# Resource Prototypes

*Bruno Jouhier*
*26 March 2012*

## Introduction

This document proposes a new approach to metadata handling for SData 2.0, based on prototypes and value templates.

This approach is intended to fix the most blatant shortcomings of SData 1.X schemas; basically their complexity and their static nature. It also improves the situation with payload size.

This document uses the JSON syntax to describe the proposal. This does not mean that the proposal is restricted to JSON. It should be relatively straightforwards to derive an XML mapping from this proposal but this is beyond the scope of this document.

This document just provides an introduction to prototypes and value templates. It does not cover all the details. Detailed specs will come later.

### *Prototypes and Value Templates*

The core idea behind this prototype proposal is to have a single format that encompasses both data and metadata, instead of having two separate formats like we had before (Atom for data, XSD for metadata).

The complete resource with all its metadata and data is obtained by merging the data object with the prototype object.

The value template feature is a further refinement which makes it possible to include variables in prototype values (URLs but also titles, etc.). Thanks to this, all the heavy markup carried by a resource can be described at the prototype level, with variables that refer to properties of the object. This leads to a drastic reduction of the data payload. The idea is to put all the bloat in the prototype, which will be transferred once and cached, and keep the data payloads as lean as possible.

Let us start with a simple JSON data entry for a single resource:

```
GET /sdata/myApp/myContract/_/salesOrder('43660') HTTP/1.1

200 OK
Content-Type: application/json; vnd=sage.sdata

{
  "$prototype": "http://www.example.com/sdata/myApp/myContract/-/$prototypes('salesOrders.
$details')",
  "$updated": "2008-03-31T13:46:45Z",
  "id": "43660",
  "orderDate": "2001-07-01",
  "shipDate": null,
  "contact": {
    "id": "216"
  },
  "subTotal": 1553.10
}
```

The prototype for this entry will be a JSON object like the following:

```
GET /sdata/myApp/myContract/_/$prototypes('salesOrders.$details') HTTP/1.1

200 OK
Content-Type: application/json; vnd=sage.sdata

{
  "$baseUrl": "http://www.example.com/sdata/myApp/myContract/-",
  "$url": "{$baseUrl}/salesOrders('{id}')",
  "$key": "{id}",
  "$title": "Sales Order {id}",
  "$properties": {
    "id": {
      "$title": "ID",
      "$type": "application/x-integer",
      "$isMandatory": true
    },
    "orderDate": {
      "$title": "Date",
      "$type": "application/x-date",
      "$isMandatory": true
    },
    "shipDate": {
      "$title": "Shipping Date",
      "$type": "application/x-date",
    },
    "contact": {
      "$title": "Contact",
      "$type": "application/x-reference",
      "$url": "{$baseUrl}/contacts('{id}')",
      "$key": "{id}",
      "$properties": {
        "id": {
          "$type": "application/x-string"
        }
      }
    },
    "subTotal": {
      "$title": "Subtotal",
      "$type": "application/x-decimal"
    }
  },
  "$links": {
    "$print": [{
      "$title": "Print",
      "$url": "{$url}",
      "$type": "application/pdf"
    }],
    "$update": [{
      "$title": "Update",
      "$url": "{$url}",
      "$method": "PUT"
    }],
    "$delete": [{
      "$title": "Delete order {id}?",
      "$url": "{$url}",
      "$method": "DELETE"
    }],
    "createBOM": [{
      "$title: "Create Bill of Material",
      "$url": "{$url}/$service/createBOM",
      "$method": "POST"
    }]
  }
}
```

The prototype may look a bit overwhelming at first sight but it carries a lot of information. It describes all the properties of the resource, their data type, their title, their constraints (`$isMandatory`), etc. It also defines the links that all resources will have. In the example above, the salesOrder object will have a reference to a contact resource, as well as "Print", "Update", "Delete" and "Create BOM" links.

The complete resource is obtained by merging the prototype and the individual resource, and by substituting the {…} variables in the values, which gives:

```
{
  "$baseUrl": "http://www.example.com/sdata/myApp/myContract/-",
  "$url": "http://www.example.com/sdata/myApp/myContract/-/salesOrders('43660')",
  "$key": "43660",
  "$title": "Sales Order 43660",
  "$prototype": "http://www.example.com/sdata/myApp/myContract/-/$prototypes('salesOrder.
$details')",
  "$updated": "2008-03-31T13:46:45Z",
  "id": "43660",
  "orderDate": "2001-07-01",
  "shipDate": null,
  "contact": {
    "$title": "Contact",
    "$type": "application/x-reference",
    "$url": "http://www.example.com/sdata/myApp/myContract/-/contacts('216')",
    "$key": "216",
    "id": "216"
  },
  "subTotal": 1553.10
  "$properties": { ... }
  "$links": [{
    "$print": {
      "$title": "Print",
      "$url": "http://www.example.com/sdata/myApp/myContract/-/salesOrders('43660')",
      "$type": "application/pdf"
    }],
    "$update": [{
      "$title": "Update",
      "$url": "http://www.example.com/sdata/myApp/myContract/-/salesOrders('43660')",
      "$method": "PUT"
    }],
    "$delete": [{
      "$title": "Delete order 43660?",
      "$url": "http://www.example.com/sdata/myApp/myContract/-/salesOrders('43660')",
      "$method": "DELETE"
    }],
    "createBOM": [{
      "$title: "Create Bill of Material",
      "$url": "http://www.example.com/sdata/myApp/myContract/-/salesOrders('43660')/
$service/createBOM",
      "$method": "POST"
    }]
  }
}
```

Note: the `$properties` block has been omitted from the merged result for clarity.

## *Overriding*

Prototypes are much more dynamic than schemas because prototype elements can be overridden by data elements. For example:

```
{
  "$prototype": "http://www.example.com/sdata/myApp/myContract/-/$prototypes('salesOrders')",
  "$updated": "2008-03-31T13:46:45Z",
  "id": "43660",
  "orderDate": "2001-07-01",
  "shipDate": null,
  "contact": {
    "id": "216"
  },
  "subTotal": 1553.10,
  "$properties": {
    "contact": {
      "$title": "Contact (mandatory)",
      "$isMandatory": true
    }
  },
  "$links": {
    "$delete": null
  }
}
```

This sales order resource overrides several prototype elements:

- The definition of the `contact` property: it make is mandatory and it changes the `$title` to reflect this.
- The `$delete` link: it removes this link by setting it to `null`. This tells the consumer that the sales order cannot be deleted. The same effect could have been achieved the other way around, by omitting the `$delete` link in the prototype and adding it to the data only if the resource can be deleted.

This overriding mechanism allows us to cope with applications that have dynamic metadata. The idea is to put all the default values in the prototype and to let individual resources override when the default does not apply to them.

## *Granularity and Modularity*

The first example above shows a prototype for a single entry. But what about a feed of entries? Will the feed use the same prototype, or a different one?

With SData 1.1 schemas, the metadata was centralized in a schema and was articulated around "Resource Kinds". The schema captured all the metadata for all the RKs, and thus all the possible requests, in a given contract.

The prototype proposal takes a much more modular approach: a prototype carries the metadata for a given request (a given URL) or a set of similar requests (a family of URLs). In the example above, the `salesOrder.$details` prototype carries the metadata for all requests on `http://www.example.com/sdata/myApp/myContract/-/salesOrders('{id}')` URLs but that's all. A query on `http://www.example.com/sdata/myApp/myContract/-/salesOrders`, the RK's collection URL, will point to a different prototype. For example:

```
GET /sdata/myApp/myContract/_/salesOrders HTTP/1.1

200 OK
Content-Type: application/json; vnd=sage.sdata

{
  "$prototype": "http://www.example.com/sdata/myApp/myContract/-/$prototypes('salesOrders.
$query')",
  "$updated": "2008-03-31T13:46:45Z",
  "$startIndex": 0,
  "$itemsPerPage": 20,
```

```
  "$totalResults": 24173,
  "$resources": [{
    "id": "43660",
    "orderDate": "2001-07-01",
    "shipDate": null,
    "contact": {
      "id": "216"
    },
    "subTotal": 1553.10
  }, {
    "id": "43661",
    "orderDate": "2001-07-04",
    "shipDate": null,
    "contact": {
      "id": "035"
    },
    "subTotal": 812.00
  }, {
    ...
  }]
}
```

The prototype is now `salesOrder.$query` rather than `salesOrder.$details`. It typically looks like:

```
GET /sdata/myApp/myContract/_/$prototypes('salesOrders.$query') HTTP/1.1

200 OK
Content-Type: application/json; vnd=sage.sdata

{
  "$baseUrl": "http://www.example.com/sdata/myApp/myContract/-",
  "$url": {$baseUrl}/salesOrders",
  "$properties": {
    "$resources": {
      "$type": application/x-collection",
      "$item": {
        "$url": "{$baseUrl}/salesOrders('{id}')",
        "$key": "{id}",
        "$title": "Sales Order {id}",
        "$properties": {
          "id": {
          "$title": "ID",
          "$type": "application/x-integer",
          "$canSort": true
        },
        "orderDate": {
          "$title": "Date",
          "$type": "application/x-date",
          "$canSort": true
        }, {
          ...
        },
        "contact": {
          "$title": "Contact",
          "$type": "application/x-reference",
          "$url": "{$baseUrl}/contacts('{id}')",
          "$key": "{id}",
          "$properties": {
            "id": {
              "$type": "application/x-string"
            }
          }
        }, {
          ...
        }
```

```
      },
      "$links": {
        "$print": [{
          "$title": "Print",
          "$url": "{$url}",
          "$type": "application/pdf"
        }],
          ...
        "createBOM": [{
          "$title: "Create Bill of Material",
          "$url": "http://www.example.com/sdata/myApp/myContract/-/salesOrders('43660')/
$service/createBOM",
          "$method": "POST"
        }]
      }
    }
  },
  "$links": {
    "$print": [{
      "$title": "Print",
      "$url": "{$url}",
      "$type": "application/pdf"
    }],
    "$template": [{
      "$title": "Create",
      "$url": "{$url}/$template",
    }]
  }
}
```

A few points are worth mentioning here:

- The prototype describes an object with a `$resources` property of type `application/x-collection`. This is logical because it describes a feed, not a single entry.
- The entry structure is described by the `$item` member of the `$resources` property.
- The property descriptors (`id`, `orderDate`, etc) do not carry `$isMandatory` attributes any more. This is logical because the prototype does not contains any link with PUT or POST methods. The whole feed is read-only. On the other hand, the property descriptors carry `$canSort` attributes because sorting is relevant in this context.
- The `contact` property is a reference to a contact, like in the `$details` prototype, and all the boilerplate markup for the contact is inside the prototype rather than inside the feed.
- `$links` are present both at the entry level (first "Print" link) and the feed level (second "Print" and "Create" links). The prototype describes all the operations that are available at all levels. The data feed just provides the variables that will be substituted in the URLs.

So, to summarize:

- Prototypes describe a URL (`…/salesOrders`) or a family of URLs (all the `…/salesOrders('{id}')`).
- There are several prototypes for a given Resource Kind. One for each "facet" (`$query`, `$details`, …) of the RK.
- A prototype does not describe an entity, it describes the resource (entry or feed) which is accessed through a given URL or family of URLs.
- Every resource response contains a `$prototype` member with the URL of its prototype (assuming that the contract mandates prototypes).

## *Discoverability and Links*

The discoverability mechanism is very different than with schemas. Instead of discovering

everything at once though a global schema, the consumer discovers the operations that are available progressively. Every prototype contains links for all the operations that are available on the resource (entry or feed).

Schemas gave us some kind of static map of all the operations that were available in the contract. With prototypes we have instead a more dynamic exploration where each step gives us a local map of the operations that we can perform on the current resource.

Schemas were problematic because it was impossible to express all the subtle conditions that define which operations are allowed and which ones are not in different contexts. We only had a small set of static flags (`canGet`, `canPut`, …) at the RK level. Prototypes solve this problem: the consumer does not rely on a static map any more; instead, each response contains links to all the available operations (via the prototype, which may be overridden). This is much more dynamic and modular.

Discoverability relies on `$links`. It is important to have standard link names, so that service consumers can immediately find the links for standard operations. For example:

- To create a new resource, the consumer would first follow the `$template` link to load a template resource, and then a `$create` link to save the modified template. Note that the `$create` link is provided by the template's prototype.
- To update an existing resource, the consumer would just load the `$details` entry and follow the `$update` link to save the modified resource.

But there may also be non-standard (contract specific) links. For example the `createBOM` link in the example above. The `$` sign prefix distinguishes standard links from contract-specific ones.

Note: There could be situations were a link leads to alternatives, for example different `$print` links for different media types (different `$type` values). This is why every link name points to an array rather than to a single object. This is a bit heavy and maybe we could solve the problem differently (by chaining the alternatives with a `$altLink`) as multiple links seems to be the exception rather than the norm.

## *From schemas to prototypes*

At this point, I'd like to quickly review the metadata elements that we had in SData 1.1. schemas and explain how they will be captured in prototypes.

### Resource Kind Definitions

The following XSD schema:

```
<xs:element name="salesOrder" type="tns:salesOrder-type"
    sme:role="resourceKind" sme:pluralName="salesOrders" sme:label="Sales Order"
    sme:canGet="true" sme:canPost="true" sme:canPut="true" sme:canDelete="true"
    sme:hasTemplate="true"
    sme:canPageNext="true" sme:canPagePrevious="true" sme:canPageIndex="true"
    sme:hasUuid="true" sme:supportsETag="true" sme:batchingMode="syncOrAsync" />
  <xs:complexType name="salesOrder-type">
    <xs:all>
      <xs:element name="orderNumber" type="xs:string" minOccurs="0"
                  sme:label="#" sme:canSort="true" sme:canFilter="true" sme:precedence="1"
                  sme:isUnique="true" sme:isReadOnly="true"/>
      <xs:element name="orderDate" type="xs:date" minOccurs="0"
                  sme:label="Date" sme:canSort="true" sme:canFilter="true" sme:precedence="2" />
      <xs:element name="shipDate" type="xs:date" minOccurs="0" nillable="true"
                  sme:label="Shipping Date" sme:canSort="true" sme:canFilter="true"
sme:precedence="3" />
```

```
      <xs:element name="subTotal" type="xs:decimal" minOccurs="0"
                  sme:label="Sub-total" sme:canSort="true" sme:canFilter="true" sme:precedence="2"
                  sme:isReadOnly="true" />
      <xs:element name="billAddress" type="tns:address--type" minOccurs="0"
                  sme:relationship="child" sme:isCollection="false" sme:label="Billing Address"
                  sme:canGet="true" sme:canPut="true" />
      <xs:element name="shipAddress" type="tns:address--type" minOccurs="0"
                  sme:relationship="child" sme:isCollection="false" sme:label="Shipping Address"
                  sme:canGet="true" sme:canPut="true" />
      <xs:element name="orderLines" type="tns:salesOrderLine--list" minOccurs="0"
                  sme:relationship="child" sme:isCollection="true" sme:label="Order Lines"
                  sme:canGet="true" sme:canPost="true" />
      <xs:element name="contact" type="tns:contact--type" minOccurs="0"
                  sme:relationship="reference" sme:label="Contact"
                  sme:canGet="true" />
    </xs:all>
  </xs:complexType>
  <xs:complexType name="salesOrder-list">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="salesOrder" type="tns:salesOrder--
type" />
    </xs:sequence>
  </xs:complexType>
```

becomes the following `$details` prototype:

```
{
  "$baseUrl": "http://www.example.com/sdata/myApp/myContract/-",
  "$url": "{$baseUrl}/salesOrders('{orderNumber}')",
  "$title": "Sales Order {orderNumber}",
  "$properties": {
    "orderNumber": {
      "$type": "application/x-string",
      "$title": "#",
      "$isReadOnly": true
    },
    "orderDate": {
      "$type": "application/x-date",
      "$title": "Date",
      "$isMandatory": true
    },
    "shipDate": {
      "$type": "application/x-date",
      "$title": "Shipping Date"
    },
    "subTotal": {
      "$type": "application/x-decimal",
      "$title": "Sub total",
      "$isReadOnly": true
    },
    "billAddress": {
      "$type": "application/json; vnd.sage=sdata",
      "$title": "Billing address",
      "$prototype": "{$baseUrl}/address.$child"
    },
    "shipAddress": {
      "$type": "application/json; vnd.sage=sdata",
      "$title": "Shipping address",
      "$prototype": "{$baseUrl}/address.$child"
    },
    "orderLines": {
      "$type": "application/x-collection",
      "$title": "Lines",
      "$item": {
        "$type": "application/json; vnd.sage=sdata",
        "$prototype": "{$baseUrl}/salesOrderLine.$child"
      }
    },
```

```
    "contact": {
      "$type": "application/x-reference",
      "$url": "{$baseUrl}/contacts('{$key}')",
    }
  },
  "$links": {
    // operations from the schema
  }
}
```

A lot of attributes (`canGet`, `canPost`, …) have disappeared but this is because discoverability is now done very differently. First, the above prototype is only "one" of the prototypes exposed by the `salesOrder` RK (its `$details` prototype). The possible actions are described by the `$links` present in the prototype. If an action is not available on a given RK, there won't be any link for it.

## *"Menu" resources*

Something's lacking at this point, though: the schema was a static map of the contract and allowed the consumer to discover all the entities of the contract. With the new discoverability approach we don't have such a global map. But this can be solved by having the contract URL return a "menu" resource containing only links:

```
GET /sdata/myApp/myContract/- HTTP/1.1

200 OK
Content-Type: application/json; vnd=sage.sdata

{
  "$baseUrl": "http://www.example.com/sdata/myApp/myContract/-",
  "$url": "{$baseUrl}",
  "$title": "My ERP",
  "$links": {
    "customers": [{
      "$url": "{$baseUrl}/customers",
      "$title": "Customers"
    }],
    "salesOrders": [{
      "$url": "{$baseUrl}/salesOrders",
      "$title": "Sales Orders"
    }],
    ...
  }
}
```

Note: this scheme is more flexible than what we had previously. For example, it gives us the freedom to have hierarchical menus: the top level could just have `$links` to "financials", "commercials", "manufacturing", etc. submenus, and each one would have links to its top level entities. It also allows us to have standard links (a `$schema`, with a `text/xml` media type, for example) in the top menu or a service, etc.

## *More on Discoverability and Hypermedia*

Discoverability works very differently from 1.1. The main difference is that the consumer does not need to know the URL syntax rules because it does not build URLs any more. On the other hand, the consumer needs to know the "link names" that it will follow. So we have changed the focus of our standardization effort. There is no need to standardize URL syntax any more. There is no need to standardize most of the "capability" flags we had before (`canGet`, `canPost`, `canPageXxx`, …) because the availability of a given operation is now inferred from the presence of a specific link in the payload rather than from a flag in a schema.

Instead, what really matters now are:

- link names
- property and link types (`$type` attributes).

For `$type` values, the proposal is to use:

- standard internet media types for all properties and links that refer to existing hypermedia types (`application/pdf`, `image`, …)
- `application/json; vnd.sage=sdata` for all structured contents described by metadata (`$properties` and `$links` elements in the prototype). This type is the default one so it can be omitted.
- `application/x-yyy` (`x-boolean`, `x-string`, `x-decimal`, `x-reference`, etc.) for properties that make up structured contents.