OData Extension for JSON Data

A Directional White Paper

Introduction

This paper documents some use cases, initial requirements, examples and design principles for an OData extension for JSON data. It is non-normative and is intended to seed discussion in the OASIS OData TC for the development of an OASIS standard OData extension defining retrieval and manipulation of properties representing JSON documents in OData.

JSON [1] has achieved widespread adoption as a result of its use as a data structure in JavaScript, a language that was first introduced as the page scripting language for Netscape Navigator in the mid 1990s. In the 21st Century, JavaScript is widely used on a variety of devices including mobiles [1], and JSON has emerged as a popular interchange format. JavaScript JSON was standardized in ECMAScript [2]. The JSON Data Interchange Format is described in IETF RFC 4627 [3]

JSON documents were initially stored in databases as character strings, character large objects (CLOBs), or shredded into numerous rows in several related tables. Following in the steps of XML, databases now have emerged with native support for JSON documents such PostGres [4], CouchDB [5], and MongoDB [6]. JSON databases are an important category of Document Databases [7] in NoSQL [8]. One of the main cited attractions of JSON databases is schema-less processing, where developers do not need to consult database administrators when data structures change.

Common use cases for JSON databases include:

- Logging the exchanged JSON for audit purposes
- Examining and querying stored JSON
- Updating stored JSON
- Altering subsequent user experiences in accordance with what was learnt from user exchanges from the stored JSON

Just as the SQL query language was extended to support XML via SQL/XML[9], query languages such as XQuery are evolving to explore support for JSON, e.g., XQilla [10] and JSONiq [11]. XML databases such as MarkLogic [12] offer JSON support.

Note that for document constructs such as XML and JSON, temporal considerations, such as versioning, typically occur at the granularity of the whole document. Concrete examples include versions of an insurance policy, contract, and mortgage or of a user interface.

JSON properties are not currently supported in OData. We suggest that an OData extension be defined to add this support. Properties that contain JSON documents will be identified as such, and additional operations will be made available on such properties.

Status

Version 1.0 (May 18, 2012)

Authors

Ralf Handl, SAP

Susan Malaika, IBM

Michael Pizzo, Microsoft

Background

JSON databases allow JSON collections to be defined implicitly (at first JSON insert) or explicitly, e.g. as in PostGres to create a table with a column of JSON data type:

```
Create table employees
   (empid INTEGER, resume JSON)
```

The advantage of defining a resume as a document, rather than shredding the resume into structured entity properties, is that a resume’s structure can evolve easily, without needing to alter the services data model.

Modeling on MongoDB syntax, operating on a database db and on a named JSON collection employees.resume, a find function can be used to evaluate an expression on JSON documents and return JSON documents that satisfy the expression (filtering). The number of JSON documents returned can be restricted, e.g., via the limit function:

db.employees.resume.find({firstname:"Colleen" }).limit(1)

A find function can be used to evaluate multiple expressions on JSON documents and return JSON documents that satisfy both expressions as results:

db.employees.resume.find({firstname:"Hubert",lastname: "Heijkers" })

Conditions such as greater than (analogous to predicate conditions in SQL WHERE) can be applied in the find function:

db.employees.resume.find({firstname:"Diane", age:{$gt:20}}).limit(1)

The sort function can be used to sort the results. These results are being sorted in ascending order of age:

db.employees.resume.find({lastname:"Eisenberg", age:{$gt:20}}).sort({age:1})

These results are being sorted in descending order of age:

db.employees.resume.find({lastname:"Hartel", age:{$gt:20}}).sort({age:-1})

Options in the find function can be used to select portions of the JSON documents (to project), e.g., to return just the lastnames:

db.employees.resume.find({},{lastname:1})

The filtering and projection options in the find function can be used together, e.g., return just the lastnames of the people whose first name is Asad:

db.employees.resume.find({firstname: "Asad"},{lastname:1})

It is possible to specify that all parts of JSON documents be returned with the exception of a portion. In this example, the JSON resumes are returned for people with firstname Alex, excluding the references portion:

db.employees.resume.find({firstname: "Alex"},{references:-1})

Motivation

An OData service might publish an Employees entity set, with a resume property representing a JSON document:

```
<Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
        Namespace="Personnel">
   <EntityContainer Name="MyCompany">
      <EntitySet Name="Employees" EntityType="Employee"/>
   </EntityContainer>
   <EntityType Name="Employee">
      <Key>
         <PropertyRef Name="empid"/>
      </Key>
      <Property Name="empid" Type="Edm.Int32" Nullable="false"/>
      <Property Name="lastname" Type="Edm.String" Nullable="false"
         MaxLength="30" FixedLength="false" Unicode="true"/>
      <Property Name="resume" Type="Edm.Stream" Nullable="true"
         MaxLength="Max" FixedLength="false"/>
   </EntityType>
</Schema>
```

A client might wish to query this entity set in a number of ways.

1. Retrieve only those employees that have a reference with a lastname “Pizzo” in the references located in their resume.
2. Return every employee, ordering the result based on the state in which they live, where that state is located in their resume.
3. Return the employee identifier and phone number of every employee, where the phone number has been taken from their resume.
4. Replace an old resume with a new resume for a specific employee.
5. Returned summarized resume, just a subset of fields, for each employee.

Requirements

The following capabilities must be supported in this extension to OData:

- An OData Stream data type may be annotated to represent a JSON data type
- JSON properties may be returned separately from non-JSON properties
- Entities may be filtered based on the content of their JSON properties
- JSON values that have been derived from JSON properties may be retrieved
- Scalar values that have been derived from JSON properties may be retrieved
- Find operations may be applied to JSON properties
- The values of JSON properties may be updated

Examples

The following examples describe possible annotations and extensions to OData to support JSON data. Although concrete annotations, functions, and behavior are described, they are intended to be purely illustrative and not prescriptive.

Some notes on the following examples:

- We have avoided using curly braces {} in the OData URLs. Instead we use parentheses ()
- We use a slash / instead of the more usual dot . in JavaScript, to navigate along the JSON structure, because OData uses dot for namespaces
- Functions that return results associated with OData EDM data types have been used, e.g., find_string instead of find, to ensure that function return data type is known

Annotation Example:

The Employees entity set might now be published as:

```
<Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
        Namespace="Personnel">
   <EntityContainer Name="MyCompany">
      <EntitySet Name="Employees" EntityType="Employee"/>
   </EntityContainer>
   <EntityType Name="Employee">
      <Key>
         <PropertyRef Name="empid"/>
      </Key>
      <Property Name="empid" Type="Edm.Int32" Nullable="false"/>
      <Property Name="lastname" Type="Edm.String" Nullable="false"
            MaxLength="30" FixedLength="false" Unicode="true"/>
      <Property Name="resume" Type="Edm.Stream" Nullable="true"
            MaxLength="Max" FixedLength="false">
         <ValueAnnotation Name="OData.ContentType"
                          value="application/json"/>
      </Property>
   </EntityType>
</Schema>
```

Query Example:

To retrieve only those employees that have “Pizzo” as reference.lastname in their resume, one might submit:

```
http://www.example.com/mycompany/Employees
?$filter=resume/JSON.find_string
                 (‘(reference/lastname=″Pizzo″)’)
```

In this example, we use parentheses () instead of curly braces {}

Query with Sort Example:

To return resumes, ordering the result based on the state in which they live, where that state is located in their resume, one might submit:

```
http://www.example.com/mycompany/Employees/resume
?$orderby=resume/JSON.find_string(‘()’,’(state:1)’)
```

In this case the results are returned in ascending order of state.

Query with Mixed Data Returned Example:

To return the employee identifier and phone number of every employee, where the phone number has been taken from their resume, one might submit:

```
http://www.example.com/mycompany/Employees
?$select=empid,resume/JSON.find_string(‘()’,’(phone:1)’)
```

This query would require an extension to OData, allowing an expression to appear in a $select query option.

Complete Update Example:

To replace an old resume with a new resume for a specific employee, one might submit:

```
PUT /resume166549.json HTTP/1.1
Host: host
Content-Type: application/json
DataServiceVersion: 1.0
MaxDataServiceVersion: 3.0
If-Match: ...Etag...
Content-Length: ####

resume : {
          ssn:1234,
          lastname:"Handl",
          address:{zipcode:"10022", street:"ABC st"
          experience:"excellent",
          ……
         }
```

Summary Results Example:

To return a summarized resume for each employee, that includes lastname and experience only, one might submit:

```
http://www.example.com/mycompany/Employees
 /resume/JSON.find_string
('()','(resume/lastname:1)','(resume/experience:1)')
```

Design Principles

OData is an application-level protocol for interacting with data via RESTful web services. An OData Service's contract is defined by simple, well-defined conventions and semantics applied to the data model exposed by the service, providing a high level of semantic interoperability between loosely coupled clients and services.

The design principles of OData are to:

- Make it easy to implement and consume a basic OData service over a variety of data sources. Rather than try and expose the full functionality of all stores, define common features for core data retrieval and update scenarios and incremental, optional features for more advanced scenarios.
- Leverage Data Models to guide clients through common interaction patterns rather than force clients to write complex queries against raw data
- Define consistency across the protocol and a single way to express each piece of functionality

The design principles of OData extensions are to:

- Ensure extensions do not violate the core semantics of OData

- Avoid defining different representations for common concepts across extensions
- Ensure independent extensions compose well
- Ensure clients can ignore extended functionality and still query and consume data correctly

Technical Directions

The following are some technical directions for the JSON extension to OData:

- An OData vocabulary for JSON shall be defined.
- An annotation from a common vocabulary defining the JSON content type should be applied to a Stream property that represents JSON documents.
- The JSON vocabulary will define functions that can be applied to JSON properties.
- These functions will be based on common functions found in native JSON databases

Open questions, issues and work items

- The JSON annotation may contain additional properties describing the JSON document, e.g., one or more JSON schemas [13].
- Support may be provided for updating only a portion of a JSON property.
- OData could be extended to allow expressions in the $select query option, allowing derived values to be returned along with the properties of an entity.
- OData could be extended with an operator that returns the content of a Stream as either a String or Binary value.
- The OData.ContentType value annotation could be defined to allow multiple content types as its value.
- An alternative approach for the use of JSON in OData is to map JSON to dynamic properties of open data types. The rationale for choosing a document oriented approach is to treat the JSON as a single unit.
- This paper describes functions that operate on JSON encoded documents. These functions are applicable to other encodings such as ATOM, and the technical committee could consider a set of common functions across different encodings.

References

[1] JSON http://en.wikipedia.org/wiki/JSON
[2] ECMAScript http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
[3] IETF RFC 4627 http://www.ietf.org/rfc/rfc4627.txt
[4] PostGres http://www.postgresql.org/docs/devel/static/datatype-json.html
[5] Apache CouchDB http://en.wikipedia.org/wiki/CouchDB
[6] MongoDB http://en.wikipedia.org/wiki/MongoDB
[7] Document databases http://en.wikipedia.org/wiki/Document-oriented_database

[8] NoSQL databases http://en.wikipedia.org/wiki/NoSQL
[9] SQL/XML http://en.wikipedia.org/wiki/SQL/XML
[10] XQilla: XQuery extensions for JSON
http://xqilla.sourceforge.net/ExtensionFunctions
[11] JSONiq : XQuery extension for JSON http://www.w3.org/2011/10/integration-workshop/p/Documentation-0.1-JSONiq-Article-en-US.pdf
[12] Marklogic : http://en.wikipedia.org/wiki/MarkLogic
[13] JSON Schema: http://tools.ietf.org/html/draft-zyp-json-schema-03