JSON and AJAX

Paul Fodor
CSE316: Fundamentals of Software Development
Stony Brook University

http://www.cs.stonybrook.edu/~cse316
Evolution of DBMS

- Semi-structured era (~2000+)
  - Schema Evolution OR Schema "later": data is self describing

<table>
<thead>
<tr>
<th>Person:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: Joe Jones</td>
</tr>
<tr>
<td>Wages: 14.75</td>
</tr>
<tr>
<td>Employer: My_accounting</td>
</tr>
<tr>
<td>Hobbies: skiing, bicycling</td>
</tr>
<tr>
<td>Works_for: ref (Fred Smith)</td>
</tr>
<tr>
<td>Favorite_joke: Why did the chicken cross the road? To get to the other side</td>
</tr>
<tr>
<td>Office_number: 247</td>
</tr>
<tr>
<td>Major_skill: accountant</td>
</tr>
<tr>
<td>End Person</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: Smith, Vanessa</td>
</tr>
<tr>
<td>Wages: 2000</td>
</tr>
<tr>
<td>Favorite_coffee: Arabian</td>
</tr>
<tr>
<td>Pastimes: sewing, swimming</td>
</tr>
<tr>
<td>Works_for: Between jobs</td>
</tr>
<tr>
<td>Favorite_restaurant: Panera</td>
</tr>
<tr>
<td>Number_of_children: 3</td>
</tr>
<tr>
<td>End Person:</td>
</tr>
</tbody>
</table>

**Semantic Heterogeneity:**
* Different sets of attributes
* Same attributes have different formats
* Different attributes have same meaning

- A Response to the growth of Web services (AJAX) and XML as a language (same for JSON as Javascript)
Evolution of DBMS

• Semi-structured era (~2000+)
  • Relational DBMS have heavy-weight mechanisms to change schema (ALTER)
  • XML and JSON as a data model:
    • records can be hierarchical
    • records can still reference to other records through paths (i.e., XPath)
    • schema can be defined "later" in DTDs and XMLSchema
Evolution of DBMS

- For machine consumption on the Web, data should have these characteristics:
  - Be *object-like*
  - Be *schemaless* (not guaranteed to conform exactly to any schema, but different objects have some commonality among themselves)
  - Be *self-describing* (some schema-like information, like attribute names, is part of data itself)
  - Data with these characteristics are referred to as *semistructured*. 
Non-self-describing Data

- Non-self-describing (relational, object-oriented):

  **Data part:**

  ```
  (#123, ["Students", 
          {["John", s111111111, [123,"Main St"]],
           ["Joe", s222222222, [321, "Pine St"]])
  ```

  **Schema part:**

  ```
  PersonList[ ListName: String,
              Contents: [ Name: String,
                        Id: String,
                        Address: [Number: Integer, Street: String] ] ]
  ```
Evolution of DBMS

- **Self-describing:**
  - Attribute names embedded in the data itself, *but are distinguished* from values
  - Doesn’t need schema to figure out what is what (but schema might be useful nonetheless)

```
(#12345,
   [ListName: “Students”,
    Contents: { [ Name: “John Doe”,
                 Id: “s111111111”,
                 Address: [Number: 123, Street: “Main St.”] ],
               [Name: “Joe Public”,
                Id: “s222222222”,
                Address: [Number: 321, Street: “Pine St.”] } }
  )
```
JSON

- Java Script Object Notation
- Lightweight data interchange
- Used with 'RESTful' APIs and AJAX (Asynchronous Javascript and XML)
JSON – Data Types

- **Number** – Integers and Floating point numbers do not have separate types
- **String** – A sequence of characters
- **Boolean** – true/false
- **Array** – An ordered list
- **Objects** – Sets of name/value pairs
- **Null** – an empty (non-existent) value
JSON - Syntax

• Data in Key/Value pairs: { 'key': 'value' }
  • Key must be quoted!
  • Value must be one of the described data types
• File extension should be .json
• MIME Types: Application/json
JSON - Syntax

- XMLHttpRequest.readyState


<table>
<thead>
<tr>
<th>Value</th>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNSENT</td>
<td>Client has been created. open() not called yet.</td>
</tr>
<tr>
<td>1</td>
<td>OPENED</td>
<td>open() has been called.</td>
</tr>
<tr>
<td>2</td>
<td>HEADERS_RECEIVED</td>
<td>send() has been called, and headers and status are available.</td>
</tr>
<tr>
<td>3</td>
<td>LOADING</td>
<td>Downloading; responseText holds partial data.</td>
</tr>
<tr>
<td>4</td>
<td>DONE</td>
<td>The operation is complete.</td>
</tr>
</tbody>
</table>

JSON - Example

people.json

```json
{
    "people" : [
    {
        "name":"Tony",
        "age":55
    },
    {
        "name":"Tina",
        "age":35
    },
    {
        "name":"Joe",
        "age":10
    }
    ]
}
```

json3.html

```html
<body>
<h1>People Array</h1>
<ul id='people'></ul>
<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        // Typical actions to be performed when the document is ready:
        console.log(xhttp.responseText);
        var response = JSON.parse(xhttp.responseText);
        var people = response.people;
        var output='';
        for (var i = 0; i < people.length; i++) {
            output += '<li>'+people[i].name+'</li>
        }
        document.getElementById('people').innerHTML = output;
    }
};
xhttp.open('GET', 'people.json', true);
xhttp.send();
</script>
</body>
```
JSON – Utility Functions

- `JSON.parse()` – Read a string as a JSON string, parse it, and generate a Javascript object with the contents of the string

- `JSON.stringify()` – Convert data or a Javascript Object into JSON notation
JSON.parse()

- JSON.parse() reads JSON strings and converts them to objects for use by Javascript
- Syntax:

  `JSON.parse(<string>, <reviver>);`

- `<string>` is the string to be parsed and converted to a Javascript object
- `<reviver>` is an optional parameter holding a function to convert or modify values
JSON – Example modify data w/parse()

people.json

```json
{
  "people": [
    {
      "name": "Tony",
      "age": 55
    },
    {
      "name": "Tina",
      "age": 35
    },
    {
      "name": "Joe",
      "age": 10
    }
  ]
}
```

json5.html

```html
<body>
<h1>People Array</h1>
<ul id='people'></ul>
<script>
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      // Typical action to be performed when the document is ready:
      var response = JSON.parse(xhttp.responseText, (key, value) =>
        key === 'age' ? value + 10 : value);
      var people = response.people;
      var output='';
      for (var i = 0; i < people.length; i++) {
        output += '<li>'+people[i].name+'...'+people[i].age+'</li>);
      }
      document.getElementById('people').innerHTML = output;
  }
</script>

xhttp.open('GET', 'people.json', true);
  xhttp.send();
</script>
</body>
```

People Array

- Tony...65
- Tina...45
- Joe...20
JSON.stringify()

- This convert any Javascript data or object into correct JSON syntax

- Syntax:

  JSON.stringify(<value>, <replacer>, <space>)

- The second two arguments are optional. The args are:
  - <value> - The data to be converted
  - <replacer> - This can be either:
    - A function that alters the behavior of stringify by selecting properties to include.
    - An array of strings that are used to filter/select which properties stringify() includes
  - <space> - This is either:
    - A number (up to 10) that indicate how many spaces to use between elements
    - A string (up to 10 characters long) used as the space separator
if (this.readyState == 4 && this.status == 200) {
    // Typical action to be performed when the document is ready:
    var response = JSON.parse(xhttp.responseText, (key, value) =>
        key === 'age' ? value + 10 : value);
    console.log(response.people);
    var newpeople = JSON.stringify(response.people, replacer);
    var finalpeople = JSON.parse(newpeople);
    var output = ''; 
    for (var i = 0; i < finalpeople.length; i++) {
        output += '<li>' + finalpeople[i].name + '...' + finalpeople[i].age + '</li>;
    }
    document.getElementById('people').innerHTML = output;
}