3 JSON – A Short Overview

Besides XML another data representation is gaining importance in designing data flow in multi-tier systems: JSON – the *JavaScript Object Notation*. We'll go briefly through three aspects:

- basic JSON syntax
- use of JSON in Javascript
- use of JSON in Java.

3.1 JSON Syntax and Javascript use

The examples marked "W3S" are taken directly from http://www.w3schools.com/json/json_intro.asp

W3S:

```
<!DOCTYPE html>
<html>
   <body>
      <h2>JSON Object Creation in JavaScript</h2>

      <p>
         Name: <span id="jname"></span><br />
         Age: <span id="jage"></span><br />
         Address: <span id="jstreet"></span><br />
         Phone: <span id="jphone"></span><br />
      </p>

      <script>
         var JSONObject = {
            "name": "John Johnson",
            "street": "Oslo West 555",
            "age": 33,
            "phone": "555 1234567"};
         document.getElementById("jname").innerHTML = JSONObject.name;
         document.getElementById("jage").innerHTML = JSONObject.age;
         document.getElementById("jstreet").innerHTML = JSONObject.street;
         document.getElementById("jphone").innerHTML = JSONObject.phone;
      </script>
   </body>
</html>
```

For simple data structures the JSON format shows similarities with XML; the differences:

- instead of elements limited by tags, JSON uses key/value associations
- the only types supported by JSON are strings, numbers, booleans, arrays, objects and null
- JSON also supports arrays as a means of grouping repeating elements (see below)

The reasons for adopting JSON over XML are based on the observation that for many applications

- schema-based validation and
- CSS/XSLT support

might not be necessary. Furthermore, the additional text needed for tags can lead to an unwanted ratio of tags to usable "payload" information, rendering XML as a somewhat "verbose" format. All this makes JSON a valid contender for particular applications.

JSON structures can be nested; substructures are accessed in standard component notation:

```
<!DOCTYPE html>
<html>
   <head>
      <title></title>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <script type="text/javascript"
            src="http://ajax.googleapis.com/ajax/libs/jquery/1.5/jquery.min.js">
      </script>
      <script type="text/javascript">
         $(document).ready(function() {
            $('#room').html(room.description);
            $('#video').html("<iframe width=\"560\" height=\"315\"
              src=\"http://www.youtube.com/embed/" + room.media.youtubeId +
              "\"frameborder=\"0\" allowfullscreen></iframe>");
         });
         var room = {"id": "scavenger://uwf.edu/location/r435",
            "is-a": {"id": "scavenger://uwf.edu/ontology/locations#room"},
            "located-in": {"id": "scavenger://uwf.edu/location/bldg4"},
            "contains": {"id": "scavenger://uwf.edu/actor/berndok"},
            "description": "Room 435 in Building 4",
            "media": {"youtubeId": "fvW6OE7WVB0"}
         };
      </script>
   </head>
   <body>
      <h1 id="room"></h1>
      <div id="video"></div>
   </body>
</html>
```

Multiple occurrences of a structure can be kept in an array – delimited by square brackets and accessed in standard JavaScript array notation (starting with 0):

W3S:

```
<!DOCTYPE html>
<html>
   <body>
      <h2>Create Object from JSON String</h2>
      <p>Original name: <span id="origname"></span></p>
      <p>New name: <span id="newname"></span></p>
      <script>
         var employees = [
            {"firstName": "John", "lastName": "Doe"},
            {"firstName": "Anna", "lastName": "Smith"},
            {"firstName": "Peter", "lastName": "Jones"}, ];

         document.getElementById("origname").innerHTML = employees[0].firstName +
                              " " + employees[0].lastName;
         employees[0].firstName = "Gilbert";
         document.getElementById("newname").innerHTML = employees[0].firstName +
                              " " + employees[0].lastName;
      </script>
   </body>
</html>
```

A Javascript string can be “evaluated” – the JSON text in this string is converted into a native JSON object:

W3S:

```
<!DOCTYPE html>
<html>
   <body>
      <h2>Create Object from JSON String</h2>
      <p>
         First Name: <span id="fname"></span><br>
         Last Name: <span id="lname"></span><br>
      </p>
      <script>
         var txt = '{"employees":[' +
               '{"firstName":"John","lastName":"Doe" },' +
               '{"firstName":"Anna","lastName":"Smith" },' +
               '{"firstName":"Peter","lastName":"Jones" }]}';

         var obj = eval("(" + txt + ")");
         document.getElementById("fname").innerHTML = obj.employees[1].firstName;
         document.getElementById("lname").innerHTML = obj.employees[1].lastName;
      </script>
   </body>
</html>
```

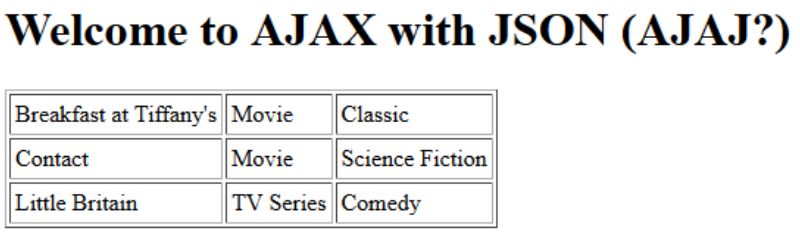
One of the interesting aspects of JSON is its usefulness as a representation for information transport, like XML – the fundamental idea of AJAX applies to JSON as well:

```
<!DOCTYPE html>
<html>
   <head>
      <title></title>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <script type="text/javascript"
              src="http://ajax.googleapis.com/ajax/libs/jquery/1.5/jquery.min.js">
      </script>
      <script type="text/javascript">
         $(document).ready(function() {
            $.ajax({
               type: "GET",
               url: "library.json",
               dataType: "json",
               success:
                     function(json) {
                        for (i = 0; i < json.library.length; i++) {
                           $('<tr></tr>').
                                 html("<td>" + json.library[i].dvd.title + "</td><td>" +
                                 json.library[i].dvd.format + "</td><td>" +
                                 json.library[i].dvd.genre + "</td>").
                            appendTo('#dvds');
                        }
                     }
            });
         });
      </script>
   </head>
   <body>
      <h1>Welcome to AJAX with JSON (AJAJ?)</h1>
      <table id="dvds" border="1" cellpadding="3">
      </table>
   </body>
</html>
```

With this library.json

```
{"library": [
      {"dvd": {"title": "Breakfast at Tiffany's", "format": "Movie", "genre": "Classic"}},
      {"dvd": {"title": "Contact", "format": "Movie", "genre": "Science Fiction"}},
      {"dvd": {"title": "Little Britain", "format": "TV Series", "genre": "Comedy"}}
   ]
}
```

the result looks like this:



3.2 JSON in Java

At the moment (Fall 2013) there are a couple of Java JSON libraries available, amongst others

- org.json maintained by json.org – compare http://json.org/java/
- javax.json – JSONP (Java API for JSON Processing) based on JSR 353 and presumably about to become part of Java EE7; http://jcp.org/en/jsr/detail?id=353
- com.fasterxml.jackson – the Jackson parser project, see https://github.com/FasterXML/jackson
- com.mongodb – the JSON processing part of MongoDB, see http://www.mongodb.org/

Since JSONP is probably going to be the upcoming standard API for JSON, we'll run though a couple of examples. It's important to understand that JSONP resembles JAXP in providing streaming and DOM-like parsing, a binding API like JAXB is not available at the moment[1]; the "Java EE 7 Tutorial" states at http://docs.oracle.com/javaee/7/tutorial/doc/jsonp005.htm:

> *The Java API for JSON Processing (JSR-353) does not explicitly support JSON binding in Java. A future JSR (JSON Binding) that is similar to JAXB for XML is under consideration for a future release of Java EE.*

JSONP was developed to be used within JAX-RS (Java API for RESTful Web Services) – see later in this course.

3.2.1 The JSONP Object Model

In general, the JSONP JsonObject resembles the JAXP Document – both are representatives of the internal tree structure of the underlying document. The JSONP handling of this object is a little less cumbersome, specifically because of the (smart!) convention to have the basic "build" method always return the receiving instance (or "implicit parameter" in Horstmann vernacular); technically it means that this method ends with a "return this;", neatly allowing to cascade these operations without having to repeat the instance every time (this is how Smalltalk and jQuery work; it makes programming so much easier!)

```
JsonObjectBuilder builder = Json.createObjectBuilder();
JsonObject o = builder
      .add("firstName", "Duke")
      .add("lastName", "Java")
      .add("age", 18)
      .add("streetAddress", "100 Internet Dr")
```

1 Jackson offers a binding version; since many ideas of Jackson went into the JSR 353 implementation, it can be assumed that there will be a binding standard a la JAXB sooner or later.

```
        .add("city", "JavaTown")
        .add("state", "JA")
        .add("postalCode", "12345")
        .build();
```

The resulting JSON structure looks like this (although technically the sequence of key/value pairs is undefined):

```
{"firstName":"Duke","lastName":"Java","age":18,
  "streetAddress":"100 Internet Dr","city":"JavaTown",
  "state":"JA","postalCode":"12345"}
```

Embedded arrays and objects are constructed by passing the appropriate builder instances:

```
    JsonObjectBuilder builder = Json.createObjectBuilder();
    JsonObject o = builder
        .add("firstName", "Duke")
        .add("lastName", "Java")
        .add("age", 18)
        .add("streetAddress", "100 Internet Dr")
        .add("city", "JavaTown")
        .add("state", "JA")
        .add("postalCode", "12345")
        .add("phoneNumbers", Json.createArrayBuilder()
          .add(Json.createObjectBuilder()
            .add("type", "mobile")
            .add("number", "111-111-1111"))
          .add(Json.createObjectBuilder()
            .add("type", "home")
            .add("number", "222-222-2222")))
        .build();
```

yielding

```
{"firstName":"Duke", "lastName":"Java", "age":18,
  "streetAddress":"100 Internet Dr","city":"JavaTown",
  "state":"JA", "postalCode":"12345",
  "phoneNumbers": [{"type":"mobile","number":"111-111-1111"},
                   {"type":"home","number":"222-222-2222"}]}
```

The object model is captured in the javax.json.JsonObject interface (http://docs.oracle.com/javaee/7/api/javax/json/JsonObject.html). It integrates the basic access methods, such as getXYZ for types XYZ and is a special case of javax.json.JsonValue (see below):

```
javax.json.JsonValue
    javax.json.JsonNumber
    javax.json.JsonString
    javax.json.JsonStructure
        javax.json.JsonObject
        javax.json.JsonArray
```

The mapping of JSON types into this interface hierarchy covers the types available in a JSON document, yet maybe not in the most intuitive manner: booleans are represented by constants TRUE and FALSE[2] and null is just NULL.

The structure of a JSON model looks (understandably!) pretty much like an XML DOM:

```
Object
|
+-- firstName:     STRING Duke
|
+-- lastName:      STRING Java
|
+-- age:           NUMBER 18
|
+-- streetAddress: STRING 100 Internet Dr
|
+-- city:          STRING JavaTown
|
+-- state:         STRING Ja
|
+-- streetAddress: STRING 100 Internet Dr
|
+-- postalCode:    STRING 12345
|
+-- phoneNumbers:  ARRAY [0] OBJECT
                             |
                             +-- type:   STRING mobile
                             |
                             +-- number: STRING 111-111-1111
                           [1] OBJECT
                             |
                             +-- type:   STRING home
                             |
                             +-- number: STRING 222-222-2222
```

A simple recursive traversal method for JSON model is given in the provided NetBeans project. Also simple examples for the use of the org.json and com.fasterxml.jackson APIs are given there.

2 So, there is no javax.json.JsonBoolean, the type names as well as TRUE, FALSE and NULL are kept in an Enum (JsonValue.ValueType, http://docs.oracle.com/javaee/7/api/javax/json/JsonValue.ValueType.html)