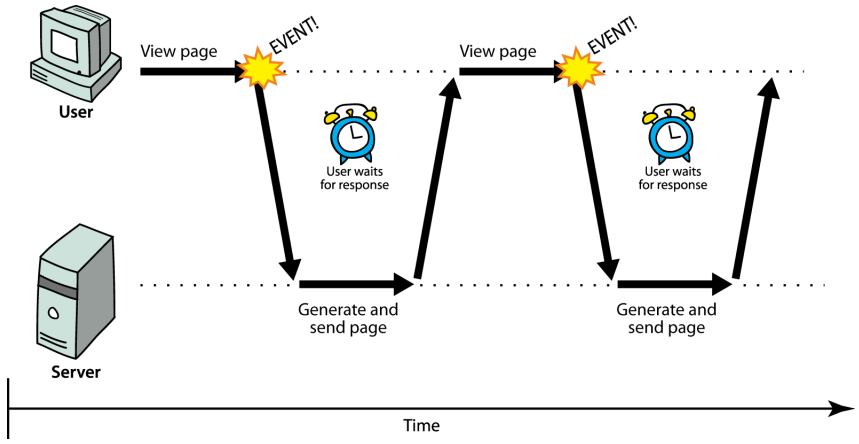
Web Programming Step by Step

Chapter 10
Ajax and XML for Accessing Data

10.1: Ajax Concepts

Synchronous web communication (10.1)



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

Web applications

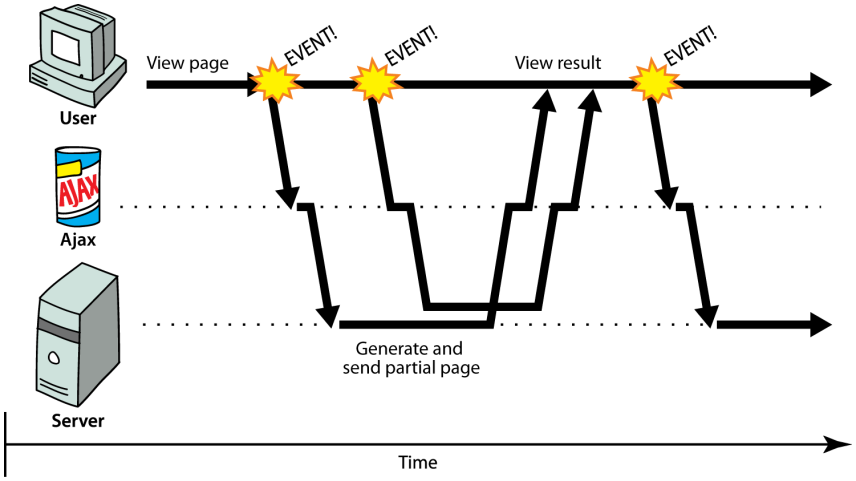
- **web application**: a web site that mimics the look, feel, and overall user experience of a desktop application
  - a web app presents a continuous user experience rather than disjoint pages
  - as much as possible, "feels" like a normal program to the user
- examples of web apps
  - Gmail, Google Maps, Google Docs and Spreadsheets, Flickr, A9
- many web apps use Ajax to battle these problems of web pages:
  - slowness / lack of UI responsiveness
  - lack of user-friendliness
  - jarring nature of "click-wait-refresh" pattern

What is Ajax?

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
- aids in the creation of rich, user-friendly web sites
    - examples: UW's CSE 14x Diff Tool, Practice-It; Google Suggest

Asynchronous web communication



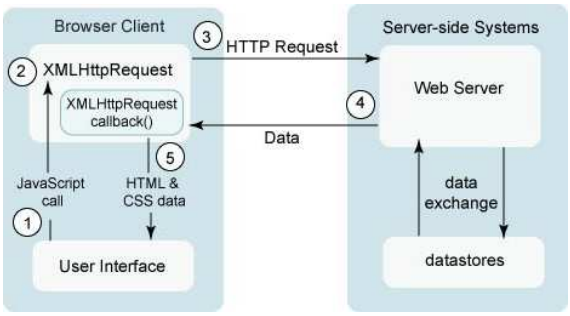
- **asynchronous**: user can keep interacting with page while data loads
    - communication pattern made possible by Ajax

Core Ajax concepts

- JavaScript's `XMLHttpRequest` object can fetch files from a web server
    - supported in IE5+, Safari, Firefox, Opera (with minor compatibilities)
- it can do this **asynchronously** (in the background, transparent to user)
- contents of fetched file can be put into current web page using DOM
- result: user's web page updates dynamically without a page reload

A typical Ajax request

1. user clicks, invoking event handler
2. that handler's JS code creates an `XMLHttpRequest` object
3. `XMLHttpRequest` object requests a document from a web server
4. server retrieves appropriate data, sends it back
5. `XMLHttpRequest` fires event to say that the data has arrived
    - this is often called a **callback**
    - you can attach a handler to be notified when the data has arrived
6. your callback event handler processes the data and displays it



10.2: Using XMLHttpRequest

- 10.1: Ajax Concepts
- **10.2: Using `XMLHttpRequest`**
- 10.3: XML

The `XMLHttpRequest` object

the core JavaScript object that makes Ajax possible

- methods: `abort`, `getAllResponseHeaders`, `getResponseHeader`, **`open`**, **`send`**, `setRequestHeader`
- properties: **`onreadystatechange`**, `readyState`, **`responseText`**, `responseXML`, `status`, `statusText`
- IE6 doesn't follow standards and uses its own `ActiveXObject` instead
- we'll learn to use Ajax in 4 steps:
  1. synchronous, text-only (SJAT?)
  2. asynchronous, text-only (AJAT?)
  3. asynchronous w/ Prototype (AJAP?)
  4. asynchronous w/ XML data (real Ajax)

1. Synchronous requests (10.2.1)

```
var ajax = new XMLHttpRequest();
ajax.open("GET", url, false);
ajax.send(null);

// at this point, the request will have returned with its data
do something with ajax.responseText;
```
JS

- create the request object, open a connection, send the request
- when `send` returns, the fetched text will be stored in request's `responseText` property

Why synchronous requests are bad

- your code waits for the request to completely finish before proceeding
- easier for you to program, but ...
- the user's *entire browser locks up* until the download is completed
- a terrible user experience (especially if the file is very large)

2. Asynchronous requests, basic idea (10.2.3)

```
var ajax = new XMLHttpRequest();
ajax.onreadystatechange = functionName;
ajax.open("get", url, true);
ajax.send(null);

// don't process ajax.responseText here, but in your function
...
```
JS

- attach an event handler to the request's `onreadystatechange` event
- pass `true` as third parameter to `open`
- handler will be called when request state changes, e.g. finishes
- function's code will be run when request is complete

The `readyState` property

- holds the status of the `XMLHttpRequest`
- possible values for the `readyState` property:

State	Description
0	not initialized
1	set up
2	sent
3	in progress
4	complete

- `readyState` changes → `onreadystatechange` handler runs
- usually we are only interested in `readyState` of 4 (complete)

Asynchronous `XMLHttpRequest` template

```
var ajax = new XMLHttpRequest();
ajax.onreadystatechange = function() {
  if (ajax.readyState == 4) {    // 4 means request is finished
    do something with ajax.responseText;
  }
};
ajax.open("get", url, true);
ajax.send(null);
```
JS

- most Ajax code uses an **anonymous function** as the event handler
  - useful to declare it as an inner anonymous function, because then it can access surrounding local variables (e.g. `ajax`)

Checking for request errors (10.2.2)

```
var ajax = new XMLHttpRequest();
ajax.onreadystatechange = function() {
  if (ajax.readyState == 4) {
    if (ajax.status == 200) {   // 200 means request succeeded
      do something with ajax.responseText;
    } else {
      code to handle the error;
    }
  }
};
ajax.open("get", url, true);
ajax.send(null);
```
JS

- web servers return status codes for requests (200 means Success)
- you may wish to display a message or take action on a failed request

Prototype's Ajax model (10.2.4)

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

- Prototype's `Ajax.Request` object 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
- hides some of the icky details (`onreadystatechange`, etc.)
- works in all browsers: IE, Firefox, etc.

Prototype Ajax methods and properties

- options that can be passed to the `Ajax.Request` constructor:
  - **`method`** : how to fetch the request from the server (default `"post"`)
  - **`parameters`** : query parameters to pass to the server, if any
  - `asynchronous` (default `true`), `contentType`, `encoding`, `requestHeaders`
- events in the `Ajax.Request` object that you can handle:
  - **`onSuccess`** : request completed successfully
  - **`onFailure`** : request was unsuccessful
  - `onCreate`, `onComplete`, `onException`, `on###` (handler for HTTP error code ###)

Prototype Ajax template

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

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

- most Ajax requests we'll do in this course are GET requests
- attach a handler to the request's `onSuccess` event
- the handler accepts the `XMLHttpRequest` object, `ajax`, as a parameter

Handling Ajax errors w/ Prototype

```
  new Ajax.Request(
    "url",
    {
      method: "get",
      onSuccess: functionName,
      onFailure: ajaxFailure
    }
  );
  ...
function ajaxFailure(ajax) {
  alert("Error making Ajax request:" +
        "\n\nServer status:\n" + ajax.status + " " + ajax.statusText +
        "\n\nServer response text:\n" + ajax.responseText);
}
```
JS

- for user's (and developer's) benefit, show a message if a request fails
- a good failure message shows the HTTP error code and status text

Creating a `POST` request

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

- `Ajax.Request` can also be used to post data to a web server
- `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 *name*: *value* pairs
  - `get` request parameters can also be passed this way, if you like

Prototype's Ajax Updater

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

- `Ajax.Updater` can be used if you want to fetch a file via Ajax and inject its text/HTML contents into an onscreen element
- additional (1st) parameter specifies the `id` of the element into which to inject the content

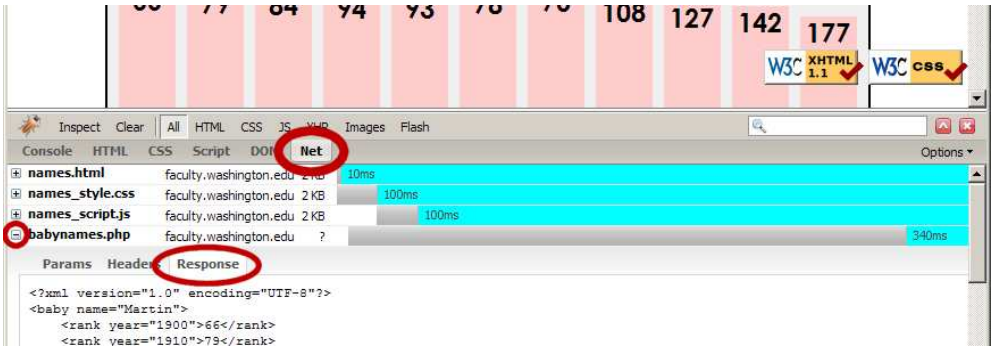
Ajax code bugs (10.2.5)

When writing Ajax programs, there are new kinds of bugs that are likely to appear.

- Nothing happens!
- The `responseText` or `responseXML` has no properties.
- The data isn't what I expect.

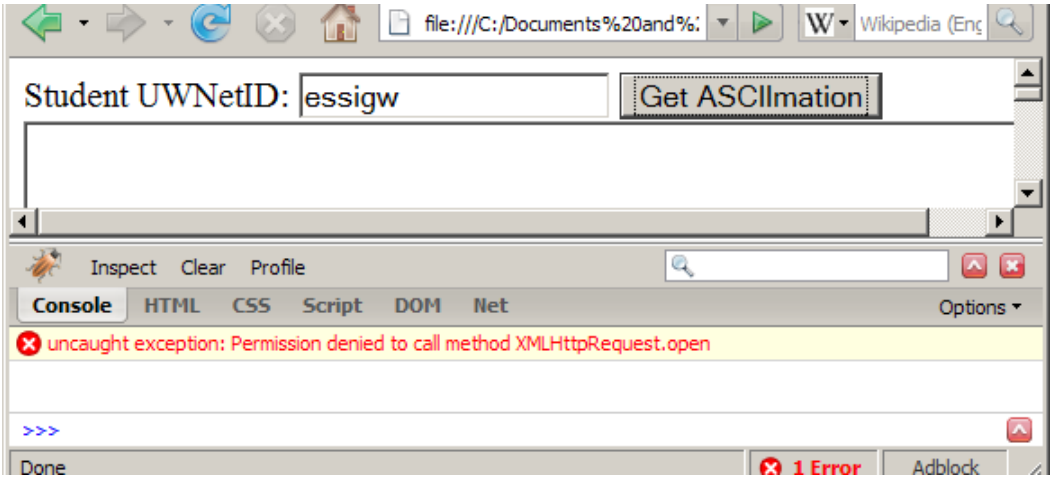
How do we find and fix such bugs?

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

`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
  - `www.foo.com/a/b/c.html` can only fetch from `www.foo.com`

10.3: XML

What is XML?

- **XML**: a specification for creating languages to store data; used to share data between systems
- a basic syntax of tags & attributes
- languages written in XML specify tag names, attribute names, and rules of use
- Example: XHTML is a "flavor" of XML
  - an adaptation of old HTML to fit XML's syntax requirements
  - XML specifies tag syntax: `<... ...="..."></...>`
  - HTML contributes tag names (e.g. `h1`, `img`) and attributes (`id/class` on all elements, `src/alt` on `img` tag)

An example XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <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, then a single **document tag** (in this case, `note`)
- tag, attribute, and comment syntax is identical to XHTML's

What tags are legal in XML?

- *any tag you want*; the person storing the data can make up their own tag structure
- example: a person storing data about email messages may want tags named `to`, `from`, `subject`
- example: a person storing data about books may want tags named `book`, `title`, `author`
- "Garden State" XML: if you're feeling unoriginal, make up some XML nobody's ever done before
  - `<bloop bleep="flibbetygibbet">quirkleblat</bloop>`

Schemas

- **schema**: an optional set of rules specifying which tags and attributes are valid, and how they can be used together
- used to *validate* XML files to make sure they follow the rules of that "flavor"
  - XHTML has a schema; W3C validator uses it to validate
  - doctype at top of XHTML file specifies schema
- two ways to define a schema:
  - Document Type Definition (DTD)
  - W3C XML Schema
- (we won't cover schemas any further here)

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 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 an intuitive XML format
    - won't need to know how for this class

Fetching XML using AJAX (template)

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

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

- `ajax.responseText` contains the XML data in plain text
- `ajax.responseXML` is a pre-parsed DOM object representing the XML file as a tree (more useful)

Using XML data in a web page

- custom flavor of XML needs to be converted to XHTML, then injected into page
- we will transform using Javascript XML DOM
- basic technique:
  1. fetch XML data using Ajax
  2. examine the `responseXML` object, using DOM methods and properties
  3. extract data from XML elements and wrap them in HTML elements
  4. inject HTML elements into web page
- other ways to transform XML (not covered): CSS, XSLT

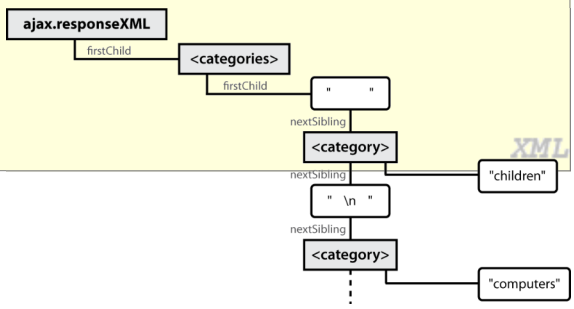
Recall: Javascript XML (XHTML) DOM

All of 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`**

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

Analyzing a fetched XML file using DOM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<foo bloop="bleep">
  <bar/>
  <baz><quux/></baz>
  <baz><xyzzy/></baz>
</foo>
```

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

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

// same
var bar = foo.getElementsByTagName("bar")[0];

// array of length 2
var all_bazzes = foo.getElementsByTagName("baz");

// string "bleep"
var bloop = foo.getAttribute("bloop");
```

Recall: Pitfalls of the DOM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<foo bloop="bleep">
  <bar/>
  <baz><quux/></baz>
  <baz><xyzzy/></baz>
</foo>
```

We are reminded of some pitfalls of the DOM:

```js
// works - XML prolog is removed from document tree
var foo = ajax.responseXML.firstChild;

// WRONG - just a text node with whitespace!
var bar = foo.firstChild;

// works
var first_baz = foo.getElementsByTagName("baz")[0];

// WRONG - just a text node with whitespace!
var second_baz = first_baz.nextSibling;

// works - why?
var xyzzy = second_baz.firstChild;
```

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 the node tree

- don't have `ids` or `classes` to use to get specific nodes
- `firstChild`/`nextSibling` properties are unreliable
- best way to walk the tree is using `getElementsByTagName`:

```
node.getElementsByTagName("tagName")
```
JS

  - get an array of all *node*'s children that are of the given tag (`"book"`, `"subject"`, etc.)
  - can be called on the overall XML document or on a specific node

- ```
  node.getAttribute("attributeName")
  ```
  JS

  - gets an attribute from a node (e.g., `category`, `lang`)

Navigating node tree example

```
// 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") {
    var title = books[i].getElementsByTagName("title")[0].firstChild.nodeValue;
    var author = books[i].getElementsByTagName("author")[0].firstChild.nodeValue;

    // make an XHTML <p> tag based on the book's XML data
    var p = document.createElement("p");
    p.innerHTML = title + ", by " + author;
    document.body.appendChild(p);
  }
}
```
JS
```

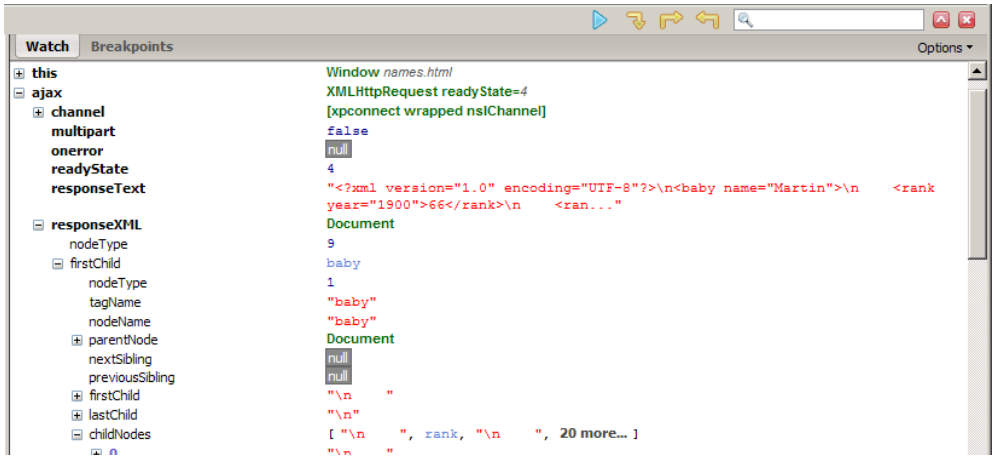
A historical interlude: why XHTML?

- 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

Why XML in AJAX?

- most data you want are provided in XML
  - the *de facto* universal format
- the browser can already parse XML (i.e., XHTML) into DOM objects
  - DOM only defined for XML-based formats, may not map directly to another format
- would have to manually parse a different format
  - simple formats can be parsed manually from `ajax.responseText`
  - most data are easier to manipulate as DOM objects than to parse manually

Debugging `responseXML` in Firebug



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