Overview

This is a concrete proof of concept for the use of a global schema + schema service as the basis for our metadata workflows.

To recap, the major parts to this proposal are:

- A global schema. This defines the list of properties that can be present in the metadata blob. The metadata blob itself is a single JSON object that contains properties defined in that global schema. It is not divided into sections such as "common", "crossref", etc. The advantage of this approach is its simplicity - anything contributing to the metadata blob simply merges its output with the blob, and consumers of the blob simply look for the fields they need.

- Individual schemas reference properties defined in the global schema, and are used for the dual purpose of display (to define the fields present in Alpaca forms displayed in the UI), and for validation.
- A schema service that produces the N individual schemas drive Ember's UI and validation. These N schemas are based on the circumstances of a particular submission (who submitted it, what repositories it's going into, etc)

Schema Details

Alpaca js renders forms based upon the contents of a JSON schema. That being said, a key insight to keep in mind is that *form display and validation may be treated as separate concerns.* This means that it is possible to write a schema that will *render* only a couple fields in Alpaca, but can be used to *validate* the required presence of many more in the metadata blob.

The global schema

The global schema defines all properties allowed to be present in the metadata blob (e.g. their names and types), as well as Alpaca options for each field. The property definitions are used for validation, and the alpaca options for influencing display and custom validation characteristics in Alpaca. An example cut down global schema (defining only two fields) is as follows. The colors represent sections of different significance:

```
{
   "$schema": "http://json-schema.org/draft-07/schema#",
   "$id": "http://localhost:8080/schemas/global.json",
   "title": "JHU global schema",
   "type": "object",
   "required": ["title", "journal-title"],
   "additionalProperties": false,
   "properties": {
       "title": {
           "title": "Article / Manuscript Title",
           "description": "The title of the individual article or manuscript being submitted",
           "type": "string"
       },
       "journal-title": {
           "title": "Journal title",
           "description": "Title of the journal being submitted to",
           "type": "string"
       }
   },
```

```
   "options": {
       "fields": {
           "title": {
               "type": "textarea",
               "rows": 2,
               "cols": 100,
               "label": "Article / Manuscript Title",
               "placeholder": "Enter the manuscript title",
               "hidden": false
           },
           "journal-title": {
               "type": "text",
               "label": "Journal Title",
               "placeholder": "Enter the journal title",
               "hidden": false
           },
       }
}
```

Note: a global schema isn't strictly required for this proposal to work. That is to say, Ember forms, validation, and the general pattern of merging to a single metadata blob would still work without a global schema. The global schema just adds discipline and uniformity to the process of creating and validating individual schemas, and serves as a convenient single point of documentation for known-valid fields.

required, additionalProperties:false

These properties are used for the purpose of schema validation.

If the metadata blob is validated against the global schema, the **`required`** field enumerates the fields that MUST be present in the blob under all circumstances for it to be valid with respect to the global schema. If this is not present, then no fields are globally required.

The **additionalProperties: false** field means that if the metadata blob is validated against the global schema, all properties in the blob must correspond to properties defined in the properties section of the global schema. Any unknown properties in the metadata blob (say, `"foo" : "bar"`) will result in a schema validation error.

properties

The properties section defines the global set of properties allowable in the metadata blob. It is straightforward JSON schema.

options

The options section of the global schema defines Alpaca options for each field. These values influence the display characteristics of each form field. Individual schemas that reference the global options will display form fields in a consistent manner.

Individual schemas

Individual schemas list the properties to be displayed in Alpaca forms, and possibly define additional schema validation constraints.

An example hypothetical individual schema is as follows. It defines a form containing ONE field, `journal-NLMTA-ID`, but defines additional prerequisites - fields on the metadata blob that, while not input by this particular form, nevertheless must be present in order to successfully validate. In particular, this schema requires that `authors` be present. One reason for doing this is if we wanted an NIHMS form to follow a "common" form, in which several fields have already been defined and filled out, and we (for whatever reason) don't wish to repeat the fields on the NIH-specific form. For now, though, just focus on the parts of the schema and how they work

```
{
   "title": "NIHMS schema",
   "type": "object",
   "$schema": "http://json-schema.org/draft-07/schema#",
   "definitions": {
       "form": {
           "title": "This is the title Alpaca displays in the UI",
           "type": "object",
           "required": ["journal-NLMTA-ID"],
           "properties": {
               "journal-NLMTA-ID": {"$ref":
"http://localhost:8080/schemas/global.json#/properties/journal-NLMTA-ID"}
           },
          "options": {"$ref": "http://localhost:8080/schemas/global.json#/options"}
       },
       "prerequisites": {
           "title": "prerequisites",
           "type": "object",
           "required": ["authors"],
           "properties": {
               "authors": {"$ref": "http://localhost:8080/schemas/global.json#/properties/authors"}
           }
       }
   },
   "allOf": [
       {"$ref": "http://localhost:8080/schemas/global.json#"},
       {"$ref": "#/definitions/prerequisites"},
```

```
        {"$ref": "#/definitions/form"}
    ]
}
```

The schema is divided into three sections, colored red, green, and blue. These designate different parts of the schema that have different functions

/definitions/form

The "form" field of "definitions" contains the schema that will be displayed by Alpaca. It MAY designate certain fields as required (in this example, journal-NLMTA-ID is required), and MUST contain property names that correspond to properties defined in the global schema. Each property SHOULD reference the global schema property via a JSON reference ($ref). All fields defined here MUST be displayed by Alpaca in the Ember UI.

This is a convention defined by PASS for the purpose of explicitly demarcating which forms are intended to be displayed. The contents of /definition/form is a valid JSON schema as defined by the JSON schema spec.

/definitions/form/options

Individual schemas SHOULD define a /definitions/form/options property containing Alpaca-specific field definitions. These contain labels for display, formatting information (e.g. text area height), and custom validation and/or behavior not specifically defined by JSON schema. In general, for the sake of consistency, the global schema defines a global set of options, and individual schemas SHOULD reference i with a JSON reference. That being said, it is fine for an individual form to customize the appearance of firm fields by defining their own set of options

The location of the "options" property is a convention defined by PASS. The contents and semantics of the "options" object are defined by Alpaca

/definitions/prerequisites

The "prerequisites" field of "definitions" defines a schema that is NOT displayed in screen. Its sole purpose is for introducing validation constraints for metadata that is present in the blob, but not populated by the forms being displayed by this schema. In the example, the "authors" field is required to be populated. Schemas MAY have a prerequisites section if they have this additional validation need, but may omit if if not. If validation with respect to this prerequisites is desired, the schema author should reference this section in the `allOf` section of the schema

The naming of this section is a convention for schema writers. Ember will not specifically look at this section, and it is possible to name this section something else. Only by referencing this

section in the allOf section of the schema will this section be used for validation, as part of the normal JSON schema validation spec

/allOf

allOf is defined by JSON schema as a list of schemas (or references to schemas) that MUST be satisfied in order for the data to be considered valid with respect to this schema. In the example shown, data must be valid with respect to three schemas, included by reference:

- The global schema
- The form schema
- The prerequisites schema

It is up to the author to decide the set of schemas to validate against. Alpaca only validates against the form schema. Individual schemas SHOULD validate against the global schema as well. In any case, Ember SHOULD display a warning and MUST prevent submission (but possibly allow saving, in the case of proxy submission if incomplete metadata) of the metadata blob. The validation MUST be against the contents of the metadata blob after all form data from a given form has been merged into it.

Extending schemas with additional information

Additional properties may easily be added to /definitions in a schema without affecting this proposal. For example, suppose one wishes to include mapping information from JSON fields to metadata fields in a particular format, for a particular repository. This could be as easy as placing it in /definitions/mappings

```
{
    "definitions": {
         "form": {
             // form stuff
         },
         "options" : {$ref: whatever},
         "mappings": {
             "repository": "JScholarship",
             "fields": {
                 "title": "dc.title",
                 // etc
             }
         }
    }
}
```

Concrete proposals for specific extensions is out of scope for this document

Implementation notes

Using Alpaca

As mentioned earlier, forms intended to be displayed in individual schemas are present in **/definitions/form**. This location is a convention defined by the PASS specification. Therefore, Ember must retrieve the contents of this section of the schema, and send it to Alpaca for rendering. Likewise, the same must be done with Alpaca's options. Furthermore, Alpaca must be provided the metadata blob in order to pre-populate fields according to pre-existing content This may be done as follows:

```
var blob = getMetadataBlob() // Get the pre-existing metadata blob somehow
var schema = getSchemaFromSchemaService()  // get the schema somehow

var form = {
   "data": data,
   "schema": schema.definitions.form,
   "options: schema.definitions.options
}

// Have Alpaca render the form
$("#whateverTheFormIs').alpaca(form)
```

Merging and validation.

Any operation that updates the contents of the metadata blob (including Alpaca forms) MUST merge the new contents with the existing contents of the blob. For example, using JQuery:

```
$.extend( true, metadataBlob, alpacaFormData )
validationResults = validator.validate(metadataBlob, alpacaSchema)
```

This takes the content of the javascript object `alpacaFormData`, and merges it into `metadataBlob`.

Schema validation should occur after relevant content has been merged in to the blob. It can occur immediately after entering in the form data, or as a final step prior to submission. In the example above, `alpacaSchema` was the individual schema used to generate the form that produced `alpacaFormData`, and validation compared the resulting merged contents of `metadataBlob` with the individual schema.

Furthermore, please note that *Alpaca mutates the schemas it is given, in ways that are incompatible with the JSON schema specification*. Therefore, if you are validating a schema immediately after having displayed that schema in Alpaca, it is important to make a defensive copy - either of the schema given to Alpaca, or the validator. For example:

```
var safeSchemaCopy = JSON.parse(JSON.stringify(schema))
var form = {
   "data": data,
   "schema": schema.definitions.form,
   "options: schema.definitions.options
}

// This will taint the contents of 'schema'
$("#whateverTheFormIs').alpaca(form)

// This will validate OK
validationResults = validator.validate(metadataBlob, safeSchemaCopy)
```

Dereferencing

Alpaca has some ability to dereference any $ref that appears in the schema, but this ability is limited. Most notably, Alpaca cannot include parts of external documents (for example `"$ref": "http://localhost:8080/schemas/global.json#/properties/authors"`). To work around this either the schema service needs to serve already dereferenced schemas (where the contents of $ref are replaced with their dereferenced content), or Ember will need to include a dependency such as `json-schema-deref` which will dereference all $refs before handing the schema to Alpaca for rendering.

Strategies

For each individual schema that defines forms in /definitions/form, Ember shall display a form. This means that the number and ordering of schemas produced by the schema service significantly affects the user experience. Some approaches for producing a list of schemas include:

- One single form. The schema service produces a single individual schema, based on merging the required and optional fields from the list of repositories applicable to a given deposit

- One form per repository. This is perhaps the simplest approach implementation-wise, where each repository has a single schema associated with it, which enumerates the optional or required fields for the repository.
- A common form, with additional forms to capture repository-specific metadata. This resembles the current approach. It would involve designing a "common" schema, and individual schemas for each repository. Care and coordination is necessary in this case for maintaining the common schemas and individual schemas in lock-step.

Advocating for a particular strategy, and designing a schema service to implement that strategy, is out of scope for this document.

Demo

There is a demo on github at https://github.com/birkland/schema-demo
This shows a simple alpaca form based on schemas using the pattern described in the doc.
You are encouraged to clone it, and play with the schemas or data!