Web Programming Step by Step, 2nd Edition

Chapter 12: Ajax, XML, and JSON

Except where otherwise noted, the contents of this document are Copyright 2012 Marty Stepp, Jessica Miller, and Victoria Kirst. All rights reserved. Any redistribution, reproduction, transmission, or storage of part or all of the contents in any form is prohibited without the author's expressed written permission.

12.1: Ajax Concepts

- 12.1: Ajax Concepts
- 12.2: Using XMLHttpRequest
- 12.3: XML
- 12.4: JSON
Synchronous web communication

- **synchronous**: user must wait while new pages load
  - the typical communication pattern used in web pages (click, wait, refresh)

Web applications and Ajax

- **web application**: a dynamic web site that mimics the feel of a desktop app
  - presents a continuous user experience rather than disjoint pages
  - examples: Gmail, Google Maps, Google Docs and Spreadsheets, Flickr, A9
- **Ajax**: Asynchronous JavaScript and XML
  - not a programming language; a particular way of using JavaScript
  - downloads data from a server in the background
  - allows dynamically updating a page without making the user wait
  - avoids the "click-wait-refresh" pattern
  - examples: UW's CSE 14x Diff Tool, Practice-It; Google Suggest
Asynchronous web communication

- **asynchronous**: user can keep interacting with page while data loads
  - communication pattern made possible by Ajax

**12.2: Using XMLHttpRequest**

- 12.1: Ajax Concepts
- **12.2: Using XMLHttpRequest**
- 12.3: XML
- 12.4: JSON
**XMLHttpRequest** *(and why we won't use it)*

- JavaScript includes an **XMLHttpRequest** object that can fetch files from a web server
  - supported in IE5+, Safari, Firefox, Opera, Chrome, etc. (with minor compatibilities)
- it can do this **asynchronously** (in the background, transparent to user)
- the contents of the fetched file can be put into current web page using the DOM

- sounds great!...
- ... but it is clunky to use, and has various browser incompatibilities
- Prototype provides a better wrapper for Ajax, so we will use that instead

---

**A typical Ajax request**

1. user clicks, invoking an event handler
2. handler's code creates an **XMLHttpRequest** object
3. **XMLHttpRequest** object requests page from server
4. server retrieves appropriate data, sends it back
5. **XMLHttpRequest** fires an event when data arrives
   - this is often called a **callback**
   - you can attach a handler function to this event
6. your callback event handler processes the data and displays it
Prototype's Ajax model

new Ajax.Request("url",
    {
        option : value,
        option : value,
        ...
        option : value
    }
);

- construct a Prototype Ajax.Request object to request a page from a server using Ajax
- constructor accepts 2 parameters:
  1. the URL to fetch, as a String,
  2. a set of options, as an array of key: value pairs in {} braces (an anonymous JS object)
- hides icky details from the raw XMLHttpRequest; works well in all browsers

Prototype Ajax options

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>how to fetch the request from the server (default &quot;post&quot;)</td>
</tr>
<tr>
<td>parameters</td>
<td>query parameters to pass to the server, if any (as a string or object)</td>
</tr>
<tr>
<td>asynchronous</td>
<td>should request be sent asynchronously in the background? (default true)</td>
</tr>
</tbody>
</table>

new Ajax.Request("http://www.example.com/foo/bar.txt",
    {
        method: "get",
        parameters: {name: "Ed Smith", age: 29}, // "name=Ed+Smith&age=29"
        ...
    }
);
Prototype Ajax event options

<table>
<thead>
<tr>
<th>event</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onSuccess</td>
<td>request completed successfully</td>
</tr>
<tr>
<td>onFailure</td>
<td>request was unsuccessful</td>
</tr>
<tr>
<td>onException</td>
<td>request has a syntax error, security error, etc.</td>
</tr>
</tbody>
</table>

others: onCreate, onComplete, on### (for HTTP error code ###)

```javascript
{parameters: {password: "abcdef"}, // "password=abcdef"
onSuccess: mySuccessFunction
});
```

Basic Prototype Ajax template

```javascript
new Ajax.Request("url",
{method: "get",
onSuccess: functionName
});...

function functionName(ajax) {
do something with ajax.responseText;
}
```

- attach a handler to the request's onSuccess event
- the handler takes an Ajax response object, which we'll name ajax, as a parameter
Ajax response object's properties

<table>
<thead>
<tr>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>the request's HTTP error code (200 = OK, etc.)</td>
</tr>
<tr>
<td>statusText</td>
<td>HTTP error code text</td>
</tr>
<tr>
<td>responseText</td>
<td>the entire text of the fetched file, as a <strong>String</strong></td>
</tr>
<tr>
<td>responseXML</td>
<td>the entire contents of the fetched file, as a DOM tree (seen later)</td>
</tr>
</tbody>
</table>

```javascript
function handleRequest(ajax) {
    alert(ajax.responseText);
}
```

- most commonly used property is **responseText**, to access the fetched text content

XMLHttpRequest security restrictions

- cannot be run from a web page stored on your hard drive
- can only be run on a web page stored on a web server
- can only fetch files from the same site that the page is on
  - **http://www.foo.com/a/b/c.html** can only connect to **www.foo.com**
Handling Ajax errors

```javascript
new Ajax.Request("url",
{
    method: "get",
    onSuccess: functionName,
    onFailure: ajaxFailure,
    onException: ajaxFailure
}
);
...

function ajaxFailure(ajax, exception) {
    alert("Error making Ajax request:" +
        "\n\nServer status: " + ajax.status + " " + ajax.statusText +
        "\n\nServer response text: " + ajax.responseText);
    if (exception) {
        throw exception;
    }
}
```

- for user's (and developer's) benefit, show an error message if a request fails

Debugging Ajax code

- **Net** tab shows each request, its parameters, response, any errors
- expand a request with + and look at **Response** tab to see Ajax result
Creating a POST request

```javascript
new Ajax.Request("url",
{
    method: "post",   // optional
    parameters: { name: value, name: value, ... },
    onSuccess: functionName,
    onFailure: functionName,
    onException: functionName
}
);
```

- **method** should be changed to "post" (or omitted; post is default)
- any query parameters should be passed as a **parameters** parameter
  - written between {} braces as a set of **name**: **value** pairs (another anonymous object)
  - **get** request parameters can also be passed this way, if you like

Prototype's Ajax Updater

```javascript
new Ajax.Updater("id", "url",
{
    method: "get"
}
);
```

- **Ajax.Updater** fetches a file and injects its content into an element as **innerHTML**
- additional (1st) parameter specifies the **id** of element to inject into
- **onSuccess** handler not needed (but **onFailure, onException** handlers may still be useful)
PeriodicalUpdater

new Ajax.PeriodicalUpdater("id", "url",
    {
        frequency: seconds,
        name: value, ...
    }
);

- **Ajax.PeriodicalUpdater** repeatedly fetches a file at a given interval and injects its content into an element as `innerHTML`
- **onSuccess** handler not needed (but **onFailure, onFailure** handlers may still be useful)

Ajax.Responders

Ajax.Responders.register(
    {
        onEvent: functionName,
        onEvent: functionName,
        ...
    }
);

- sets up a default handler for a given kind of event for all Ajax requests
- useful for attaching a common failure/exception handler to all requests in one place
12.3: XML

- 12.1: Ajax Concepts
- 12.2: Using XMLHttpRequest
- **12.3: XML**
- 12.4: JSON

The bad way to store data

```
BEGIN
  TO: Tove
  FROM: Jani
  SUBJECT: Reminder
  MESSAGE (english):
    Hey there,
    Don't forget to call me this weekend!
END
```

- we could send a file like this from the server to browser with Ajax
- what's wrong with this approach?
What is XML?

- XML: a "skeleton" for creating markup languages
- you already know it!
  - syntax is identical to XHTML's:
    
    ```xml
    <element attribute="value">content</element>
    ```
  
  - languages written in XML specify:
    - names of tags in XHTML: h1, div, img, etc.
    - names of attributes in XHTML: id/class, src, href, etc.
    - rules about how they go together in XHTML: inline vs. block-level elements

- used to present complex data in human-readable form
  - "self-describing data"

Anatomy of an XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!‐‐ XML prolog -->
<note>  
  <!‐‐ root element -->
  <to>Tove</to>
  <from>Jani</from>
  <subject>Reminder</subject>
  <message language="english">
    Don't forget me this weekend!
  </message>
</note>
```

- begins with an `<?xml ... ?>` header tag ("prolog")
- has a single root element (in this case, `note`)
- tag, attribute, and comment syntax is just like XHTML
Uses of XML

- XML data comes from many sources on the web:
  - **web servers** store data as XML files
  - **databases** sometimes return query results as XML
  - **web services** use XML to communicate
- XML is the *de facto* universal format for exchange of data
- XML languages are used for **music**, **math**, **vector graphics**
- popular use: **RSS** for news feeds & podcasts

Pros and cons of XML

- **pro:**
  - easy to read (for humans and computers)
  - standard format makes automation easy
  - don't have to "reinvent the wheel" for storing new types of data
  - international, platform-independent, open/free standard
  - can represent almost any general kind of data (record, list, tree)
- **con:**
  - bulky syntax/structure makes files large; can decrease performance
    - example: **quadratic formula in MathML**
  - can be hard to "shoehorn" data into a good XML format
What tags are legal in XML?

- any tags you want!
- examples:
  - an email message might use tags called **to**, **from**, **subject**
  - a library might use tags called **book**, **title**, **author**
- when designing an XML file, you choose the tags and attributes that best represent the data
- rule of thumb: data = tag, metadata = attribute

Doctypes and Schemas

- "rule books" for individual flavors of XML
  - list which tags and attributes are valid in that language, and how they can be used together
- used to validate XML files to make sure they follow the rules of that "flavor"
  - the W3C HTML validator uses the XHTML doctype to validate your HTML
- for more info:
  - Document Type Definition (DTD) ("doctype")
  - W3C XML Schema
- optional — if you don't have one, there are no rules beyond having well-formed XML syntax
- (we won't cover these any further here)
XML and Ajax

- web browsers can display XML files, but often you instead want to fetch one and analyze its data
- the XML data is fetched, processed, and displayed using Ajax
  - (XML is the "X" in "Ajax")
- It would be very clunky to examine a complex XML structure as just a giant string!
- luckily, the browser can break apart (parse) XML data into a set of objects
  - there is an XML DOM, very similar to the (X)HTML DOM

XML DOM tree structure

```xml
<?xml version="1.0" encoding="UTF-8"?>
<categories>
  <category>children</category>
  <category>computers</category>
  ...
</categories>
```

- the XML tags have a tree structure
- DOM nodes have parents, children, and siblings
Recall: Javascript XML (XHTML) DOM

The DOM properties and methods* we already know can be used on XML nodes:

- properties:
  - `firstChild, lastChild, childNodes, nextSibling, previousSibling, parentNode`
  - `nodeName, nodeType,nodeValue, attributes`
- methods:
  - `appendChild, insertBefore, removeChild, replaceChild`
  - `getElementsByTagName, getAttribute, hasAttributes, hasChildNodes`
- caution: cannot use HTML-specific properties like `innerHTML` in the XML DOM!

* (though not Prototype's, such as `up, down, ancestors, childElements, or siblings`)

Navigating the node tree

- caution: can only use standard DOM methods/properties in XML DOM (NOT Prototype's)
- caution: can't use `ids` or `classes` to use to get specific nodes (no `$` or `$$`). Instead:

```javascript
// returns all child tags inside node that use the given element
var elms = node.getElementsByTagName("tagName");
```

- caution: can't use `innerHTML` to get the text inside a node. Instead:

```javascript
var text = node.firstChild.nodeValue;
```

- caution: can't use `.attributeName` to get an attribute's value from a node. Instead:

```javascript
var attrValue = node.getAttribute("attrName");
```
Using XML data in a web page

1. use Ajax to fetch data
2. use DOM methods to examine XML:
   - `XMLnode.getElementsByTagName("tag")`
3. extract the data we need from the XML:
   - `XMLelement.getAttribute("name"), XMLelement.firstChild.nodeValue, etc.`
4. create new HTML nodes and populate with extracted data:
   - `document.createElement("tag"), HTMLelement.innerHTML`
5. inject newly-created HTML nodes into page
   - `HTMLelement.appendChild(element)`

Fetching XML using AJAX (template)

```javascript
new Ajax.Request("url",
{
    method: "get",
    onSuccess: functionName
}
);
...

function functionName(ajax) {
    do something with ajax.responseTextXML;
}
```

- `ajax.responseText` contains the XML data in plain text
- `ajax.responseXML` is a pre-parsed XML DOM object
Analyzing a fetched XML file using DOM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<employees>
  <lawyer money="5"/>
  <janitor name="Sue">
    <vacuumcleaner/>
  </janitor>
  <janitor name="Bill">too poor</janitor>
</employees>
```

We can use DOM properties and methods on `ajax.responseXML`:

```javascript
// zeroth element of array of length 1
var employeesTag = ajax.responseXML.getElementsByTagName("employees")[0];

// how much money does the lawyer make?
var lawyerTag = employeesTag.getElementsByTagName("lawyer")[0];
var salary = lawyerTag.getAttribute("money"); // "5"

// array of 2 janitors
var janitorTags = employeesTag.getElementsByTagName("janitor");
var excuse = janitorTags[1].firstChild.nodeValue; // "too poor"
```

Analyzing a fetched XML file using DOM (2)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<employees>
  <lawyer money="5"/>
  <janitor name="Bill">
    <vacuumcleaner/>
  </janitor>
  <janitor name="Sue">too poor</janitor>
</employees>
```

What are the results of the following expressions?

```javascript
// zeroth element of array of length 1
var employeesTag = ajax.responseXML.getElementsByTagName("employees")[0];
```

- `employeesTag.firstChild`
- `ajax.responseXML.getElementsByTagName("lawyer")`
- `employeesTag.getElementsByTagName("janitor").length`
- `employeesTag.getElementsByTagName("janitor")[0].firstChild`
- `employeesTag.getElementsByTagName("janitor")[1].firstChild`
- `employeesTag.getElementsByTagName("janitor")[0].nextSibling`
Larger XML file example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year><price>30.00</price>
  </book>
  <book category="computers">
    <title lang="en">XQuery Kick Start</title>
    <author>James McGovern</author>
    <year>2003</year><price>49.99</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year><price>29.99</price>
  </book>
  <book category="computers">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year><price>39.95</price>
  </book>
</bookstore>
```

Navigating node tree example

```javascript
// make a paragraph for each book about computers
var books = ajax.responseXML.getElementsByTagName("book");
for (var i = 0; i < books.length; i++) {
  var category = books[i].getAttribute("category");
  if (category == "computers") {
    // extract data from XML
    var title = books[i].getElementsByTagName("title")[0].firstChild.nodeValue;
    var author = books[i].getElementsByTagName("author")[0].firstChild.nodeValue;

    // make an XHTML <p> tag containing data from XML
    var p = document.createElement("p");
    p.innerHTML = title + "", by " + author;
    document.body.appendChild(p);
  }
}
```
Exercise: Late day distribution

- Write a program that shows how many students turn homework in late for each assignment.
- Data service here: [http://webster.cs.washington.edu/cse190m/hw/hw.php](http://webster.cs.washington.edu/cse190m/hw/hw.php)
  - parameter: assignment=hw/N

A historical interlude: why XHTML?

- in XML, different "flavors" can be combined in single document
- theoretical benefit of including other XML data in XHTML
  - nobody does this
- most embedded data are in non-XML formats (e.g., Flash)
  - non-XML data must be embedded another way (we'll talk about this later on)
- requires browser/plugin support for other "flavor" of XML
  - development slow to nonexistent
  - most XML flavors are specialized uses
Exercise: Animal game

- Write a program that guesses which animal the user is thinking of. The program will arrive at a guess based on the user's responses to yes or no questions. The questions come from a web app named animalgame.php.

The Animal Game

Think of an animal, then let me guess it!

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can it fly?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Practice problem: Animal game (cont'd)

- The data comes in the following format:

```xml
<nod nodeid="id">
  <question>question</question>
  <yes nodeid="id"/>
  <no nodeid="id"/>
</node>

<nod nodeid="id">
  <answer>answer</answer>
</node>
```

- to get a node with a given id: animalgame.php?nodeid=id
- start by requesting the node with nodeid of 1 to get the first question
Attacking the problem

Questions we should ask ourselves:

- How do I retrieve data from the web app? (what URL, etc.)
- Once I retrieve a piece of data, what should I do with it?
- When the user clicks "Yes", what should I do?
- When the user clicks "No", what should I do?
- How do I know when the game is over? What should I do in this case?

Debugging responseXML in Firebug

- can examine the entire XML document, its node/tree structure
12.4: JSON

- 12.1: Ajax Concepts
- 12.2: Using XMLHttpRequest
- 12.3: XML
- 12.4: JSON

Pros and cons of XML

- pro:
  - standard open format; don't have to "reinvent the wheel" for storing new types of data
  - can represent almost any general kind of data (record, list, tree)
  - easy to read (for humans and computers)
  - lots of tools exist for working with XML in many languages
- con:
  - bulky syntax/structure makes files large; can decrease performance (example)
  - can be hard to "shoehorn" data into a good XML format
  - JavaScript code to navigate the XML DOM is bulky and generally not fun
JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON): Data format that represents data as a set of JavaScript objects

- invented by JS guru Douglas Crockford of Yahoo!
- natively supported by all modern browsers (and libraries to support it in old ones)
- not yet as popular as XML, but steadily rising due to its simplicity and ease of use

Recall: JavaScript object syntax

```
var person = {
    name: "Philip J. Fry",       // string
    age: 23,                     // number
    "weight": 172.5,             // number
    friends: ["Farnsworth", "Hermes", "Zoidberg"],  // array
    getBeloved: function() { return this.name + " loves Leela"; }
};
alert(person.age);          // 23
alert(person["weight"]);   // 172.5
alert(person.friends[2]);   // Zoidberg
alert(person.getBeloved()); // Philip J. Fry loves Leela
```

- in JavaScript, you can create a new object without creating a class
- the object can have methods (function properties) that refer to itself as this
- can refer to the fields with .fieldName or ["fieldName"] syntax
- field names can optionally be put in quotes (e.g. weight above)
An example of XML data

```xml
<?xml version="1.0" encoding="UTF-8"?>
<note private="true">
  <from>Alice Smith (alice@example.com)</from>
  <to>Robert Jones (roberto@example.com)</to>
  <to>Charles Dodd (cdodd@example.com)</to>
  <subject>Tomorrow's "Birthday Bash" event!</subject>
  <message language="english">
    Hey guys, don't forget to call me this weekend!
  </message>
</note>
```

The equivalent JSON data

```json
{
  "private": "true",
  "from": "Alice Smith (alice@example.com)",
  "to": [
    "Robert Jones (roberto@example.com)",
    "Charles Dodd (cdodd@example.com)"
  ],
  "subject": "Tomorrow's "Birthday Bash" event!",
  "message": {
    "language": "english",
    "text": "Hey guys, don't forget to call me this weekend!"
  }
}
```
Browser JSON methods

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON.parse(string)</td>
<td>converts the given string of JSON data into an equivalent JavaScript object and returns it</td>
</tr>
<tr>
<td>JSON.stringify(object)</td>
<td>converts the given object into a string of JSON data (the opposite of JSON.parse)</td>
</tr>
</tbody>
</table>

- you can use Ajax to fetch data that is in JSON format
- then call `JSON.parse` on it to convert it into an object
- then interact with that object as you would with any other JavaScript object

JSON expressions exercise

Given the JSON data at right, what expressions would produce:

- The window's title?
- The image's third coordinate?
- The number of messages?
- The y-offset of the last message?

```javascript
var data = JSON.parse(ajax.responseText);

var title = data.window.title;
var coord = data.image.coords[2];
var len = data.messages.length;
var y = data.messages[len - 1].offset[1];
```
**JSON example: Books**

Suppose we have a service `books_json.php` about library books.

- If no query parameters are passed, it outputs a list of book categories:

  ```json
  { "categories": ["computers", "cooking", "finance", ...] }
  ```

- Supply a `category` query parameter to see all books in one category:

  ```
  ```

  ```json
  {
  "books": [  
  {"category": "cooking", "year": 2009, "price": 22.00,  
  "title": "Breakfast for Dinner", "author": "Amanda Camp"},  
  {"category": "cooking", "year": 2010, "price": 75.00,  
  "title": "21 Burgers for the 21st Century", "author": "Stuart Reges"},  
  ...
  ]
  }
  ```

**JSON exercise**

Write a page that processes this JSON book data.

- Initially the page lets the user choose a category, created from the JSON data.
  - Children  ○  Computers  ○  Finance  [List Books]
- After choosing a category, the list of books in it appears:

  Books in category "Cooking":
  - Breakfast for Dinner, by Amanda Camp (2009)
  - The Four Food Groups of Chocolate, by Victoria Kirst (2005)
Working with JSON book data

```javascript
function showBooks(ajax) {
  // add all books from the JSON data to the page's bulleted list
  var data = JSON.parse(ajax.responseText);
  for (var i = 0; i < data.books.length; i++) {
    var li = document.createElement("li");
    li.innerHTML = data.books[i].title + ", by " +
      data.books[i].author + " (" + data.books[i].year + ")";
    $("books").appendChild(li);
  }
}
```

Bad style: the eval function

```javascript
// var data = JSON.parse(ajax.responseText);
var data = eval(ajax.responseText);  // don't do this!
...
```

- JavaScript includes an eval keyword that takes a string and runs it as code
- this is essentially the same as what JSON.parse does,
- but JSON.parse filters out potentially dangerous code; eval doesn't
- eval is evil and should not be used!