTitle", "pi" ], "additionalProperties": false
   },
   "studyTitle": { ... },
   "pi": { ... }
 },
 "required":
   [ "schema:isBasedOn", "schema:isBasedOn", "schema:name",
     "schema:description", "pav:createdOn", "pav:createdBy",
     "pav:lastUpdatedOn", "oslc:modifiedBy",
     "@context", "@type", "@id", "studyTitle", "pi" ],
 "additionalProperties": false,
 "schema:name": "Study",
 "schema:description": "A clinical study",
 "schema:schemaVersion": "1.6.0"
 "pav:version": "1.1.0"
 "bibo:status": "bibo:published"
 "pav:createdOn": "2017-05-03T09:00:52-0700",
 "pav:createdBy": "https://metadatacenter.org/users/8d787b98",
 "pav:lastUpdatedOn": "2017-05-03T09:00:52-0700",
 "oslc:modifiedBy": "https://metadatacenter.org/users/8d787b98"
}
```

As can be seen in this example, the JSON Schema template specification can ensure that template instances contain a significant amount of JSON-LD–encoded type information. Here, we are forcing the `@context`, `@type`, and `@id` fields in an instance to carry specific controlled terms. These instances can be automatically checked for conformance against the template specification. This use of JSON Schema is completely standard and instance validation can be performed with off-the-shelf tools. We also developed a JSON Schema-based validation schema that can be used to validate template, elements, and fields [CEDAR-SCHEMA].

Representing Additional Artifact Metadata using JSON-LD

The CEDAR model requires some basic metadata for CEDAR artifacts, such as the name of the artifact (`schema:name`), its version (`schema:schemaVersion`), and its creator (`pav:createdOn`)[7]. However, sometimes these basic metadata fields are not enough. In some cases, users need to attach additional metadata to their artifacts. Some examples of these additional metadata can be the identifier used to refer to the artifact in an external system, the identifier of the organization that developed the artifact, and the name of the project where the artifact is being used.

To allow users to add custom metadata to CEDAR artifacts, the CEDAR model provides support for additional metadata fields in Templates, Elements, and Fields. These additional metadata are represented as part of an optional, nested object at the root level of the artifact's JSON Schema specification, defined using the JSON-LD keyword `@nest`. By using `@nest`, JSON-LD processors will ignore the nesting and will process the contents as if they were created directly within the containing object, that is, at the root level of the artifact's JSON Schema, where all the other artifact metadata are declared. In our model, values for the metadata fields in the `@nest` object are optional and are specified using any of the six accepted JSON data types (i.e., string, number, boolean, null/empty, object, and array).

For example, here is a JSON Schema Template specification for a *Study*, with a custom metadata field *protocol ID*, which captures an array of protocol identifiers associated with the template. Note that the *required* field does not contain a `@nest` value, because these additional metadata fields are optional.

```
{
  "schema:name": "Study",
  "@id": "https://repo.metadatacenter.org/templates/ee2f28",
  "@context": {
    ...
    "protocol ID": "https://schema.metadatacenter.org/properties/c80ace7"
  },
  ...
  "@nest": {
    "protocol ID": ["0904", "0374", "1232"]
  },
  "required": [
    "@context",
    "@id",
    "schema:isBasedOn",
    "schema:name",
    "schema:description",
```

[7] For a full list of CEDAR artifacts' provenance metadata, see Representing Artifact Provenance.

```
    "pav:createdOn",
    "pav:createdBy",
    "pav:lastUpdatedOn",
    "oslc:modifiedBy",
    "studyTitle"
  ],
  ...
}
```

Here is the representation of the additional metadata (protocol identifiers) from the previous example expressed in RDF syntax. The example shows how there is no intermediate node between the template and the protocol ids associated to it. Because we used `@nest`, the JSON processor understands that the fields defined inside the `@nest` object refer to properties of the containing object (the template).

```
...
<https://repo.metadatacenter.org/templates/ee2f28>
      <https://schema.metadatacenter.org/properties/c80ace7> "0374" .
<https://repo.metadatacenter.org/templates/ee2f28>
      <https://schema.metadatacenter.org/properties/c80ace7> "0904" .
<https://repo.metadatacenter.org/templates/ee2f28>
      <https://schema.metadatacenter.org/properties/c80ace7> "1232" .
...
```

Representing Instances as RDF

Note that CEDAR's JSON-LD instance representation can be automatically converted to an RDF representation. Here, for example, is a Turtle representation of the above study template instance:

```
<https://repo.metadatacenter.org/template_instances/55417>
  a <http://semantic-dicom.org/dcm#Study> ;
  schema:name "Immune biomarkers" ;
  schema:description "Metadata about an immune biomarkers study" ;
  schema:isBasedOn <https://repo.metadatacenter.org/template/4343> ;
  oslc:modifiedBy <https://repo.metadatacenter.net/users/6d21a887> ;
  pav:createdBy <https://repo.metadatacenter.net/users/6d21a887> ;
  pav:createdOn "2016-06-29T10:58:26-0700"^^xsd:dateTime ;
  pav:lastUpdatedOn "2016-06-29T10:58:26-0700"^^xsd:dateTime ;
  myschema:hasStudyTitle "Immune biomarkers study" ;
  myschema:hasPI [
    a <https://schema.org/Person> ;
    schema:name "Dr. P.I";
    schema:address "Stanford, CA 94305, USA"
  ] .
```

Expressing Field Value Constraints

JSON Schema allows us to express a very limited set of value constraints. We can, for example, state that the value of a field should be a particular value, or selected from a set of values. We can also restrict a field value to be of a particular type or format.

In CEDAR, we require more advanced constraints on field values that we want to come from controlled terminologies. For example, we may specify that the value of a field should be the IRI of a class in a particular ontology.

There are four main constraint types provided by CEDAR. We want to encode the constraints on the possible values for a particular field to (1) specific ontology classes, (2) ontology branches, (3) classes from specific ontologies, and (4) value sets, which are simple collections of values. Where a constraint is a collections of values, individual values may be excluded from consideration.

The possible values of a field could also be composed of some combination of the above four constraint types; the union of all the constraints is used as the set of values that may be entered by the user.

Additional constraints may be placed on numeric or string field values. The field called `_valueContraints` is used to express all constraints that cannot be represented directly in JSON Schema.

The `_valueConstraints` field

A `_valueConstraints` field that is contained inside a template field. The `_valueConstraints` field will have four possible array subfields for the four types of value sources (ontologies, classes, branches, and value sets) and and additional array field containing specifications for literal values. This field can also indicate whether the field is a multi-choice field, whether a value is required or not, and may also contain a default value. Other options include restrictions for numeric and string fields.

The overall JSON format adopted is as follows:

```
{
  "_valueConstraints": {
    "requiredValue": true | false,
    "multipleChoice": true | false,
    "numberType": "xsd:integer" | ...,
    "unitOfMeasure": "...",
    "minValue":
    "maxValue":
```

```
    "decimalPlaces":
    "minLength":
    "maxLength":
    "temporalType": "xsd:date" | "xsd:dateTime" | "xsd:time",
    "ontologies": [ ... ],
    "valueSets": [ ... ],
    "classes": [ ... ],
    "branches": [ ... ],
    "literals": [ ... ],
    "defaultValue": "..."
  }
}
```

These fields are now explained in turn.

General Value Constraint Fields

Boolean field `requiredValue` indicates whether a value is required for a field. A boolean `multipleChoice` field indicates if more than one answer is acceptable for a field.

String-based fields may have properties `minLength` and `maxLength` that indicate minimum and maximum lengths for strings.

Numeric Value Constraint Fields

Numeric fields can contain a field called `numberType` in the `_valueConstraints` object indicating the field datatype and a `unitOfMeasure` field indicating the associated units. Numeric fields can also contain fields called `minValue` and `maxValue` to indicate minimum and maximum values for numeric fields. Floating point fields can also contain a `decimalPlace` field specifying the number of decimal places displayed.

Temporal Value Constraint Fields

Temporal fields can contain a field called `temporalType` in the `_valueConstraints` object. Three temporal types are currently supported: `xsd:date`, `xsd:dateTime`, and `xsd:time`.

The `ontologies` Value Constraint Field

This field contains a set of ontologies from which controlled terms can be selected. In stores an array of IRIs of ontologies, together with an acronym and short name for each specified ontology.

The following example shows an ontologies value constraint that specifies that field values should come only from the MEDDRA and RXNORM ontologies:

```
"ontologies": [
  {
   "uri": "http://bioportal.bioontology.org/ontologies/MEDDRA",
   "acronym": "MEDDRA",
   "name": "Medical Dictionary for Regulatory Activities Terminology"
  },
  {
   "uri": "http://bioportal.bioontology.org/ontologies/RXNORM",
   "acronym": "RXNORM",
   "name": "RxNorm Vocabulary"
  }
]
```

The `classes` Value Constraint Field

A common use case is to constrain the values of a field to a predefined set of classes, not necessarily from the same ontology. For example, to constrain the possible values for a field called `studyType` to one of the classes “Observational Study” (`http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#C16084`) from the NCIT ontology and “Longitudinal Study” (`http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#Longitudinal_Study`) from the SYN ontology one can do the following:

```
{
 "studyType": {
  ...
  "_valueConstraints": {
   "classes": [
    {
     "uri": "http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#C16084",
     "label": "Observational",
     "default": true
    },
    {
     "uri":
"http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#Longitudinal_Study",
     "label": "Longitudinal",
     "default": false
    }
   ],
   "multipleChoice": false
  }
 }
}
```

The `branches` Value Constraint Field

The branches field is analogous to the ontologies field, but restricts values to branches within ontologies.

Hers is an example that restricts the possible values to classes in branches rooted in assay classes in the Ontology of Biomedical Investigation and in the GALEN ontology.

```
{
  "_valueConstraints": {
    "branches": [
     {
        "uri": "http://purl.obolibrary.org/obo/OBI_0000070",
        "maxDepth": 3,
        "includesRoot": false
     },
     {
        "uri": "http://www.co-ode.org/ontologies/galen#Assay",
        "includesRoot": false
     }
    ],
    "multipleChoice": true
  }
}
```

The `valueSets` Value Constraint Field

This field constrains the accepted values to one of several classes from particular value sets.

The following example shows a value set constraint that specifies that field values should come only from the ACE Inhibitor or ARB and ADHD Medications NLM value sets:

```
{
  "_valueConstraints": {
    "valueSets": [
     {
      "name": "ACE Inhibitor or ARB",
      "vsCollection": "http://data.bioontology.org/ontologies/NLMVS",
      "uri": "http://purl.bioontology.org/ontology/NLMVS/2.16.32",
      "numTerms": 3
     },
     {
      "name": "ADHD Medications",
      "vsCollection": "http://data.bioontology.org/ontologies/NLMVS",
      "uri": "http://purl.bioontology.org/ontology/NLMVS/2.16.840",
      "numTerms": 33
     }
    ]
  }
}
```

The `literals` Value Constraint Field

This field constrains the accepted values to one or several string literals.

The following example shows a literals constraint that specifies that field values should come only from the values “Germany”, “France”, and “UK”:

```
{
  "_valueConstraints": {
    "literals": [
     {
      "label": "Germany"
     },
     {
      "label": "France"
     },
     {
      "label": "U.K."
     }
    ]
  }
}
```

Each option within a literal choice can also have an optional boolean field called `selectedByDefault`. When present and set to true this field indicates that this option is selected when presented.

The `defaultValue` Value Constraint Field

This field can be used to store the default value for string- and URI-based fields.

For string-based fields the value is simply stored as a string. For example, of the default value for a field is "Yes" it would be stored as follows:

```
"defaultValue": "Yes"
```

For URI-based fields the type of the value is stored in a field called `type`. Possible values are "Value" or "OntologyClass". The default URI is stored in a field called `termURI`; the source ontology for the term is contained in a field called `sourceURI`.

```
"defaultValue": {
    "type": "Value" | "OntologyClass",
    "sourceUri": <URI>,
    "termUri": <URI>
}
```

Representing User Interface Rendering Specifications

CEDAR's templates can also contain markup that can drive knowledge acquisition tools. This markup has no effect on the semantics of templates, elements, or fields. It is used at design time to indicate rendering preferences when displaying templates, and at instance population time to specify types of user interface elements that should be used when generating instances from templates.

In a CEDAR template all user interface markup is contained in a field called `_ui`. This field is present in templates, template elements, and field elements. Instances do not contain this user interface field. The associated template is used to indicate how populated instances are displayed. The user interface field contains no modeling information - it specifies rendering choices only.

Template Rendering Information

CEDAR templates contain an ordered collection of template elements and fields. Since JSON Schema does not have directives to specify field ordering we store this order in the `_ui` field. We use a field called `order` to store this information. This field contains an array containing the names of the enclosed fields and elements, with the order following the array order.

Since a template can have multiple pages and each page can contain a mixture of template elements and fields, a pages specification is also needed. A field call `pages` contains the information. This field contains a two-dimensional array. The first dimension stores the page ordering. Each element in this array stores the order of template elements and fields on a page.

A field called `propertyLabels` is used to map JSON field names to customized display names for the enclosing template or element. It contains a map of JSON field names to display names. Similarly, an optional field called `propertyDescriptions` is used to customize field descriptions.

The optional fields `header` and `footer` are used to specify header and footer information for the template. The header and the footer will be displayed by the metadata acquisition tool when rendering the template.

For example, a template containing two pages, each of which has two template elements or fields could look as follows:

```
"_ui": {
  "order": [
    "principalInvestigator", "study", "contactInformation", "institution"
  ],
  "pages": [
    [ "principalInvestigator", "study" ],
    [ "contactInformation", "institution" ]
  ],
  "propertyLabels": {
    "name": "PI Name"
  },
  "propertyDescriptions": {
    "name": "Enter the name of the PI"
  },
  "header": "This template should be filled out by the Principal Investigator",
  "footer": "This template must be used without any changes to the questions or
to the order of questions. To suggest any changes, please contact
john.doe@acme.com"
}
```

Template Element Rendering Information

As with a template, a template element can have nested fields and elements so we also need an `order` field to indicate their order. Template elements are not paged so a `pages` field is not needed. Like templates, elements can optionally contain `propertyLabels` and `propertyDescriptions` fields, as well as `header` and `footer` fields.

Here is an example of a `_ui` field for a template element:

```
"_ui": {
  "order": [ "name", "description" ],
  "propertyLabels": {
    "address": "Address"
  },
  "propertyDescriptions": {
    "address": "An address"
  }
}
```

Template Field Rendering Information

Every field has an `inputType` field that indicates the type of user interface element that can be used to display the field.

The current possible core field types are `textfield`, `textarea`, `radio`, `checkbox`, `temporal`, `email`, `list`, `numeric`, `phone-number`, `section-break`, `richtext`, `image`, `link`, and `youtube`.

Finally, a field called `valueRecommendationEnabled` indicates whether the field's value should be used for CEDAR's intelligent authoring facilities.

Here is an example `_ui` field for a text field:

```
"_ui": {
  "inputType": "textfield",
  "valueRecommendationEnabled": true
}
```

A template field can also be indicated as hidden, in which case the field will not be rendered in an acquisition tool. An optional boolean-valued field called `hidden` can be used inside the `_ui` object to indicate this state. In general, hidden fields must have a default value specified via an appropriate value constraint on the field.

If the field input type is indicated as `temporal` then a specific temporal type must be specified in the associated `_valueConstraints` section. A field called `temporalType` must be present in this section if the input type is temporal. Current possible values are `xsd:time`, `xsd:date`, and `xsd:dateTime`.

For these three temporal types, a matching `temporalGranularity` field must also be present in the field's `_ui` section. This field indicates the finest granularity at which temporal information should be acquired and displayed. Possible values are `year`, `month`, `day`, `hour`, `minute`, `second`, and `decimalSecond`. Irrespective of the values of these granularity fields, the actual stored time, date, or datetime value in the field instance must follow the [XML Schema Datatype specification](), meaning that padding of values may be required. For example, if a finest granularity of `month` is specified for a date field and a user enters 1999-12 that value must be padded to, say 1999-12-01 to ensure that the stored value satisfies the specification of `xsd:date` values.

Two additional fields can be added to a temporal field's `_ui` section to control display and acquisition: `timeZoneEnabled` and `inputTimeFormat`. The `timeZoneEnabled` field is boolean and indicates if time zone information should be acquired and displayed for the field. The `inputTimeFormat` field is currently used to indicate whether a 24-hour or 12-hour clock is to be used to display and acquire time. Possible values are `12h` and `24h`, respectively.

Here is an example `_ui` and `_valueConstraints` specification for a datetime field that uses a 24-hour clock, has a finest granularity of days, and displays and acquires time zone information:

```
"_ui": {
   "inputType": "temporal",
   "temporalGranularity": "day",
   "inputTimeFormat: "24h",
   "timeZoneEnabled": true
},
"_valueConstraints": {
   "temporalType": "xsd:dateTime"
}
```

Attribute-Value Field Rendering Information

CEDAR also supports a type of field attribute-value fields, which allow users to dynamically add fields to a template instance in a controlled way.

For example, if we'd like users to be able to add new fields in a particular place in the form we can position an attribute-value field in the desired location in the enclosing template or element.

Since we cannot directly specify a JSON Schema specification for an instance field whose name is not known we need a level of indirection for these types of fields. Basically, the field specification for an attribute-value field specifies an array that will contain the user-defined attribute names in an instance. This array will contain an (implicitly) ordered list of attribute field names. We then use a JSON Schema `additionalProperties` specification to indicate how the values of such fields must appear in the instance.

For example, a field specification for an attribute-value field called `"My User-Defined Fields"` would look as follows:

```
"My User-Defined Fields": {
   "type": "array",
   "minItems": 0,
   "items": {
      "@type": "https://schema.metadatacenter.org/core/TemplateField",
      "@id": "https://repo.metadatacenter.org/template-fields/7ee0",
      ...
      "_ui": { "inputType": "attribute-value" },
      "_valueConstraints": { "requiredValue": false },
      "type": "string"
      ...
  }
}
```

Here we have a field type of `attribute-value`. The schema specifies that the instance contains a field called "My User-Defined Field" which is an array of strings. These string store the names and (implicitly) the order of the attribute fields added to instances. We do NOT add the attribute-value field name to the enclosing template or element `required` field because a field of this name will not actually appear in the instances - the user will be specifying the names of attribute fields in the instance itself.

For example, a template instance with two attribute fields defined using the "My User-Defined Fields" attribute-value field with the names "Name" and "Alias" would have an entry as follows:

```
"My User-Defined Fields": [ "Name", "Alias" ]
```

We can use an `additionalProperties` specification on enclosing templates and elements as follows to allow new attribute fields to appear in the instances:

```
"additionalProperties": {
  "type": "object",
  "properties": {
    "@value": { "type": [ "string", "null" ] },
    "@type": { "type": "string", "format": "uri" }
   },
   "required": [ "@value" ],
   "additionalProperties": false
}
```

Any attribute fields in an instance must follow this specification. They will look like a normal fields.

Note that here the source field specification in the schema is not actually used to specify the format of the field instance - instead, the `additionalProperties` specification is. The source field specification is however could be used for value constraints, for example.

A template instance with values for the above attribute fields "Name" and "Alias" could look as follows:

```
"My User-Defined Fields": [ "Name", "Alias" ],
"Name": { "@value": "Fredrick" },
"Alias": { "@value": "Fred" }
```

Templates and elements can contain an unlimited number of attribute-value fields. Similarly, instances can contain an unlimited number of fields derived from a particular attribute-value field specification. In an instance, the name of an attribute-value field must be present in the associated attribute-value field array. This name can be used to resolve the source attribute-value field specification in the template and thus allow disambiguation.

At present only string values are allowed in attribute-value fields. This value can optionally be typed using a JSON-LD `@type` specification.

Note that when a user adds a new attribute-value field an `@context` entry must be made in the instance for that field (and removed when an attribute-value field is removed). This entry will allow users to label these fields with RDF properties, thus allowing the fields and their values to appear as first class entities when generating RDF.

Appendix A: JSON Schema

JSON Schema is a technology for describing and validating the structure of JSON data [JSON SCHEMA]. Its directives—themselves represented as standard JSON elements—can be used to provide a structural description of any JSON document. JSON documents that are specified with JSON Schema can be structurally validated against their associated schemas via off-the-shelf tools.

JSON Schema provides a set of directives to describes the structure of a JSON document. A JSON Schema specification contains a set of JSON Schema directives and is represented as a standard JSON document. A JSON Schema description is specified as a JSON object. The presence of a top-level field named `$schema` in a JSON object signals that it is a JSON Schema specification. The value of this field identifies the particular version of the JSON Schema specification that is being used.

A `type` field indicates the required type of the conforming JSON object. The possible value for this field are the core JSON types: `object`, `array`, `string`, `boolean`, `numeric`, and `null`. JSON Schema description objects can also optionally contain `title` and a `description` fields, which are descriptive only.

Here is a minimal JSON Schema description:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Minimal JSON Schema",
  "description": "Minimal JSON Schema specification"
}
```

The core JSON Schema directive is represented using a field called `properties`. This directive describes the fields that a conformant JSON object might or must have, together with associated sub-schema that constrain the values of these fields. The various fields in a schema and restrictions on them are listed in the `properties` field. The field names and their type information can be specified at this level.

As associated field called `required` is used to signal if those fields are required in a conforming JSON document. An `additionalProperties` Boolean field can also be included to indicate whether properties beyond those listed in the `properties` field can be included in a conforming instance data.

Here is a JSON Schema description for an empty JSON document:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Empty JSON Schema",
  "description": "An empty JSON Schema specification",
  "properties": {},
  "required": [],
  "additionalProperties": false
}
```

This schema description specifies a JSON object that must not contain any fields. The only conforming JSON instance will be an empty object, i.e., it will be: `{}`.

The field values are in turn specified using JSON Schema.

For example, a schema description for a simple JSON object representing a basic study design, which has two required fields called `briefTitle` and `principalInvestigatorName`, can be specified in JSON Schema as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Basic study design",
  "description": "Basic study design JSON Schema specification",
  "properties": {
    "briefTitle": { "type": "string" },
    "principalInvestigatorName": { "type": "string" }
  },
  "required": [ "briefTitle", "principalInvestigatorName" ],
  "additionalProperties" : false
}
```

The above definition indicates that in any JSON document that follows this schema, `briefTitle` and `principalInvestigator` must be present and contain JSON strings, even if their values are empty. The field `additionalProperties` is `false`, which means that properties other than those listed in `properties` are not allowed. A conforming instance could look as follows:

```
{
  "briefTitle": "A Big Study",
  "principalInvestigatorName": "Dr. P.I."
}
```

Restricting Property Values

In addition to restricting the type of a property, JSON Schema can also be used to restrict the values that a property can take. A JSON Schema field called `enum` can be used in a property definition directive to specify this value restriction. This field must have a value that is an array with at least one element, where each element is unique. The values in this array effectively specify the allowed values for the field.

For example, let's suppose that we would like a JSON object with a single required field called `language` and would like to restrict the possible values of that field to one of the strings "English" and "Spanish". This can be achieved using the following schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Language type",
  "description": "Language type JSON Schema specification",
  "properties": {
    "language": {
      "type": "string",
      "enum": [ "English", "Spanish" ]
    }
  },
  "required": [ "language" ],
  "additionalProperties": false
}
```

A conforming JSON document fragment would be:

```
{
  "language": "English",
}
```

An array with a single element can be used to restrict a field to a single value. For example, if we would like to specify that the above `language` field should only contain the language "English" we can simply specify that language as the single value in the array (i.e., `"enum" : ["English"]`).

Nesting JSON Schema specifications

As mentioned, field definitions inside a JSON Schema specification can themselves contain JSON Schema specifications, which effectively allows JSON Schema specification to be nested to arbitrary depths. For example, if we wish to indicate that the principal investigator named in the previous study design Schema is actually a compound object containing forename and surname fields we can express this as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Basic study design",
  "description": "Basic study design JSON Schema specification",
  "properties": {
    "briefTitle": { "type": "string" },
    "principalInvestigatorName":
    {
      "type": "object",
      "properties": {
        "forename": { "type": "string" },
        "surname": { "type": "string" }
      },
      "required": [ "forename", "surname" ],
      "additionalProperties" : false
    }
  },
  "required": [ "briefTitle", "principalInvestigatorName" ],
  "additionalProperties" : false
}
```

As can be seen above, it is not necessary to repeat the `$schema` field inside nested elements. An instance conforming to the above specification would look as follows:

```
{
  "briefTitle": "A Big Study",
  "principalInvestigatorName": {
    "forename": "Patrick",
    "surname": "O'Bannion"
  }
}
```

Reusing JSON Schema specifications with `$ref`

To support the reuse of schema specifications, JSON Schema also includes a `$ref` directive. This directive can be used to refer to external JSON Schema descriptions. For example, instead of inlining the principal investigator name specification inside the study design specification, we can separately define the the principal investigator name and use the `$ref` directive to refer to it inside the study design specification:

```
{
  "principalInvestigatorName": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "title": "Principal Investigator Name",
    "description": "Principal investigator name JSON Schema specification",
    "properties": {
      "forename": { "type": "string" },
      "surname": { "type": "string" }
    },
    "required": [ "forename", "surname" ],
    "additionalProperties" : false
  },
  "basicStudyDesign": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "title": "Basic study design",
    "description": "Basic study design JSON Schema specification",
    "properties": {
      "briefTitle": { "type": "string" },
      "principalInvestigatorName": { "$ref": "#/principalInvestigatorName" }
    },
    "required": [ "briefTitle", "principalInvestigatorName" ],
    "additionalProperties" : false
  }
}
```

The reference uses JSON Pointer [JSON-POINTER] to specify the location of the referenced JSON Schema object. Here, the reference is to a field inside a JSON object in the same file. The reference can also be prefixed with a relative or absolute URL to reference a web-accessible resource.

Representing Arrays in JSON Schema

JSON Schema has inbuilt support for indicating that the value of a JSON field can be an array (see Section 4.1.4 of [JSON SCHEMA]).

For example, if we have a JSON field called `f1` that can contain an array of 2 to 4 objects we can express this in JSON Schema as:

```
"f1": {
  "type": "array", "minItems" : 2, "maxItems" : 4,
  "items" : {
    "type" : "object"
  }
}
```

A JSON document fragment for the property `f1` conforming to this schema could then look something like:

```
"f1": [ { <some JSON object> }, { <some JSON object> } ]
```

Here, the array elements could contain any JSON object.

If we want to restrict the schema of the objects in the array we can simply embed standard JSON Schema description inside the `items` field value.

For example, if we have the following JSON Schema description of a person object:

```
{
  "$schema": "http://json-schema.org/schema#"
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age { "type": "number" }
  },
  "required": [ "name", "age" ],
}
```

we can extend the earlier definition of the `f1` field to indicate that the array must consist of person objects as follows:

```
"f1": {
  "$schema": "http://json-schema.org/schema#"
  "type": array, "minItems" : 2, "maxItems" : 4,
  "items" : {
    "type": "object",
    "properties": {
      "name": { "type": "string" },
      "age { "type": "number" }
    },
    "required": [ "name", "age" ],
  }
}
```

An example of a JSON document fragment for property "f1" conforming to this JSON Schema could then be:

```
"f1": [ { "name" : "Fred", "age": 55 },
        { "name" : "Bob", "age": 26 }
      ]
```

Appendix B: JSON-LD

JSON-LD provides a lightweight syntax to add semantic annotations to JSON documents [JSON-LD]. The key goals of JSON-LD are to support the use of Linked Data in Web-based programming environments, to build interoperable Web services, and to store Linked Data in JSON-based storage engines. JSON-LD effectively allows JSON documents and their contents to be made available as Linked Data, offering the potential for machine-interpretable RDF semantics

Core JSON-LD functionality is provided with just three fields: `@type`, `@id`, and `@context`. We will first describe these fields and outline how they can be used to add semantic markup to JSON documents.

JSON-LD @type Field

The `@type` field is used by JSON-LD to provide a principled way of adding additional type information to JSON objects. The value of this field is one or more URIs indicating the type or types of the associated object or field. (This constraint must be specified with JSON Schema on each `@type` declaration.)

For example, here is a simple template element for a study design where we indicate that each metadata instance must have a `@type` field, and that the value of the field must be a URI:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Basic Study Design",
  "description": "Basic example of a template element to describe a study",
  "properties": {
    "@type": {
      "type": "string", "format": "uri"
    },
    "briefTitle": { ... },
    "principalInvestigator": { ... }
  },
  "required": [ "@type", "briefTitle", "principalInvestigator" ],
  "additionalProperties": false
}
```

The following is an example of a conforming instance:

```
{
  "@type":  "https://example.com/SomeType",
  "briefTitle": { ... },
  "principalInvestigator": { ... },
}
```

It contains a `@type` field with a URI identifying a type.

If we wish to constrain the value of the `@type` field we can use JSON Schema's `enum` clause to constrain the field value.

For example, to force the `@type` field in the instance data to contain the URI `http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#C19067` we can extend the above specification as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "Basic Study Design",
  "description": "Basic example of a schema to describe a study",
  "properties": {
    "@type": {
      "type": "string", "format": "uri",
      "enum": [ "http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#C19067" ]
    },
    "briefTitle": { ... },
    "principalInvestigator": { ... }
  },
  "required": [ "@type", "briefTitle", "principalInvestigator" ],
  "additionalProperties": false
}
```

A conforming instance would then need to include a `@type` field with the specified URI as its value:

```
{
  "@type":  "http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#C19067",
  "briefTitle": { ... },
  "principalInvestigator": { ... },
}
```

This is basically a standard JSON Schema approach to forcing the values in JSON instance data to come from controlled term lists. The data will not validate if each field does not contain an exact value from the enumerated list.

If we wish to indicate that `@type` field may contain one or more URIs, we can use the JSON Schema `oneOf` directive to add that option. Here is the resulting specification:

```
"@type": {
  "oneOf": [
    {
      "type": "string", "format": "uri"
    },
    {
      "type": "array",
      "items": {
        "type": "string", "format": "uri"
      },
      "minItems": 1,
      "uniqueItems": true
    }
  ]
}
```

The above specification states that the `@type` field must contain **either** a single quotation-enclosed URI value, **or** one or more quotation-enclosed URIs in a JSON array.

Examples of conforming field instances include:

```
"@type": "https://schema.org/Person"
"@type": [ "https://schema.org/Person" ]
"@type": [ "https://schema.org/Person", "https://schema.org/Place" ]
```

JSON-LD @id Field

JSON-LD also provides a universal identifier mechanism for JSON objects. It includes an identifier field called `@id` which contains an URI-encoded identifier. This field allows JSON objects to be identified via a web-accessible URI and allows the values of JSON fields to refer to a JSON object on a different site on the Web.

For example, here is a JSON Schema definition for a study design, which has been enhanced with JSON-LD `@id` markup:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "@id": "https://example.com/StudyDesign",
  "title": "Basic Study Design",
  "description": "Basic example of a schema to describe a study",
  "properties": {
      "briefTitle": { ... },
      "principalInvestigator": { ... }
    }
  }
  "required": [ "briefTitle", "principalInvestigator" ],
  "additionalProperties": false
}
```

Here the `@id` field is used to define the identifier used for this metadata template (i.e., `https://example.com/StudyDesign`). This field will make it possible to uniquely identify the schema specification object and to externally reference it.

We can also indicate that an instance conforming to this JSON Schema definition must include an `@id` field.

For example, here is the above study design template extended to force instances to contain an `@id` field:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "@id": "https://examle.com/StudyDesign",
  "title": "Basic Study Design",
  "description": "Basic example of a schema to describe a study",
  "properties": {
    "@id": { "type": "string", "format": "uri" },
    "briefTitle": { ... },
    "principalInvestigator": { ... }
   }
  },
  "required": [ "@id", "briefTitle", "principalInvestigator" ],
  "additionalProperties": false
}
```

The second `@id` field is located inside the `properties` object and it has also been set as a `required` property, so it indicates that any JSON document that follows this schema must have a property `@id` to identify it, whose value will be a URI. It restricts the identifier value in the field to be a URI, though users can use their own identifier mechanism to provide the actual identifier.

JSON-LD @context Field

Another key field defined by JSON-LD is named `@context`, and it is used to establish the namespaces for the elements in the document by mapping JSON field names to URIs. Primarily it is used to map field names to URIs identifying properties.

For example, a context definition mapping a `_value` field to the Schema.org `https://schema.org/value` property be:

```
"@context": {
  "_value": "https://schema.org/value"
}
```

The `@context` field is slightly more complex to specify than `@type` or `@id` fields, but adopts their same JSON-Schema-based specification approach.

For example, to force a JSON instance to contain the following `@context` field:

```
"@context": {
    "title": "https://schema.metadatacenter.org/title",
    "year": "https://schema.metadatacenter.org/year",
    "_value":"https://schema.org/value"
}
```

we can write the following in the JSON Schema-encoded specification:

```
"properties": {
  "@context": {
    "properties": {
      "title": { "enum": [ "https://schema.metadatacenter.org/title" ] },
      "year": { "enum": [ "https://schema.metadatacenter.org/year" ] },
      "_value": { "enum": [ "https://schema.org/value" ] }
     },
    "required": [ "title", "year", "_value" ],
    "additionalProperties": false
  },
  "required": [ "@context" ]
}
```

What we are basically trying to do here is to force the data to contain specific URIs that encode type information for the properties in the JSON object.

Appendix C: Model Change Log

1.6.0

Released June 27th, 2020.

The following model updates were made in the 1.6.0 version of the model:

(1) The datetime field was superseded by a new temporal field, which supports three types of temporal data: dates, times, and datetimes. The new temporal field is described in the Template Field Rendering Information section.

(2) Template instances can now optionally contain a field called `schema:identifier`. This field can be used to store user-friendly identifiers for instances or can be used to store external identifiers for instances imported from external systems. This field is described in the Representing Artifact Provenance section.

1.5.0

Released December 4th, 2018.

The following model updates were made in the 1.5.0 version of the model:

(1) Schema artifacts (i.e., templates, elements, and fields) can now optionally contain a field called `schema:identifier`. This field can be used to store user-friendly identifiers for artifacts or can be used to store external identifiers for artifacts imported from external systems. This field is described in the Representing Artifact Provenance section.

(2) Template field specifications can now optionally contain `skos:prefLabel` and `skos:altLabel` fields. The `skos:prefLabel` field stores question text for a field; the `skos:altLabel` field can be used to store one or more alternate question text for a field. These fields are described in the Representing Template Fields section.

(3) Template field instances containing controlled term value can now optionally include a `skos:notation` field. This field can be used to store values targeted for computer interpretation (as opposed to the `rdfs:label` field, which is used to store user-friendly values). This field is described in the Representing Template Fields section.

(4) Several additions were made to the `_valueConstraints` field. Numeric fields can now contain a field called `numberType` in the `_valueConstraints` object indicating the field datatype

and a `unitOfMeasure` field indicating the associated units. Numeric fields can also contain fields called `minValue` and `maxValue` to indicate minimum and maximum values. Floating point fields can also contain a `decimalPlace` field specifying the number of decimal places displayed. New properties `minLength` and `maxLength` can be used to indicate minimum and maximum lengths for strings. Finally, the existing `defaultValue` field can now hold defaults for URI-based field values.

(5) A field called `propertyDescriptions` was added to the `_ui` field in templates and elements. It functions much like the existing `propertyLabels` field and is used to map JSON field names to customized field names for enclosing templates and elements.This field is described in the Representing User Interface Rendering Specifications section.

1.4.0

Released May 1st, 2018. Source document here.

The following model updates were made in the 1.4.0 version of the model:

(1) Schema artifacts (i.e., templates, elements, and fields) can now be versioned. The additional version fields are `pav:version`, `pav:previousVersion`, and `bibo:status`. These fields are described in the Representing Artifact Version section.

(2) Artifacts can now contain an optional `pav:derivedFrom` field to indicate that the artifact was copied from another resource. This field is described in the Describing Artifact Provenance section.

(2) The model now allows users to add additional fields to instances in a controlled way via the addition of a new attribute-value field type. This field is described in the Attribute-Value Field Rendering Information section.

1.3.0

Released November 1st, 2017. Source document here.

The following model updates were made in the 1.3.0 version of the model:

(1) Template field instances with IRI values previously used a custom `_valueLabel` field to store labels for their associated IRI. This field has now been replaced with the standard `rdfs:label` field, with the RDFS prefix mapped appropriately in the context for a template instance.

For example, an IRI field value like the following:

```
{ "@id": "https://example.com/A", "_valueLabel": "A"}
```

would now look like:

```
{ "@id": "https://example.com/A", "rdfs:label": "A"}
```

This approach will now generate a meaningful RDF graph from the JSON-LD.

(2) Previously, template instance names and descriptions were stored inside the `_ui` field. We now store these at the top level of an instance using the standard `schema:name` and `schema:description` fields.

For example, an instance like the following:

```
{
  "_ui": {
    "title": "Study", "description": "A clinical study"
  },
  ...
}
```

would now look like:

```
{
  "schema:name": "Study",
  "schema:description": "A clinical study"
  "_ui": { ... },
  ...
}
```

Again, this approach will result in more meaningful RDF being generated from the instance JSON-LD.

1.1.0

Released May 4th, 2017. Source document here.

Version 1.1.0 should be considered at the first stable release of the model. Numerous minor updates and fixes were made from earlier model releases.

Glossary

Metadata – descriptors that describe the properties of data
Metadata Template – a composite set of metadata template elements and value elements
Metadata Template Element – reusable representation of one or more metadata descriptors relating to a particular aspect of some data; metadata template elements may contain more or more value elements and may be combined recursively to create more complex elements
Metadata Template Field - an atomic piece of metadata
Metadata Template Instance – instantiated metadata template
URI – Universal Resource Identifier

References

[BIOCADDIE] Ohno-machado, Lucila; Alter, George; Fore, Ian; Martone, Maryann; Sansone, Susanna-Assunta; Xu, Hua (2015): bioCADDIE white paper - Data Discovery Index. https://dx.doi.org/10.6084/m9.figshare.1362572.v1

[CEDAR SCHEMA] CEDAR Template Model Schema https://github.com/metadatacenter/cedar-templates/blob/master/validation/template_validator.json

[EKAW2016] O'Connor MJ, Martinez-Romero M, Egyedi AL, Willrett D, Graybeal J, Musen MA. An open repository model for acquiring knowledge about scientific experiments. In: LNCS. Vol 10024; 2016:762-777.

[JSON] The JSON Data Interchange Format, 1st Edition. October 2013. ECMA International, Standard ECMA-404. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[JSON-LD] JSON-LD 1.0, A JSON-based Serialization for Linked Data, W3C Recommendation 16 January 2013. https://www.w3.org/TR/json-ld/. *See also* web site: http://json-ld.org

[JSON-LD-PLAYGROUND] http://json-ld.org/playground

[JSON-POINTER] http://spacetelescope.github.io/understanding-json-schema/structuring.html

[JSON-SCHEMA] http://json-schema.org

[JSON-SCHEMA-STSI] http://spacetelescope.github.io/understanding-json-schema/UnderstandingJSONSchema.pdf

[JSON-VALIDATE] JSON Schema Validator: http://www.jsonschemavalidator.net

[LINKED-DATA] http://en.wikipedia.org/wiki/Linked_data

[OWL-CONSTRAINTS] Motik B, Horrocks I, and Sattler U. Adding Integrity Constraints to OWL. OWLED, 2007; Vol. 258

[SHACL] Shapes Constraint Language (SHACL) https://www.w3.org/TR/shacl/

[SNOMEDCT-INSTITUTION] SNOMEDCT Institution class: http://purl.bioontology.org/ontology/SNOMEDCT/385437003

[PAV] Provenance and Versioning Ontology https://pav-ontology.github.io/pav/pav.rdf