Computer Science 75 Lecture 3: September 28, 2009
Fall 2009 Andrew Sellergren
Scribe Notes

Contents

Computer Science 75
Fall 2009
Scribe Notes

Lecture 3: September 28, 2009
Andrew Sellergren

1 Announcements (1:00–2:00)

- Project 1's specification is now available on the Projects page of the course website. Projects, keep in mind, are designed to be tackled over the course of several weeks. Don't leave it all to the last minute—instead, tackle 10 hours this week, 10 the next, and so on. This project is a noticeable step up from Project 0 in terms of difficulty.

- Tonight, Sid will be introducing Project 1 and some related material if you stick around for section. If you can't make it, it will be recorded and posted online.

- Next week, we'll be joined by a guest lecturer, David Heitmeyer, who teaches CSCI E-153, a entire course devoted to XML-based web development.

2 More on Sessions (3:00–52:00)

- Recall from last time that sessions are a form of server-side data storage which allow us as programmers to remember information associated with a given user. Session objects are stored in serialized form in the `/tmp/` directory on the server.

- What is actually sent to the user is the key—the very large random number which identifies the user. By default, the name of the cookie is `PHPSESSID`.

- To use the preloaded data in `$_SESSION`, you must call the `session_start()` function at the top of your file. Be careful: you need to call this function before *any* output is sent to the browser. This is because it actually triggers a header called Set-Cookie being sent to the browser. Your best bet is to call it at the very top of your file.[1]

- Sessions obviously have great value when designing login modules. Today we'll be looking at four different versions of a login module, all of which you can test out here. As we'll see, sessions allow us to maintain state despite HTTP being a stateless protocol. That is, once the browser has made an HTTP request and downloaded all the necessary files, the connection is actually closed. Sessions enable data to persist between connections.

- When we click the first of our four login versions, we can login using the username `jharvard` (since we've looked at the source code) and the password `crimson`. We are then redirected back to the login page which now tells us that we are logged in. We can refresh the page and even disconnect and reconnect our internet connection to see that the page will continue to remember that we are logged in.

[1]The one exception is if you have custom objects stored in a session, in which case you need to load your class definitions before starting the session.

- The logic that displays this message is as follows:

  ```
  <? if ($_SESSION["authenticated"]) { ?>
    You are logged in!
   <br />
   <a href="logout.php">log out</a>
  <? } else { ?>
    You are not logged in!
  <? } ?>
  ```

  By this logic, if the `authenticated` index of `$_SESSION` is set to true, then announce that the user is logged in. Knowing this, we can deduce that the page which `login1.php` is submitting data to must be validating the login information somehow and then setting the `authenticated` index of `$_SESSION` to true. Lo and behold, that's what we find when we examine the source code of `login1.php`:

  ```
  <?
      /**
       * login1.php
       *
       * A simple login module.
       *
       * Computer Science E-75
       * David J. Malan
       */

      // enable sessions
      session_start();

      // suppress notices
      ini_set("display_errors", true);
      error_reporting(E_ALL ^ E_NOTICE);

      // were this not a demo, these would be in some database
      define("USER", "jharvard");
      define("PASS", "crimson");

      // if username and password were submitted, check them
      if (isset($_POST["user"]) && isset($_POST["pass"]))
      {
          // if username and password are valid, log user in
          if ($_POST["user"] == USER && $_POST["pass"] == PASS)
          {
              // remember that user's logged in
              $_SESSION["authenticated"] = TRUE;
  ```

```
                // redirect user to home page, using absolute path, per
                // http://us2.php.net/manual/en/function.header.php
                $host = $_SERVER["HTTP_HOST"];
                $path = rtrim(dirname($_SERVER["PHP_SELF"]), "/\\");
                header("Location: http://$host$path/home.php");
                exit;
            }
        }
?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xh

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Log In</title>
  </head>
  <body>
    <? if (count($_POST) > 0) echo "INVALID LOGIN"; ?>
    <form action="<? echo $_SERVER["PHP_SELF"]; ?>" method="post">
      <table>
        <tr>
          <td>Username:</td>
          <td><input name="user" type="text" /></td>
        </tr>
        <tr>
          <td>Password:</td>
          <td><input name="pass" type="password" /></td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Log In" /></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

As you can see, we're validating the login credentials not against a database, but against hard-coded values, which we've defined as constants, for simplicity's sake. You can also see that the form on this page is submitting to itself. The alternative approach is of course to have this page submit to a different page altogether. One downside of this two-page approach is that if the user has made some sort of error in the form, chances are all his data will be lost if he has to press the Back button to return and resubmit. From a user experience standpoint, generally it's ideal to do some validation on the same page so that the user's entered data isn't at risk of being lost.[2].

- Using the single-page approach, we have two distinct states: either the user is visiting for the first time or he has already submitted his login data, in which case we'll perform some validation and possibly redirect him. If they've made an error in their entered data, we can easily redisplay the data and ask the user to correct it.

- The first conditional check in the above file is to determine if the user arrived at this page after submitting a form or for the first time. If they've arrived for the first time, then we want to simply display the XHTML content. If they've submitted the form already, we want to check their login information. We accomplish this using the function `isset()`, which in this case checks whether the username and password have been defined, as by a form submission. If this call to `isset()` returns false, then we know the user is visiting for the first time.

- Notice that we're not hard-coding in the name of the file to fill in the `action` attribute of the form. Instead, we're referencing the `$_SERVER` superglobal which has an index called `PHP_SELF` that gives the name of the current executing file. This is a good design decision considering that we can change the name of the file without breaking the code. Arguably, there's a slight performance hit for having to access this superglobal, but overall it's probably worth it for portability's sake.

- In the `value` attribute of the username field, we have an `echo` statement which serves to prepopulate the field if the user has gotten here after entering in a username (if his form submission was rejected due to an error, for example).

- If we submit the form and sniff the network traffic, we can see that the ways in which `GET` data and `POST` data are transmitted are not too different. The `POST` string resembles a `GET` string in how it is formed (with ampersands separating parameters). The fundamental difference is that `GET` strings have an upper-bound on their length whereas `POST` strings are transmitted following a Content-Length header which specifies just how many bytes the server should be expecting.

- Question: is there a way to encrypt the password? We could do so using JavaScript or we could encrypt the entire transaction using SSL.

- Quesiton: is it better to use `POST` or `GET`? For sending sensitive data, no doubt it's better to use `POST` to keep the data out of the URL and out of the browser's cache and history. For non-private data, it's often better to use `GET` so that URLs can be bookmarked.

[2]Even with the two-page approach, however, we can do this same-page validation using JavaScript (although client-side validation should always be backed up with server-side validation!)

- After checking if the username and password fields are not empty, we check that the values provided by the user are equal to the hardcoded username and password we specified as constants earlier. If they *are* equal, then we want to somehow remember that the user is logged in. We do this by setting the `authenticated` index. Realize that we could've called it anything, but this name is at least meaningful.
- The second part of the login validation process is to redirect the user to the homepage if he is logged in. We do this using these lines of code:

  ```
  // redirect user to home page, using absolute path
  // http://us2.php.net/manual/en/function.header.php
  $host = $_SERVER["HTTP_HOST"];
  $path = rtrim(dirname($_SERVER["PHP_SELF"]), "/\\");
  header("Location: http://$host$path/home.php");
  exit;
  ```

  We might think about doing this the "easy" way by simply redirecting to `home.php`. However, the specification for HTTP headers indicates that the URL should be fully qualified, meaning that it includes the full domain name and the path. These lines of code are simply using the superglobal `$_SERVER` to retrieve the domain name and the path and append them to the redirect URL.
- To logout, there are a few things we need to do, all of which are taken care of in `logout.php`:

  ```
  <?
      /**
       * logout.php
       *
       * A simple logout module for all of our login modules.
       *
       * Computer Science E-75
       * David J. Malan
       */

      // enable sessions (so that we can destroy them)
      session_start();

      // delete cookies, if any
      setcookie("user", "", time() - 3600);
      setcookie("pass", "", time() - 3600);

      // log user out
      setcookie(session_name(), "", time() - 3600);
      session_destroy();
  ```

```
?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xh

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Log Out</title>
  </head>
  <body>
    <h1>You are logged out!</h1>
    <h3><a href="home.php">home</a></h3>
  </body>
</html>
```

First, we enable sessions, as usual. Then we want to destroy any cookies that have been set. We do this by setting their expiration date, the second of the two parameters passed to `setcookie()`, to be a time in the past. We do this with the default cookie specified by `session_name` in addition to calling `session_destroy()`. Once we've done this, the previous contents of `$_SESSION` will no longer be accessible.

- As we've already seen, prepopulating the username field is a simple matter of an `echo` statement, so frankly there's no excuse for a website not to do this. This prepopulation of the username field is actually the only improvement between `login1.php` and `login2.php`.

- The third version of our login module has one extra line of code:

```
// save username in cookie for a week
setcookie("user", $_POST["user"], time() + 7 * 24 * 60 * 60);
```

This call to `setcookie()` comes after we set the `authenticated` index of `$_SESSION`. Previously, all of the cookie handling was done for us by PHP. If we want some control over what data is stored persistently on the user's hard drive, we can call `setcookie()`manually as we do here. In this case, we're storing what the user entered into the username field. The expiration date we set is a value in seconds from the current time. If we do the math, we can see that this cookie will expire in a week.

- If we login using version 3 of our login module and then close and reopen our browser, we can see that the username field will be prepopulated with our username because it has been stored on the local filesystem in a cookie. This value, of course, isn't being accessed from `$_POST` because we're visiting the page for the first time. Instead, we access it from the `$_COOKIE` superglobal:

```
value="<? echo ($_POST["user"]) ? $_POST["user"] : $_COOKIE["user"]; ?>"
```

Here, we're setting the `value` attribute of the username field using a ternary operator (i.e. `? :` can be translated to if-else). What we're saying is, fill it in with the value from `$_POST`, if it exists, otherwise fill it in with the value from `$_COOKIE`. If the value in `$_POST` exists, it's because the user has submitted the form to get there, so we want to use the most recent value that he entered. Otherwise, if it's not set, then he's visiting the page for the first time, so use the value stored in the cookie.

- Notice that at the top of these files we've been making a call to the function `error_reporting()` to suppress notices. Generally speaking, this is a security measure so that if your website's code breaks, you don't inadvertently give away information to a user as to how they might exploit your website. In our case, we're doing so because we are accessing in certain places variables which we aren't sure are actually set. If they aren't set, this will produce a notice that will interfere with our program. If we enable notices and clear our cookies, we can see that the username field will be populated with a PHP notice because its default value, a reference to `$_COOKIE`, isn't set. Another way to suppress this notice besides a call to `error_reporting()` is to place the `@` symbol in front of the line of code.
- The three levels of errors in PHP are notices, warnings, and errors. Notices and warnings, which are simply informational, are less severe than errors, which will actually halt execution of your program.
- When writing library code, calling `error_reporting()` is not the best approach since there's no guarantee that you'll be able to control this setting on other servers. Note that the argument we provide to `error_reporting()` is two bit flags combined with the XOR operator (`^`). If you have no idea what that means, don't worry about it!
- Version 4 of the login module exhibits a security vulnerability. It stores not only the username but also the password in a cookie. Anyone who knows this would be able to poke around on the local filesystem and find the stored password, since it's unencrypted.
- Question: are name collisions with cookies possible? Within the same domain, yes. Cookies are associated with domain names, so the fact that all your domains live on our server is not a problem.
- Question: how easy is it to forge a session ID? No need to forge it, you can simply hijack a valid one by sniffing network traffic on a public internet connection. This is assuming, of course, that the connection is non-SSL.

3 XML (0:00–1:00, 52:00–108:00)

3.1 Introduction

- We begin our exploration of XML with a look at the course's own website, which itself uses XML files as a way of separating data from aesthetics.

- Basically, what we found is that writing a few lines of code to parse these XML files was easier than manually editing the website's XHTML every time we needed to update it.

- Notice that we have XML files for sections, lectures, resources, software. These are all stored in the `/etc/xml/` directory, although that's not really important.

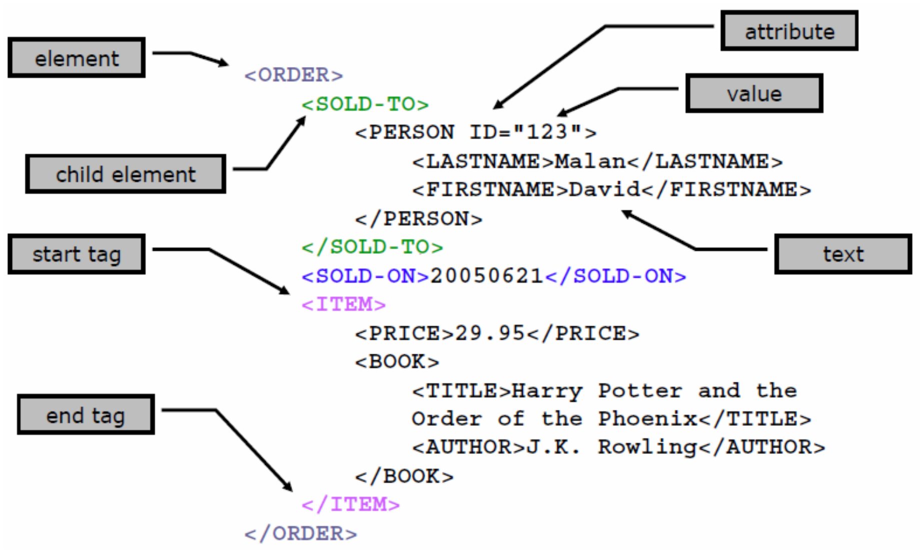
- In the `lectures.xml` file, we see that there is a `<lectures>` tag at the top, within which there are individual `<lecture>` tags. Each of these lecture tags has several elements of its own, namely `<title>`, `<dates>`, and `<resources>`. Basically, long story short, this is a way of representing the data in an organized, human-readable, and machine-readable way.

- Three Aces' menu, which you'll be tasked with representing in XML format, is quite complicated. Of course, we don't want this to be an exercise in tedium, so you won't have to implement every single item on the menu, but only a representative subset—three or more items from each category. Because of the corner cases, you'll need to make some design decisions which allow your schema to be flexible.

- The mechanism by which you implement the online ordering system is primarily up to you. You have at your disposal, of course, HTML forms which will allow you to collect user input.

- Think of XML as being much like XHTML, except there's no fixed set of tags which you can use. The number and types of tags are up to you!

- There are languages like XML Schema and DTD which allow you to define what your XML has to look like in order to establish consistency. But we're not going to worry about that in this course.



```
<ORDER>
    <SOLD-TO>
        <PERSON ID="123">
            <LASTNAME>Malan</LASTNAME>
            <FIRSTNAME>David</FIRSTNAME>
        </PERSON>
    </SOLD-TO>
    <SOLD-ON>20050621</SOLD-ON>
    <ITEM>
        <PRICE>29.95</PRICE>
        <BOOK>
            <TITLE>Harry Potter and the
            Order of the Phoenix</TITLE>
            <AUTHOR>J.K. Rowling</AUTHOR>
        </BOOK>
    </ITEM>
</ORDER>
```

- This snippet of XML seems to represent some kind of purchase order, perhaps from Amazon. Amazon, as you've probably noticed, sells a lot of merchandise on behalf of third-party vendors. How might it transmit data to them about purchases? Well, one way it might do so is using XML, which is a great language for standardizing data since it's so easy to parse. Because so much useful metadata is stored along with the actual data, XML is excellent for creating configuration files and file format specifications, for example.

- In the above example, `ORDER` represents the root *element* of the document, because it contains everything within it. Elements must be opened and closed with tags. Unlike XHTML, XML need not be all lowercase although its case must be consistent.

- `SOLD-TO` is called a *child* of `ORDER` because it is nested within it. Its indentation isn't necessary for the XML to be valid, but it helps for human readability.

- *Attributes* should be familiar to you from XHTML. They are specified within angle brackets of the open tag and their values must be enclosed in quotation marks.

- XML stands for extensible mark-up language. "Mark-up" signifies that it is not a language which executes any statements. "Extensible" signifies that features can be added to it without breaking existing applications that rely on it. In our example, above, we could add `INITIAL` and `ADDRESS` tags within the `PERSON` tag with minimal or no effect to programs designed for the previous version of our XML. One other selling point of XML is its human readability.

- Although it might seem like a waste of bandwidth and bytes to transmit all this data across the wire, consider that it's probably worth it to save resources in having to parse a proprietary, albeit smaller, file format.
- Let's take a look at a slightly more complicated XML file:

  ```
  <?xml version="1.0" encoding="UTF-8"?>

  <!-- This is an XML document that describes students -->
  <students>
      <student id="0001">
          <name>Jim Bob</name>
          <status>graduate</status>
          <dorm/>
          <major>Computer Science &amp; Music</major>
          <description>
                  <![CDATA[ <h1>Jim Bob!</h1>
                  Hi my name is jim. I look like
                  <img src="jim.jpg"> ]]>
          </description>
      </student>
      <student id="0002">
      ...
      </student>
  </students>
  ```

  This is a student database of sorts, represented in XML. XML is useful in this context if the database only has a few hundred students or perhaps if a much larger database needs to be dumped into a more universal format that could be read by another system.
- Notice this first line of code:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  ```

  This is the XML declaration, which is optional. Unfortunately, it begins and ends with PHP short tags, which we'll confuse the PHP interpreter in some cases. For that reason, some find it best to leave it out. When included, though, the XML declaration simply says "Hi, I'm an XML document!" Note that it needs to be at the very top and the very left of a document in order to be treated properly by an XML parser. The `encoding` attribute simply specifies a standard for representing letters and other special characters with numbers. `UTF-8` is a superset of ASCII.
- The convention of XML is for everything to be surrounded by the open and close tags of a single root element. In this case, it's the `<students>` tag. Beneath that root element, you can have as many children as you want.

- Elements must begin with an open tag and end with a close tag. Within the open tag, attributes can be specified. Multiple attributes must be separated from each other by at least one space and must have unique names for a single element.
- Question: "Jim Bob" is a text node as we'll see in a little while.
- Question: although `id` and `name` are token attributes, the only way to ensure unique values for `id` among different elements is manually using your program which actually writes the XML.
- One final gotcha regarding element names: they must start with letter or underscore and contain only letters, numbers, hyphens, periods, and underscores.
- Below are several content models for XML you should be aware of:
  - Element Content

    ```
    <student>
        <status>...</status>
    </student>
    ```

  - Parsed Character Data (a.k.a. PCDATA, Text)

    ```
    <name>Jim Bob</name>
    ```

  - Mixed Content

    ```
    <name>Jim <initial>J</initial>Bob</name>
    ```

  - No Content

    ```
    <dorm />
    ```

- The `<![CDATA[` and `]]>` tags denote *character data*. This tells the parser to gobble up everything between these tags but *not* to parse it. This is useful so that you can embed XHTML within XML without confusing the parser. Unlike CDATA, PCDATA is parsed.
- Mixed content we've seen in the context of XHTML in which `p`, `b`, and `i` tags are used within other tags to create paragraphs, bold text, and italic text, respectively.
- What about the `<dorm />` tag? This is a so-called empty element. This is useful in the case that you want to convey that you didn't forget the dorm data, but merely that it was absent or missing in this particular case. We've seen empty elements in XHTML already, the most familiar being the `<br />` tag. Note: some browsers still choke on `<br/>` so it's best to include the extra space, like so: `<br />`.

- Question: can you constrain what attributes and elements are allowed in an XML document? Yes, using an XML Schema or DTD which are read by the parser. Both of these are beyond the scope of this course, however.
- Some constraints on attributes:
  - Name
    - Must start with letter or underscore and can contain only letters, numbers, hyphens, periods, and underscores
  - Value
    - Can be of several types, but is almost always a string
    - Must be quoted
    - Cannot contain `<` or `&` (by itself)
- Most of XML data is of the type PCDATA, which has the following constraints:
  - Text that appears as the content of an element
  - Can reference entities
  - Cannot contain `<` or `&` (by itself)
- Entities are another useful construct of XML which are familiar from HTML:
  - Used to "escape" content or include content that is hard to enter or repeated frequently, somewhat like macros
  - Five pre-defined entities `&amp; &lt; &gt; &apos; &quot;`
  - Character entities can refer to a single character by unicode number, e.g., `&#x00A9;` is ©
  - Must be declared to be legal, e.g. `<!ENTITY nbsp"&#160;">`
  - Cannot refer to themselves
- We've seen CDATA in the context of embedding JavaScript within XHTML to prevent the XHTML parser from getting confused:

```
<html>
<body>
<script type="text/javascript">
// <![CDATA[

if (x < 3)
    document.write("x is less than 3!");

// ]]>
</script>
</body>
</html>
```

The CDATA tags tell the XHTML parser not to interpret what's enclosed as XHTML. The `//` tells JavaScript not to interpret the CDATA tags as JavaScript. Ugly, but it works.

- Some properties of CDATA:
  - Parsed in "one chunk" by the XML parser
  - Data within is not checked for subelements, entities, etc.
  - Allows you to include badly formed markup or character data that would cause a problem during parsing, e.g. including HTML tags in an XML document
- As in XHTML and PHP, comments are possible within XML:
  - Can include any text inside a comment to make it easier for human readers to understand your document
  - Generally not available to applications reading the document
  - Always begin with `<!--` and end with `-->`
  - Cannot contain `--` (thus they can't be nested)

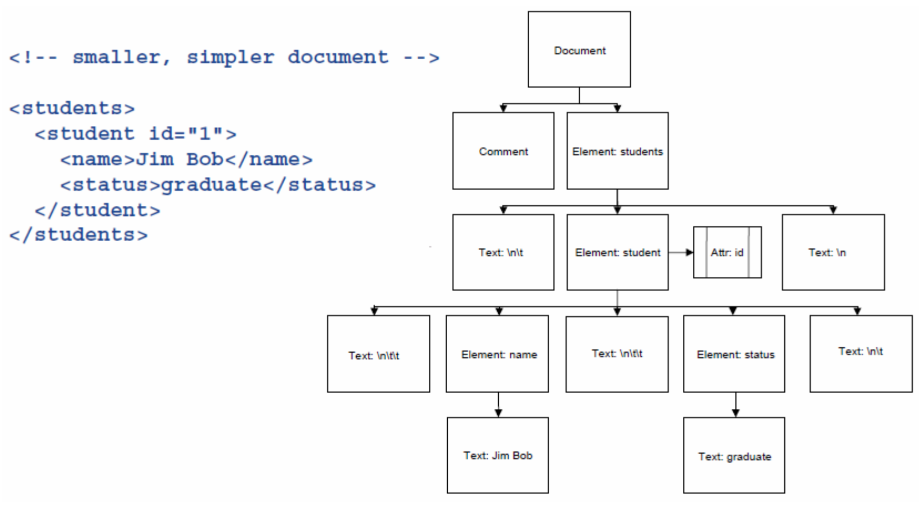
3.2 SimpleXML

- So how do we interact with this XML, particularly using PHP? Turns out that recent versions of PHP come with an API called SimpleXML which allows just that.
- Before we dive into SimpleXML, let's start by constructing an XML document to represent lectures in this course:

```
<lectures>
  <lecture number="2">
    <title>PHP, Continued</title>
    <date>21 September 2009</date>
    <resources>
      <resource name="Slides" href="2/lectures.pdf"/>
      <resource name="Source Code" href="2/src/"/>
    </resources>
  </lecture>
  <lecture number="3">
    <title>XML</title>
    <date>28 September 2009</date>
    <resources>
      <resource name="Slides" href="3/lectures.pdf"/>
      <resource name="Source Code" href="3/src/"/>
    </resources>
  </lecture>
</lectures>
```

Even in this basic XML document, we've made some significant design decisions. Why should `title` be an element while `number` is an attribute? Instinctively, we know that elements can be extended later to have their own children, whereas attributes cannot. You might think of elements as nouns and attributes as adjectives. In short, it's open to interpretation, but know that attributes are sometimes more easily accessible in a variety of programming languages.

- Question: `id` is generally best as an attribute, although the specification for RSS includes a field called `guid`, which is an element. You could make good arguments for both.
- Question: generally, it's a good idea to have identical structure among like-named elements. Otherwise, you would defeat the purpose of XML, which is to define a data structure which can be easily understood.
- Know that when you view an XML file in Firefox, it will be pretty-printed and displayed with collapsible elements. This is **not** a feature of the XML itself, but rather a visualization bundled with Firefox. item The world of XML effectively revolves around something called the *Document Object Model* (DOM). This is the very tree structure we alluded to earlier:



```
<!-- smaller, simpler document -->

<students>
  <student id="1">
    <name>Jim Bob</name>
    <status>graduate</status>
  </student>
</students>
```

In this diagram, each of the child elements is shown hierarchically below its parent. At the very top is the element from which all others stem—the `document` element. This is to be distinguished from the `root` element, however, which is `students` in this case. The `document` element is useful particularly in this context because it allows us to represent `students` as well as the introductory comment as siblings.

- What's the deal with the other elements besides `students` and `student`? Technically, although whitespace is ignored by us as humans and by the parser as well, we should include it in our DOM to be precise. So "`\n\t`" is listed as a sibling of `student`, for example.
- Why is the `id` attribute on the same level as the node it describes? Since it's not a child, it doesn't really make sense to show it beneath its element.
- Conceptually, what SimpleXML is going to do is take a document like that on the left and hand us a hierarchical structure like that on the right.
- Let's start manipulating our sample XML document using SimpleXML:

  ```
  <?
      // parse XML file
      $xml = new SimpleXMLElement(file_get_contents("lectures.xml"));
  ?>

  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <title>Lectures</title>
    </head>
    <body>
    <pre>
      <? print_r($xml); ?>
    </pre>
    </body>
  </html>
  ```

  `SimpleXMLElement` is a class which comes built in to PHP. To instantiate a new object of this type, we use the `new` keyword and call the constructor, a function which has the same name as the class itself. In the case of `SimpleXMLElement`, we need to pass in a string which contains a well-formed XML document. To convert our XML document to a string, we'll use the `file_get_contents()` function.
- Just to make sure we've loaded the file up properly, let's try dumping the contents of `$xml` using the `print_r()` function. When we do so, we get the following output to our browser:

  ```
      SimpleXMLElement Object
  (
      [lecture] => Array
          (
  ```

```
[0] => SimpleXMLElement Object
    (
        [@attributes] => Array
            (
                [number] => 2
            )

        [title] => PHP, Continued
        [date] => 21 September 2009
        [resources] => SimpleXMLElement Object
            (
                [resource] => Array
                    (
                        [0] => SimpleXMLElement Object
                            (
                                [@attributes] => Array
                                    (
                                        [name] => Slides
                                        [href] => 2/lectures.pdf
                                    )

                            )

                        [1] => SimpleXMLElement Object
                            (
                                [@attributes] => Array
                                    (
                                        [name] => Source Code
                                        [href] => 2/src/
                                    )

                            )

                    )

            )

    )

[1] => SimpleXMLElement Object
    (
        [@attributes] => Array
            (
                [number] => 3
            )
```

```
                    [title] => XML
                    [date] => 28 September 2009
                    [resources] => SimpleXMLElement Object
                        (
                            [resource] => Array
                                (
                                    [0] => SimpleXMLElement Object
                                        (
                                            [@attributes] => Array
                                                (
                                                    [name] => Slides
                                                    [href] => 3/lectures.pdf
                                                )

                                        )

                                    [1] => SimpleXMLElement Object
                                        (
                                            [@attributes] => Array
                                                (
                                                    [name] => Source Code
                                                    [href] => 3/src/
                                                )

                                        )

                                )

                        )

                )

        )

)
```

Whew, that's a lot. But notice the indentation, which seems to suggest a hierarchical tree structure.

- Let's try printing out just the titles of the lectures:

```
foreach($xml->lecture as $lecture)
{
    echo $lecture["number"];
    echo "<br />";
}
```

Not too impressive, but if we run this, we get the numbers 2 and 3 outputted to the browser. Notice that we can get an array of all the `lecture` tags that are children of the root element using the `->` operator. Then we can access the attributes of a given element using bracket notation.

- Next, let's print out the titles of the lectures, as well:

```
foreach($xml->lecture as $lecture)
{
    echo "Lecture ". $lecture["number"];
    echo "<br />";
    echo $lecture["title"];
    echo "<br />";
    echo "<br />";
}
```

Oops, this didn't work because `title` is not an attribute, but rather an element. Let's try this instead:

```
foreach($xml->lecture as $lecture)
{
    echo "<b>";
    echo "Lecture ". $lecture["number"];
    echo "</b>";
    echo "<br />";
    echo $lecture->title;
    echo "<br />";
    echo "<br />";
}
```

Now let's get a little fancier by printing out the lecture resources as an unordered list:

```
foreach($xml->lecture as $lecture)
{
    echo "<b>";
    echo "Lecture ". $lecture["number"];
    echo "</b>";
    echo "<br />";
    echo $lecture->title;

    echo "<ul>";
    foreach ($lecture->resources->resource as $resource)
    {
        echo "<li>";
        echo "<a href='{$resource["href"]}'>";
```

```
            echo $resource["name"];
            echo "</a>";
            echo "</li>";
        }
        echo "</ul>";
}
```

Here, we're printing out a list of the resources for each lecture and linking to them using the `href` attribute. Remember that the curly braces are necessary so that a value within an array can be interpolated properly between double quotes.

- For those less comfortable with object-oriented programming (OOP), there exists a function called `simplexml_load_file` which achieves the same as the constructor.

3.3 RSS and XPath

- One real-world application that employs this kind of XML parsing is HarvardEvents, created by David. This leverages the universal standard for RSS feeds (which are implemented as XML) to aggregate event announcements from groups all over Harvard's campus. The XML specification happens to also live on Harvard's campus. Check out this skeleton of an RSS feed:

```
<rssversion="2.0">
    <channel>
        <title></title>
        <description></description>
        <link></link>
        <item>
            <guid></guid>
            <title></title>
            <link></link>
            <description></description>
            <category></category>
            <pubDate></pubDate>
        </item>
        [...]
    </channel>
</rss>
```

Each `channel` belongs to a single organization, e.g. The Crimson, and each `item` is one of many articles announced by that organization. Creating HarvardEvents was a simple matter of collecting all the URLs of the RSS feeds and then parsing those feeds using the SimpleXML API.

- XPath is a language that allows us to query XML. Let's learn by way of example. Instead of using the `foreach` syntax to get at the `lecture` tags, we can use XPath:

  ```
  <?
      $lectures = $xml->xpath("/lectures/lecture[@number=2]");
  ?>
  ```

  This particular XPath query will actually return an array, but because we've specified the `number` attribute as 2, the array will only have a single element which corresponds to lecture 2. More on XPath next week!