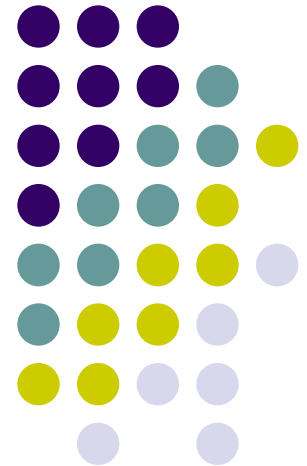


Advanced Aspects and New Trends in XML (and Related) Technologies

RNDr. Irena Holubová, Ph.D.

holubova@ksi.mff.cuni.cz

Lecture 2. JSON



JSON (JavaScript Object Notation)



- Text-based easy-to-read-and-write open standard for data interchange
 - Serializing and transmitting structured data
 - Considered as an **alternative to XML**
- Filename: `*.json`
- Internet media type (MIME type): `application/json`
- Derived from JavaScript scripting language
- Language independent
 - But uses conventions of the C-family of languages (C, C++, C#, Java, JavaScript, Perl, Python, ...)
- Originally specified by Douglas Crockford in 2001
 - RFC 4627
 - Requests for comments = "standard" publication of the Internet Engineering Task Force and the Internet Society



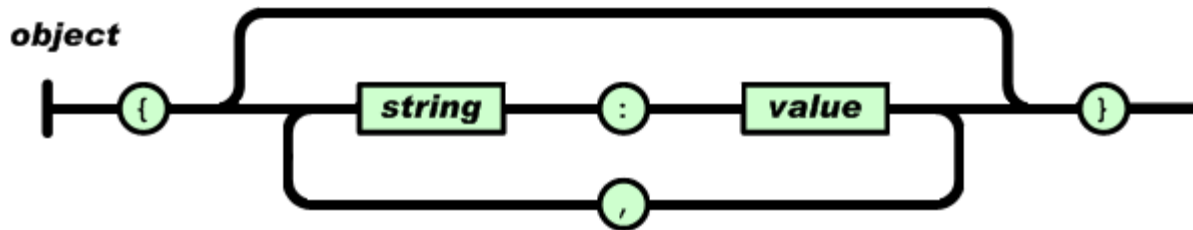
JSON – Basic Structures

- Built on two general structures:
 - Collection of name/value pairs
 - Realized as an object, record, struct, dictionary, hash table, keyed list, associative array, ...
 - Ordered list of values
 - Realized as an array, vector, list, sequence, ...
- Universal data structures
 - All modern programming languages support them

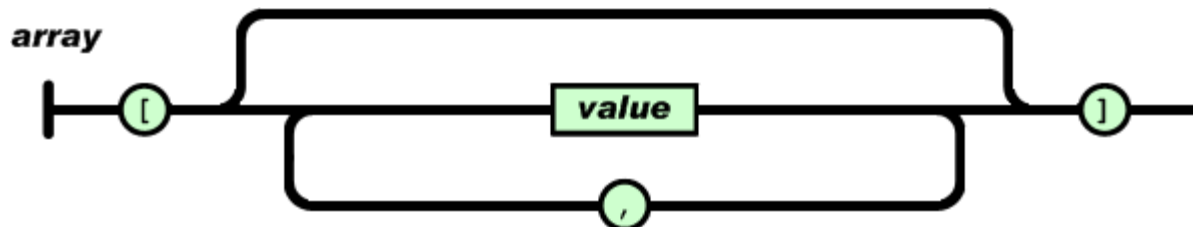


JSON – Basic Data Types

- **object** – an unordered set of name/value pairs
 - called properties (members) of an object
 - { comma-separated name : value pairs }



- **array** – an ordered collection of values
 - called items (elements) of an array
 - [comma-separated values]



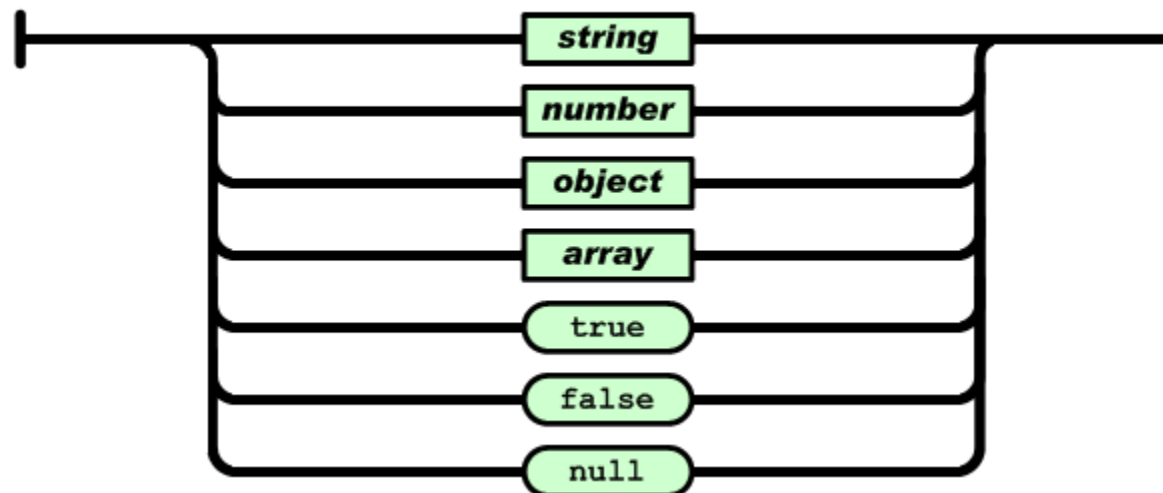


JSON – Basic Data Types

- **value** – string in double quotes / number / true or false (i.e., Boolean) / null / object / array
- Can be nested

XML subelements

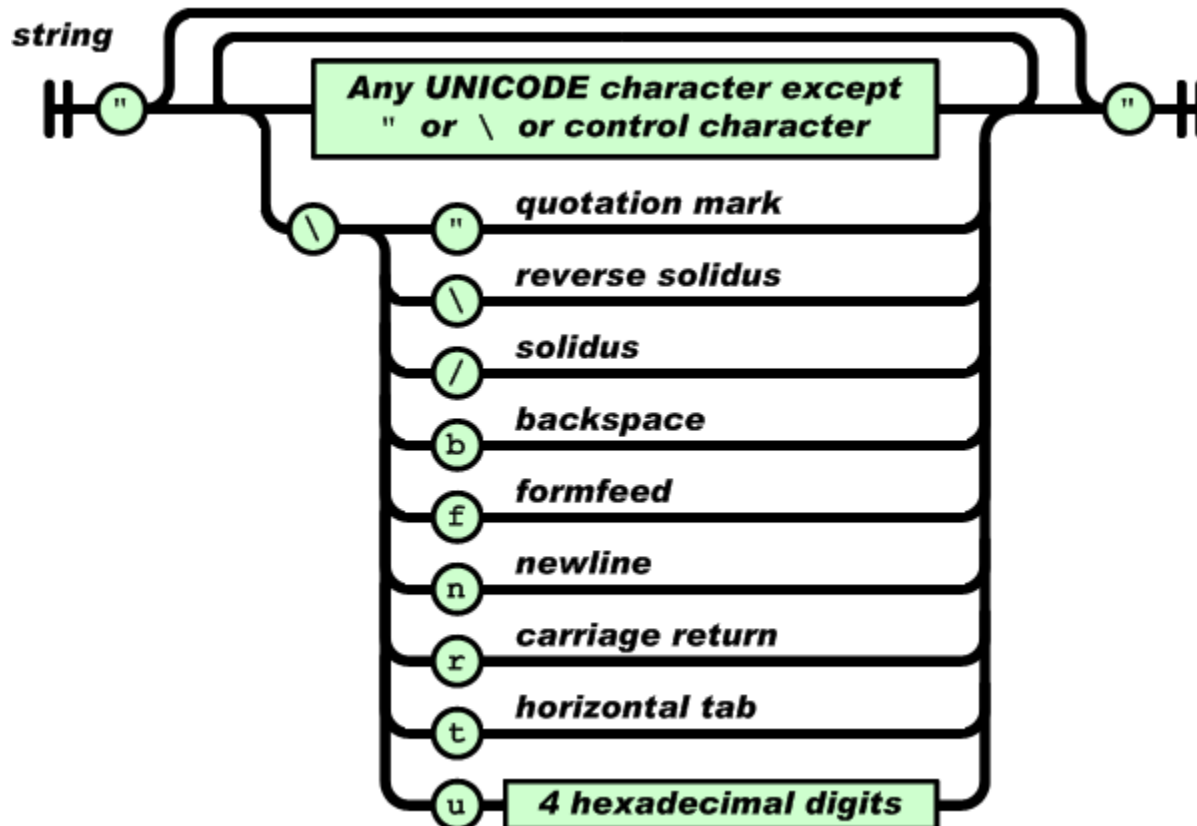
value





JSON – Basic Data Types

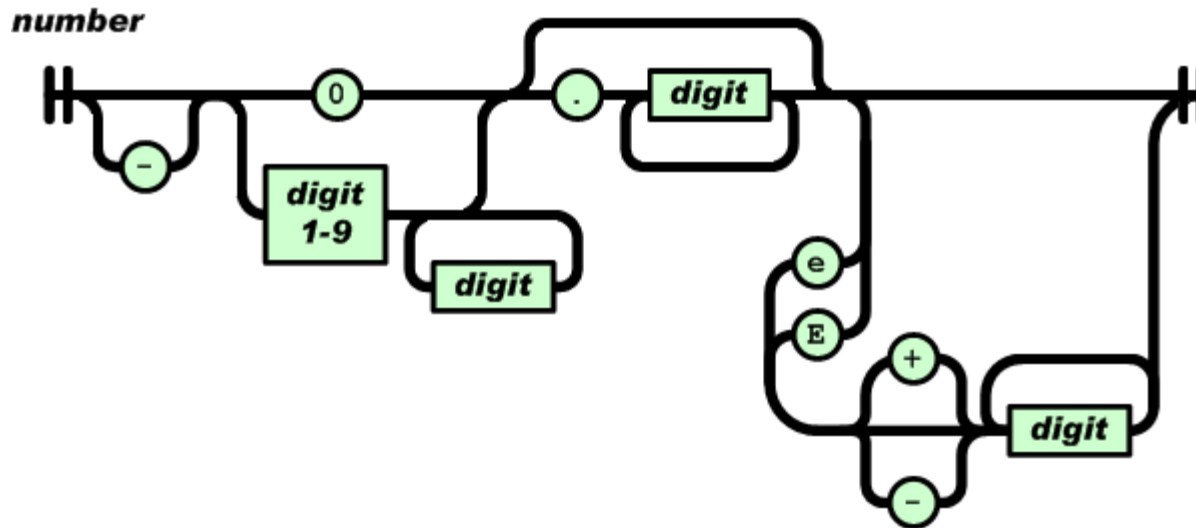
- **string** – sequence of zero or more Unicode characters, wrapped in double quotes
 - Backslash escaping





JSON – Basic Data Types

- **number** – like a C or Java number
 - Octal and hexadecimal formats are not used





JSON Example

```
{  
  "firstName": "John",  
  "lastName": "Smith",  
  "age": 25,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": 10021  
  },  
  "phoneNumbers": [  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "fax",  
      "number": "646 555-4567"  
    }  
  ]  
}
```

name/value pair

array

object



JSON Schema

- JSON-based format
- Defines the structure of JSON data for validation and/or documentation
- Based on concepts from DTD, XML Schema, RELAX NG, ...
- ...but:
 - Is JSON-based
 - The same serialization/deserialization tools can be used for schema and data
 - Is self-describing

XML: DTD vs. XML Schema

JSON Schema Example



```
{
  "id": 1,
  "name": "Foo",
  "price": 123,
  "tags": [ "Bar", "Eek" ],
  "stock": {
    "warehouse": 300,
    "retail": 20
  }
}
```

How to describe the allowed structure of these data?

```
{
  "title": "Product",
  "type": "object",
  "properties": {
    "id": {
      "type": "number",
      "description": "Product identifier"
    },
    "name": {
      "type": "string",
      "description": "Name of the product"
    },
    "price": {
      "type": "number",
      "minimum": 0
    },
    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "stock": {
      "type": "object",
      "properties": {
        "warehouse": {
          "type": "number"
        },
        "retail": {
          "type": "number"
        }
      }
    }
  }
  "required": ["id", "name", "price"]
}
```



JSON Schema – Principles

- JSON schema is a JSON document
 - Must be a single JSON object
- Properties of the object = **keywords**
 - Defined in the specification
- JSON schema may contain other properties which are not keywords
- Empty schema = schema (object) with no properties

```
{  
  "title": "root"  
}
```

root schema with no subschemas

```
{  
  "title": "root",  
  "otherSchema": {  
    "title": "nested",  
    "anotherSchema": {  
      "title": "alsoNested"  
    }  
  }  
}
```

subschemas

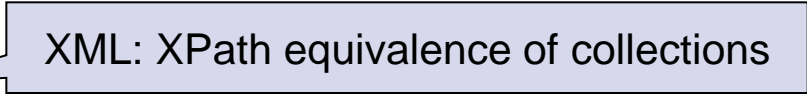
JSON Schema Primitive Types



- Correspond to basic data types
 - boolean – JSON Boolean
 - integer – JSON number without a fraction or exponent part
 - number – any JSON number
 - Includes integer
 - null – JSON null value
 - string – JSON string
 - array – JSON array
 - object – JSON object



JSON Value Equality

- Two JSON values are said to be **equal** if:
 - both are **nulls** or
 - both are **booleans** / **strings** / **numbers** and have the same value or
 - both are **arrays** and 
 - have the same number of items and
 - items at the same index are equal according to this definition or
 - both are **objects** and
 - have the same set of property names and
 - values for a same property name are equal according to this definition.



Metadata Keywords

- **title, description**
 - arbitrary strings
 - can be used anywhere
 - title is expected to be short, description may be longer
- **default**
 - a default JSON value associated with a particular (sub)schema

Keywords for Any Instance Type



XML: simple data types

- **enum**
 - an array of allowed values
- **type**
 - an array of allowed types
- **allOf**
 - an array of JSON schemas
 - against all of them an instance should be valid
- **anyOf**
 - an array of JSON schemas
 - against at least one of them an instance should be valid
- **not**
 - negation
- **definitions**
 - location for schema authors to inline other JSON schemas

Keywords for Numeric Instances



- `multipleOf`
 - an instance is a multiple of the value
- `maximum / minimum`
 - an instance is lower / greater than or equal to the value
- `exclusiveMaximum / exclusiveMinimum = true`
 - an instance is lower / greater than the value of "maximum" / "minimum"



Keywords for Strings

- `minLength` / `maxLength`
 - the length of an instance is greater / lower than or equal to the value
- `pattern`
 - an instance is considered valid if the regular expression matches the instance successfully
 - ECMA 262 regular expression dialect



Keywords for Arrays

- **items**
 - an object = an array of objects of this type
 - an array of objects = an array of objects of these types and positions
- **maxItems / minItems**
 - the size of an item is less / greater than or equal to the value
- **uniqueItems = true**
 - all items of an instance are unique
- **additionalItems**
 - only if keyword "items" represents an array of objects
 - otherwise ignored
 - Boolean = other types of objects are (not) allowed
 - an object = this type of object is also allowed

Validation Keywords for Objects



- **properties**
 - an object
 - each value of this object must be an object representing an (optional) property
- **maxProperties / minProperties**
 - the number of properties is less / greater than or equal to the value
- **required**
 - an array of required properties
- **additionalProperties**
 - Boolean = other types of properties are (not) allowed
 - an object = this type of property is also allowed
- **patternProperties**
 - an object
 - each property name of this object should be a valid regular expression describing a set of properties

similar to arrays



An Example Step-by-Step

```
{  
  "id": 1,  
  "name": "A green door",  
  "price": 12.50,  
  "tags": ["home", "green"]  
}
```

- What is id?
- Is name required?
- Can price be 0?
- Are all tags strings?

An Example Step-by-Step

To Start With...



this schema is written according
to the draft v4 specification

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "Product",  
  "description": "A product from Acme's catalog",  
  "type": "object"  
}
```

An Example Step-by-Step

What is id?



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    }
  },
  "required": ["id"]
}
```

- ID is a required property of type integer

An Example Step-by-Step

Is name required?



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "name": {
      "description": "Name of the product",
      "type": "string"
    }
  },
  "required": ["id", "name"]
}
```

- Name is a required string

An Example Step-by-Step

Can price be 0?



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "name": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "type": "number",
      "minimum": 0,
      "exclusiveMinimum": true
    }
  },
  "required": ["id", "name", "price"]
}
```

- Price is a required number greater than zero

An Example Step-by-Step

Are all tags strings?



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "name": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "type": "number",
      "minimum": 0,
      "exclusiveMinimum": true
    },
    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
    }
  },
  "required": ["id", "name", "price"]
}
```

- Tags form an array of at least one string with unique values



JSON Software

- JSON parsers
- JSON Schema validators
 - JavaScript, Java, Python, Ruby, PHP, .NET, C, Haskell, ...
- JSON Schema generators
 - <http://www.jsonschema.net/>
- XML-to-JSON transformers
- ...



JSON Hyper-Schema

- How JSON Schema can be used to define hyperlinks
- Defines keywords for the JSON Schema that allow JSON data to:
 - be understood as hyper-text
 - be interpreted as rich multimedia documents

```
{
  "title": "Written Article",
  "type": "object",
  "properties": {
    "id": {
      "title": "Article Identifier",
      "type": "number"
    },
    "title": {
      "title": "Article Title",
      "type": "string"
    },
    "authorId": {
      "type": "integer"
    },
    "imgData": {
      "title": "Article Illustration (small)",
      "type": "string",
      "media": {
        "binaryEncoding": "base64",
        "type": "image/png"
      }
    }
  },
  "required" : ["id", "title", "authorId"],
  "links": [
    {
      "rel": "full",
      "href": "{id}"
    },
    {
      "rel": "author",
      "href": "/user?id={authorId}"
    }
  ]
}
```

```
{
  "id": 15,
  "title": "Example data",
  "authorId": 105,
  "imgData": "iVBORw...kJggg=="
}
```

name of relation to the target

target URI

JsonML (JSON Markup Language)



- An application of JSON
- Lightweight markup language
 - Markup language with simple syntax
- Provides a compact format for transporting / processing XML-based markup as JSON
 - “Lossless” conversion back to original form
- Allows manipulation of XML data without the overhead of an XML parser
 - Using tools for JSON (typically in JavaScript)
 - Exploited mainly in AJAX
 - see later
- We can find:
 - An XSLT script for XML-to-JsonML transformation
 - A JavaScript for JsonML-to-DOM transformation

JsonML



- JSON vs. XML:
 - JSON arrays ↔ XML elements
 - JSON objects ↔ attributes
 - JSON strings ↔ text nodes
- Notes:
 - Comments are not supported
 - Namespaces are “supported” only in some implementations
 - Namespace declaration considered as attributes
 - Qualified names considered as names
- Result:
 - A more compact representation

```

element
    = '[' tag-name ',' attributes ',' element-list ']'
    | '[' tag-name ',' attributes ']'
    | '[' tag-name ',' element-list ']'
    | '[' tag-name ']'
    | json-string
    ;

tag-name
    = json-string
    ;

attributes
    = '{' attribute-list '}'
    | '{' '}'
    ;

attribute-list
    = attribute ',' attribute-list
    | attribute
    ;

attribute
    = attribute-name ':' attribute-value
    ;

attribute-name
    = json-string
    ;

attribute-value
    = json-string
    ;

element-list
    = element ',' element-list
    | element
    ;

```

JsonML grammar

JsonML Example

XHTML Unordered List



```
[  
  "ul",  
  { style : "list-style-type:none" },  
  [ "li", "First item" ],  
  [ "li", "Second item" ],  
  [ "li", "Third item" ],  
];
```

JsonML

Bigger Example

```
<table class="maintable">
  <tr class="odd">
    <th>Situation</th>
    <th>Result</th>
  </tr>
  <tr class="even">
    <td><a href="driving.html"
      title="Driving"
    >Driving</a></td>
    <td>Busy</td>
  </tr>
  ...
</table>
```

```
["table",
 {"class": "maintable"},
 "\n",
 ["tr",
 {"class": "odd"},
 "\n",
 ["th",
 "Situation"],
 "\n",
 ["th",
 "Result"],
 "\n"],
 "\n",
 ["tr",
 {"class": "even"},
 "\n",
 ["td",
 ["a",
 {"href": "driving.html",
 "title": "Driving"},
 "Driving"]],
 "\n",
 ["td",
 "Busy"],
 "\n"],
 ]
```

AJAX (Asynchronous JavaScript and XML)



- Idea: Web applications can send / retrieve data to / from a server asynchronously
 - At the background
 - Without interfering with the display and behavior of the existing page
- Typically using `XMLHttpRequest` object
 - Asynchronous request for data from web server
 - Must be the same that served the original web page
 - Response: used for altering the current web page without re-load
- We do not need to use XML format
 - AJAJ – Asynchronous JavaScript and JSON
- Example: Google Mail, Google Maps, ...



XMLHttpRequest object

- Standard API for transferring data between a client and a server
- Supports any text format
 - The name is just for compatibility
 - Typically: **XML**, HTML, **JSON**, plain text
- Both HTTP and HTTPS requests
 - Some applications support also other protocols

XMLHttpRequest Examples



```
function log(message) {  
  var client = new XMLHttpRequest();  
  client.open("POST", "/log");  
  client.setRequestHeader("Content-Type", "text/plain;charset=UTF-8");  
  client.send(message);  
}
```

sets request method + URL (+ other parameters)

sends the request

- to log a message to the server

```
function fetchStatus(address) {  
  var client = new XMLHttpRequest();  
  client.onreadystatechange = function() {  
    // in case of network errors this might not give reliable results  
    if(this.readyState == this.DONE)  
      returnStatus(this.status);  
  }  
  client.open("HEAD", address);  
  client.send();  
}
```

setting a handler

- to check the status of a document on the server



BSON (Binary JSON)

- Binary-encoded serialization of JSON documents
 - Allows embedding of documents, arrays, JSON simple data types + other types (e.g., date)
- Primary data representation in MongoDB database
 - Document NoSQL database
 - Documents described in JSON
 - Data storage and network transfer format

BSON Example



```
{"hello": "world"}
```

```
→ "\x16\x00\x00\x00\x02hello\x00  
\x06\x00\x00\x00world\x00\x00"
```

```
{"BSON": ["awesome",  
5.05, 1986]}
```

```
→ "\x31\x00\x00\x00\x04BSON\x00\x26\x00  
\x00\x00\x020\x00\x08\x00\x00  
\x00awesome\x00\x011\x00\x33\x33\x33  
\x33\x33\x33  
\x14\x40\x102\x00\xc2\x07\x00\x00  
\x00\x00"
```



BSON Basic Types

- `byte` – 1 byte (8-bits)
- `int32` – 4 bytes (32-bit signed integer)
- `int64` – 8 bytes (64-bit signed integer)
- `double` – 8 bytes (64-bit IEEE 754 floating point)



BSON Grammar

`document ::= int32 e_list "\x00"`

- BSON document
- `int32` = total number of bytes

`e_list ::= element e_list | ""`

- Sequence of elements



BSON Grammar

```
element ::= "\x01" e_name double
         | "\x02" e_name string
         | "\x03" e_name document
         | "\x04" e_name document
         | "\x05" e_name binary
         | ...
```

- Floating point
- UTF-8 string
- Embedded document
- Array
- Binary data
- ...

```
e_name ::= cstring
cstring ::= (byte*) "\x00"
```

- **Key name**

```
string ::= int32 (byte*) "\x00"
```

- **String = int32 bytes**

And so on...

JSONiq – the JSON Query Language

["JSONiq"]



- Query and functional programming language
- For declarative querying and transforming of hierarchical data formats...
 - JSON, XML
- ...as well as unstructured textual data
- Based on XQuery
 - Core expressions, operations on atomic types, ...
- Two syntaxes:
 - (pure/core) JSONiq syntax
 - Extended with XML support
 - XQuery syntax extended with JSON support
- Superset of JSON
 - Each JSON document is a JSONiq program

XML: similar to XQuery



JSONiq – Basics

- Aim: to extract and transform data from JSON documents
 - or any data that can be viewed as JSON data
- Major expression: SQL-like FLWOR expression from XQuery
 - FOR, LET, WHERE, ORDER BY, RETURN
 - Plus other operations
 - Grouping, windowing, ...
- Navigational syntax to extract data
- Constructs can be nested



JSONiq Syntaxes

- JSONiq syntax
 - Not concerned with XML features
 - Mixed content, ordered children, difference between attributes and elements, namespaces and QNames, ...
- XQuery syntax
 - Allows processing XML and JSON natively and with a single language
 - More complex
 - Has to follow rules of both languages (and especially more complex XQuery)

JSONiq Data Model (JDM)



- Extension of XQuery and XPath Data Model
- Tree-structured model
 - JSON objects
 - JSON arrays
 - All kinds of XML nodes
 - Atomic values
 - As defined in XML Schema specification
 - Include JSON simple types



Example 1. Join

`collection("users")`

```
{
  "name" : "Sarah",
  "age" : 13,
  "gender" : "female",
  "friends" : [ "Jim",
    "Mary", "Jennifer" ]
},
{
  "name" : "Jim",
  "age" : 13,
  "gender" : "male",
  "friends" : [ "Sarah" ]
}
```

```
{
  for $sarah in collection("users"),
    $friend in collection("users")
  where
    $sarah.name eq "Sarah"
    and
    (some $name in $sarah.friends[]
      satisfies $friend.name eq $name)
  return $friend
}
```

```
{ "name" : "Jim", "age" : 13, "gender"
: "male", "friends" : [ "Sarah" ] }
```

- Join on Sarah's friend list to return the Object representing each of her friends



Example 1. Join with Predicate

```
{  
  for $sarah in collection("users"),  
    $friend in collection("users")  
  where  
    $sarah.name eq "Sarah"  
    and  
    (some $name in $sarah.friends[]  
     satisfies $friend.name eq $name)  
  return $friend  
}
```

```
{  
  let $sarah := collection("users") [$$.name eq "Sarah"]  
  for $friend in $sarah.friends[]  
  return collection("users") [$$.name eq $friend]  
}
```




Example 2. Grouping

`collection("sales")`

```
{ "product" : "broiler", "store number" : 1, "quantity" : 20 },
{ "product" : "toaster", "store number" : 2, "quantity" : 100 },
{ "product" : "toaster", "store number" : 2, "quantity" : 50 },
{ "product" : "toaster", "store number" : 3, "quantity" : 50 },
{ "product" : "blender", "store number" : 3, "quantity" : 100 },
{ "product" : "blender", "store number" : 3, "quantity" : 150 },
{ "product" : "socks", "store number" : 1, "quantity" : 500 },
{ "product" : "socks", "store number" : 2, "quantity" : 10 },
{ "product" : "shirt", "store number" : 3, "quantity" : 10 },
...
```

```
{|
for $sales in collection("sales")
let $pname := $sales.product
group by $pname
return { $pname : sum($sales.quantity) }
|}
```

```
{
  "toaster" : 200,
  "blender" : 250,
  "shirt" : 10,
  "socks" : 510,
  "broiler" : 20
}
```

- Grouping sales by product, across stores

Example 3. More Complex Grouping – Data



collection("stores")

```
{ "store number" : 1, "state" : "CA" },  
{ "store number" : 2, "state" : "CA" },  
{ "store number" : 3, "state" : "MA" },  
{ "store number" : 4, "state" : "MA" }
```

collection("products")

```
{ "name" : "broiler", "category" : "kitchen", "price" : 100, "cost" : 70 },  
{ "name" : "toaster", "category" : "kitchen", "price" : 30, "cost" : 10 },  
{ "name" : "blender", "category" : "kitchen", "price" : 50, "cost" : 25 },  
{ "name" : "socks", "category" : "clothes", "price" : 5, "cost" : 2 },  
{ "name" : "shirt", "category" : "clothes", "price" : 10, "cost" : 3 }
```

Example 3. More Complex Grouping – Query



```
{|
for $store in collection("stores")
let $state := $store.state
group by $state
return {
  $state : {|
    for $product in collection("products")
    let $category := $product.category
    group by $category
    return {
      $category : {|
        for $sales in collection("sales")
        where (some $s in $store
              satisfies $sales."store number" eq $s."store number")
              and (some $p in $product
                   satisfies $sales.product eq $p.name)
        let $pname := $sales.product
        group by $pname
        return { $pname : sum( $sales.quantity ) }
      }
    }
  }
}
```

```
{
  "MA" :
    { "clothes" :
      { "shirt" : 10 },
      "kitchen" :
      { "toaster" : 50,
        "blender" : 250 } },
  "CA" :
    { "clothes" :
      { "socks" : 510 },
      "kitchen" :
      { "toaster" : 150,
        "broiler" : 20 } }
}
```

Example 4. JSON-to-JSON Transformation – Data



collection("satellites")

```
{
  "creator" : "Satellites plugin version 0.6.4",
  "satellites" : {
    "AAU CUBESAT" : {
      "tle1" : "1 27846U 03031G 10322.04074654 .00000056 00000-0 45693-4 0 8768",
      "visible" : false
    },
    "AJISAI (EGS)" : {
      "tle1" : "1 16908U 86061A 10321.84797408 -.00000083 00000-0 10000-3 0 3696",
      "visible" : true
    },
    "AKARI (ASTRO-F)" : {
      "tle1" : "1 28939U 06005A 10321.96319841 .00000176 00000-0 48808-4 0 4294",
      "visible" : true
    }
  }
}
```

- We want to return a summary

Example 4. JSON-to-JSON Transformation – Query



```
{|
let $sats := collection("satellites").satellites
return {
  "visible" : [
    for $sat in keys($sats)
    where $sats.$sat.visible
    return $sat
  ],
  "invisible" : [
    for $sat in keys($sats)
    where not $sats.$sat.visible
    return $sat
  ]
}
```

```
{
  "visible" : [ "AJISAI (EGS)", "AKARI (ASTRO-F)" ],
  "invisible" : [ "AAU CUBESAT" ]
}
```



Example 5. JSON Updates

`collection("users")`

```
{
  "name" : "Deadbeat Jim",
  "address" : "1 E 161st St, Bronx, NY 10451",
  "risk tolerance" : "high"
}
```

- We want to add `status` : `credit card declined` to the user's record

```
{|
let $dbj := collection("users")[ $$ .name = "Deadbeat Jim" ]
return insert { "status" : "credit card declined" } into $dbj
|}
```

```
{
  "name" : "Deadbeat Jim",
  "address" : "1 E 161st St, Bronx, NY 10451",
  "status" : "credit card declined",
  "risk tolerance" : "high"
}
```



JSONiq Future Work

- Aim: to finish standardization of:
 - JSONiq full-text support
 - Based on XQuery 1.0 and XPath 2.0 Full-Text
 - JSONiq update support
 - Based on XQuery Update Facility
 - JSONiq scripting support
 - XQuery Scripting 1.0
 - Extension of XQuery to serve as a scripting language



References

- JSON: <http://www.json.org/>
- RFC 4627: <http://tools.ietf.org/html/rfc4627>
- JSON Schema: <http://json-schema.org/>
- JSON Hyper-Schema: <http://json-schema.org/latest/json-schema-hypermedia.html>
- JsonML: <http://www.jsonml.org/>
- Jesse James Garrett – Ajax: A New Approach to Web Applications: <http://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php>
- XMLHttpRequest W3C Specification: <http://www.w3.org/TR/XMLHttpRequest/>
- BSON: <http://bsonspec.org/>
- JSONiq: <http://www.jsoniq.org/>
- JSONiq Use Cases: http://www.jsoniq.org/docs/JSONiq-usecases/pdf/JSONiq_-_the_SQL_of_NoSQL-1.0-JSONiq_Use_Cases-en-US.pdf