**Relationship metadata in NIEM**

Metadata is data about data. The distinction is created by intended use. To the person editing an image, the creation timestamp is metadata, something he does not need. To the person writing software to sort photos into creation order, the timestamp is the data for his code. One man's metadata is another man's data.

In NIEM, metadata is a mechanism allowing a message producer to add properties that were ignored or considered unimportant by a model designer, and to do this without altering the object type in the original model. The NIEM 5.0 mechanisms work for XML messages, but not for JSON. This paper describes the problem and walks through potential alternatives.

For XML messages, NIEM defines metadata attributes that can be applied to any object, each pointing to a metadata element that contains the additional properties. The example below illustrates metadata in NIEM XML. (In all examples, some components are omitted for simplicity and clarity.)

Suppose we have a model for items, and people who own items, with messages like this:

<Item s:id="map"><Name>Marauder’s Map</Name></Item>

<Item s:id="bird"><Name>Hegwid</Name></Item>

<Person>

<Name>Harry Potter</Name>

<OwnsItem s:ref="map"/>

<OwnsItem s:ref="bird"/>

Example 1: Sample NIEM XML message

The designer of this model made no provision for privacy concerns. In this example, we use metadata to mark portions of the message as private. A policy enforcement point could then use our privacy metadata to grant or withhold access as appropriate. With metadata applied, our message looks like this:

<Item s:id="map"><Name>Marauder’s Map</Name></Item>

<Item s:id="bird"><Name>Hegwid</Name></Item>

<Person>

<Name>Harry Potter</Name>

<OwnsItem s:ref="map" s:metadata="pmd"/>

<OwnsItem s:ref="bird" s:relationshipMetadata="pmd"/>

<Metadata s:id="pmd">

<Private>true

Example : NIEM XML with metadata

There are two metadata attributes in this example, with different interpretations. The attribute s:metadata applies metadata to an *object*. In this example, the very existance of the Marauder's Map is marked private. The attribute s:relationshipMetadata applies metadata to a *relationship.* In this example, anyone may know that Hegwid *exists*; only the fact that *Harry* *owns* Hegwid is private.

Let us first consider metadata applied to an object. NIEM 5.0 defines the mapping from a NIEM XML message to its RDF equivalent. The RDF interpretation of the above message (ignoring s:relationshipMetadata for now) is a graph of tuples:

\_:b0 <Item> \_:b1 .

\_:b0 <Item> \_:b2 .

\_:b0 <Person> \_:b3 .

\_:b0 <Metadata> \_:b4 .

\_:b1 <Name> "Marauder's Map" .

\_:b1 <Metadata> \_:b4 .

\_:b2 <Name> "Hegwid" .

\_:b3 <Name> "Harry Potter" .

\_:b3 <OwnsItem> \_:b1 .

\_:b3 <OwnsItem> \_:b2 .

\_:b4 <Private> "true" .

Example : RDF interpretation of object metadata

The object corresponding to the Marauder's Map is the blank node \_:b1. The metadata element becomes a property of this object. That is the desired interpretation.

Every NIEM XML message has an equivalent in NIEM JSON. Messages are equivalent when they correspond to the same RDF graph. The NIEM JSON equivalent to the above example looks like this:

{

"@context": "https://example.com/MyMessageFormat/",

"my:Item" : [

{

"@id": "map",

"my:Name": "Marauder's Map"

},

{

"@id": "bird",

"my:Name": "Hegwid"

}

],

"my:Person": {

"my:Name": "Harry Potter",

"my:OwnsItem": [

{

"@id": "map",

"s:metadata": { @id": "pmd" }

},

{ "@id": "bird" }

]

},

"pr:Metadata": {

"@id": "pmd",

"pr:Private": "true"

}

}

Example : NIEM JSON equivalent with object metadata

NIEM JSON messages are JSON-LD, and the mapping to RDF is defined by the JSON-LD specification. However, it is possible to process these messages as plain JSON, which is what many developers will expect.

Metadata applied to objects is something NIEM does well. The XML and JSON serializations are convenient for message production and consumption. Alas, when we consider metadata applied to relationships, things do not work out so well. Let's look at the RDF interpretation of our sample message, this time ignoring s:metadata and concentrating on s:relationshipMetadata. The mapping defined by NIEM 5.0 is a RDF dataset containing two graphs; one, the default graph, as follows:

\_:b0 <Item> \_:b1 .

\_:b0 <Item> \_:b2 .

\_:b0 <Person> \_:b3 .

\_:b0 <Metadata> \_:b4 .

\_:b1 <Name> "Marauder's Map" .

\_:b2 <Name> "Hegwid" .

\_:b3 <Name> "Harry Potter" .

\_:b3 <OwnsItem> \_:b1 .

\_:b4 <Private> "true" .

Example : Default RDF graph for relationship metadata

The second graph is has the metadata object as its label, and contains all of the tuples to which the metadata object applies – specifically, the tuple representative the ownership relationship from Harry to Hegwid.

\_:b3 <OwnsItem> \_:b2 \_:b4 .

Example : Labeled RDF graph for relationship metadata

So far, so good, but now we come to the equivalent message in JSON-LD, which appears below:

{

"@context": "https://example.com/MyMessageFormat/",

{

"@graph": [

{

"@id": "bird"

},

{

"@id": "\_:b3",

"my:OwnsItem": {

"@id": "bird"

}

}

],

"@id": "pmd"

},

{

"@graph": [

{

"@id": "\_:b0",

"my:Item": [

{ "@id": "map" },

{ "@id": "bird" }

],

"pr:Metadata": {

"@id": "pmd"

},

"my:Person": {

"@id": "\_:b3"

}

},

{

"@id": "pmd",

"pr:Private": {

"@value": "true"

}

},

{

"@id": "bird",

"my:Name": "Hegwid"

},

{

"@id": "map",

"my:Name": "Marauder's Map"

},

{

"@id": "\_:b3",

"my:Name": "Harry Potter",

"my:OwnsItem": {

"@id": "map"

}

}

],

"@id": "default"

}

}

Example : NIEM JSON with relationship metadata

Interpreting that message as plain JSON is not feasible. Developers will rebel. And yet, we have no choice. Relationship metadata must be interpreted as a multi-graph RDF dataset. The JSON-LD representation of multi-graph datasets is not under our control. The resulting JSON message is unacceptable. We must look for an alternative representation of relationship metadata.

*Alternative #1:* We could eliminate the s:relationshipMetadata attribute, and tell message designers that if they want metadata properties on a relationship, then they must build arelationship object holding those properties into their message schema. (We could tell them to use an association object, but that seems like overkill.) The NIEM XML message would look like this:

<Item s:id="map"><Name>Marauder’s Map</Name></Item>

<Item s:id="bird"><Name>Hegwid</Name></Item>

<Person>

<Name>Harry Potter</Name>

<OwnsItem s:ref="map"/>

<OwnsItemRelation>

<Metadata s:ref="pmd"/>

<OwnsItem s:ref="bird"/>

<Metadata s:id="pmd">

<Private>true

Example : NIEM XML with explicit relationship object

The RDF interpretation is:

\_:b0 <Item> \_:b1 .

\_:b0 <Item> \_:b2 .

\_:b0 <Person> \_:b3 .

\_:b0 <Metadata> \_:b5 .

\_:b1 <Name> "Marauder's Map" .

\_:b2 <Name> "Hegwid" .

\_:b3 <Name> "Harry Potter" .

\_:b3 <OwnsItem> \_:b1 .

\_:b3 <OwnsItemRelation> \_:b4 .

\_:b4 <OwnsItem> \_:b2 .

\_:b4 <Metadata> \_:b5 .

\_:b5 <Private> "true" .

Example : RDF interpretation of relationship object

And the NIEM JSON equivalent is:

{

"@context": "https://example.com/MyMessageFormat/",

"my:Item" : [

{

"@id": "map",

"my:Name": "Marauder's Map"

},

{

"@id": "bird",

"my:Name": "Hegwid"

}

],

"my:Person": {

"my:Name": "Harry Potter",

"my:OwnsItem": { "@id": "map" },

"my:OwnsItemRelation": {

"s:metadata": { @id": "pmd" }

"my:OwnsItem": { "@id", "bird" },

}

},

"pr:Metadata": {

"@id": "pmd",

"pr:Private": "true"

}

}

Example : NIEM JSON with relationship object

The RDF is simple enough, and we get an acceptable JSON representation. On the other hand, we give up the ability to add metadata properties without changing the original type in the message schema. Also, applying relationship metadata to objects from the NIEM model is not really possible. Suppose you want to add metadata to the relationship between nc:Person and nc:PersonName, like this:

<nc:Person>

<nc:PersonName>

<nc:PersonFullName>Clark Kent

<my:PersonNameRelation>

<my:Metadata s:ref="pmd">

<nc:PersonName>

<nc:PersonFullName>Superman

Example : Adding relationship metadata to nc:PersonName

Alas, nc:PersonType does not include the my:PersonNameRelation element you need. And you can't just add it to your message schema, because that breaks the subset rule. So you would have to define your own version of nc:PersonType in your extension schema. Blech. I don't think we are choosing this one.

*Alternative #2:* We could retain the s:relationshipMetadata attribute in XML data, but change the RDF interpretation and the NIEM JSON equivalent, defining these to be what you get with an explicit relationship element. For example, given the NIEM XML in example 2, we would just declare that example 9 is the RDF, and example 10 the JSON. In short:

|  |  |  |
| --- | --- | --- |
| <OwnsItem s:ref="bird" s:relationshipMetadata="pmd"/> | \_:b3 <OwnsItemRelation> \_:b4 .  \_:b4 <OwnsItem> \_:b2 .  \_:b4 <Metadata> \_:b5 . | "my:OwnsItemRelation": {  "s:metadata": {  @id": "pmd"  },  "my:OwnsItem": {  "@id", "bird"  }  } |
| *when you see this XML* | *you interpret as this RDF* | *and have this JSON equivalent* |

I call this the "NDR shenanigans" approach. We would have to do a lot of work in the NDR, defining the implied FooRelation and FooRelationType components. We get a good JSON representation, plus the ability to add metadata without changing the message model – at least for XML messages. However, the model defined by XML Schema and the model instance for JSON would now be different; the latter would have the OwnsItemRelation property, the former would not. That's a difficulty as long as we are using XML Schema as a model definition language. I don't think we will choose this alternative, either.

*Alternative #3:* We could add FooRelation and FooRelationType for *every property* in the NIEM core and domain models. We get a good JSON representation, we can add relationship metadata without changing the original type definitions, and we get the same message model for XML and JSON. We pay for that with a vast number of new elements and complex types in the reference schema documents. (That might not be so bad if our tools hid those new components until they are wanted.)

For implementation, I am imagining some new components in the structures namespace:

<xs:element name="RelationMetadata" type="s:MetadataType" abstract="true"/>

<xs:complexType name="RelationType" abstract="true">

<xs:sequence>

<xs:element ref="s:RelationMetadata" minOccurs="0" maxOccurs="unbounded"

<xs:attribute ref="s:id"/>

<xs:attribute ref="s:ref"/>

<xs:attribute ref="s:uri"/>

And I imagine three new rules in the NDR:

Rule: An element named *Foo*Relation must have type *Foo*RelationType

Rule: An element of *Foo*RelationType must be named *Foo*Relation

Rule: A type named *Foo*RelationType must be defined as follows:

<xs:complexType name="*Foo*RelationType">

<xs:complexContent>

<xs:extension base="s:RelationType>

<xs:sequence>

<xs:element ref="*Foo*">

When it's time to define specific relationship metadata properties in your extension schema, you do it like this:

<xs:element name="RelationMetadata" type="my:RelationMetadataType"

substitutionGroup="s:RelationMetadata"/>

<xs:complexType name="RelationMetadataType">

<xs:complexContent>

<xs:extension base="s:MetadataType">

<xs:sequence>

<xs:element ref="my:FirstMetadataProperty"…

And then you can put your RelationMetadata property on any FooRelation element, like this:

<nc:Person>

<nc:PersonNameRelation>

<my:RelationMetadata s:ref="md" xsi:nil="true"/>

<nc:PersonFullName>Superman

That metadata element works just as well as a metadata attribute. Of course, if a message can include relationship metadata, then the people writing code have to look for the PersonNameRelation element. But they already are supposed to look for the relationship metadata attribute.

What's good: We get a good JSON equivalent, we get the ability to add relationship metadata without really changing the message model, we get a single message model for all serializations. What's bad: The NIEM reference schemas become a lot longer and more complex. I suppose we might possibly choose this alternative.

*Alternative #4:* Wait on implementing relationship metadata until we have metamodel tooling. I like this one a lot. Most of the problems above go away when we use the metamodel as the modeling language for the NIEM model, and generate XML schema as needed for each message specification.

I imagine a checkbox in a modeling tool that says "can have relationship metadata". If the message designer checks the box on nc:PersonName, the tool spits out schema for nc:PersonType schema that includes nc:PersonNameRelation; leave the block empty and you get the usual nc:PersonName instead.

You can't do that today because of our rules that say (a) a reference schema document, once published, cannot be changed, and (b) a schema document for a model namespace in a message specification must be a subset of the reference schema document. We have those rules because of the way we use XML Schema as the modeling language for the core and domain models, and how we create message models by composing subsets of those schema documents. When we have metamodel tooling, we won't need those rules. The results will look like this:

<nc:Person> *runtime NIEM XML message*

<nc:PersonNameRelation>

<nc:PersonName>

<nc:PersonFullName>Clark Kent

<my:PersonNameRelation>

<my:Metadata s:ref="pmd">

<nc:PersonName>

<nc:PersonFullName>Superman

<my:Metadata s:id="pmd">

<my:Private>true

<Model> *model instance*

<Property>nc:Person

<Property>nc:PersonName

<Property>nc:PersonFullName

<Property>my:Metadata

<Property>my:Private

<Type>PersonType

<HasProperty>PersonName

<MinOccurs>0

<MaxOccurs>unbounded

<RelationshipMetadataIndicator>true

<xs:complexType name="PersonType"> *generated niem-core.xsd schema document*

<xs:sequence>

<xs:element ref="nc:PersonNameRelation" minOccurs="0" … >

<xs:complexType name="PersonNameRelationType">

<xs:extension base="s:RelationType">

<xs:sequence>

<xs:element ref="nc:PersonName"/>

{ *NIEM JSON representation*

"nc:Person": {

"nc:PersonNameRelation": [

{

"nc:PersonName": {

"nc:PersonFullName": "Clark Kent"

}

},

{

"my:Metadata": { "@id": "pmd" },

"nc:PersonName": {

"nc:PersonFullName": "Superman"

}

}

],

"my:Metadata": {

{ "@id": "pmd" },

{ "my:Private": "true" }

**Conclusion:** The NIEM 5.0 mechanism for relationship metadata doesn't work for JSON. A different mechanism for NIEM 6.0 is needed, one that works for all message serializations.