Section 2: PHP, Continued

XML

XML, or eXtensible Markup Language, is a markup language used for data that is very similar to XHTML. (In fact, all valid XHTML should conform to XML standards.) All elements that are opened must be closed, and attribute values must be enclosed in quotes. You may specify any arbitrary name for elements and attributes; you are not limited to a certain set of tags, as with XHTML.

```
<sushi>
	<fish name="Tuna">
		<available>true</available>
		<origin>North Sea</origin>
		<variations>
			<variation name="With Wasabi">
				<price>5.50</price>
			</variation>
			<variation name="Without Wasabi">
				<price>5.25</price>
			</variation>
		</variations>
	</fish>
	<fish name="Squid">
		<available>false</available>
		<origin>Mediterranean Sea</origin>
		<variations>
			<variation name="With Wasabi">
				<price>8.50</price>
			</variation>
			<variation name="Whole">
				<price>12.20</price>
			</variation>
		</variations>
	</fish>
</sushi>
```

In the example above, the XML document has one root *element*, "sushi," as well as multiple "fish" elements underneath it. Each fish has a "name" *attribute*, which specifies the name of the fish. Notice that the "variations" element inside each "fish" is also home to one or more "variation" elements, each of which represents a variation of sushi.

Accessing XML from PHP

PHP provides several frameworks for using and manipulating XML documents. This week, and for the project, we'll be using SimpleXML (http://www.php.net/simplexml).

Every time we want to read an XML file in a PHP script, we initialize a new SimpleXMLElement object with a string containing the XML document as its sole parameter:

```
$xml = new SimpleXMLElement(file_get_contents('sushi.xml'));
```

(file_get_contents() reads a file into a string and returns that string.)
$xml is now an object with the following kind of structure:

```
SimpleXMLElement Object
```

```
(
    [item] => Array
        (
            [0] => SimpleXMLElement Object
                (
                    [@attributes] => Array
                        (
                            [name] => Tuna
                        )
                    [available] => true
                    [origin] => North Sea
                    [variations] => SimpleXMLElement Object
                        (
                            [variation] => Array
                                (
                                    [0] => SimpleXMLElement Object
                                        (
                                            [@attributes] => Array
                                                (
                                                    [name] => With Wasabi
                                                )
                                            [price] => 5.50
                                        )
                                    [1] => SimpleXMLElement Object
                                        (
                                            [@attributes] => Array
                                                (
                                                    [name] => Without Wasabi
                                                )
                                            [price] => 5.25
                                        )
                                )
                        )
                )
          ...
        )
)
```

SimpleXMLElement represents the entire XML document in this manner, exposing a number of functions that you can use to read (and write to) it.

Common techniques
There are some common techniques you can use to go through XML documents.

Iteration and accessing attributes and child elements
We can "walk," or traverse, the elements in the SimpleXMLElement tree with PHP's "foreach" construct:

```
foreach ($xml->fish as $fish) {
       //Your code goes here
}
```

Within that foreach construct, we can refer to the attributes and child elements of each "fish." *Attributes* are referred to using PHP's associative array operator:

```
foreach ($xml->fish as $fish) {
       echo "This fish is called ".$fish['name'].".<br />";
```

```
}
```

If we run this, we would get:

```
This fish is called Tuna.
This fish is called Squid.
```

Child *elements* are referred to using the "->" operator:

```
foreach ($xml->fish as $fish) {
        echo "This fish hails from the ".$fish->origin.".<br />";
}
```

This generates:

```
This fish hails from the North Sea.
This fish hails from the Mediterranean Sea.
```

We can also access the grandchildren of elements by accessing the child element of the child element. (We can also use numerical indices to access specific child elements):

```
echo $fish->variations->variation[0]->price . "<br />";
```

The code above prints out the price of the first variation of each fish:

```
5.50
8.50
```

Using XPath to find and access child elements

Using the method above, we had to iterate through the XML document to find anything; this is not ideal for every web page. We can use XPath (http://www.w3schools.com/xpath/) to perform a query against XML documents, so that we can jump within the document to the sections we need.

To retrieve just the "fish" element referring to Squid, we can use the following path:

```
/sushi/fish[@name="Squid"]
```

Just like we use forward-slashes to represent directory structures, we use forward-slashes in XPath to refer to child elements. The text inside the square brackets is the "predicate" – the test that determines which data to return. In this case, we're selecting the "fish" element whose "name" attribute happens to equal "Squid."

We can use XPath in PHP using the "xpath" function of the SimpleXMLElement.

```
$squidelement = $xml->xpath('/sushi/fish[@name="Squid"]');
```

The code above returns an object representing the "fish" element for Squid. To return an array of variations of Squid, we can do the following:

```
$squidvariations = $xml->xpath('/sushi/fish[@name="Squid"]/variations/variation');
```

We can then iterate over $squidvariations to perform an operation on all the variations of Squid:

```
foreach ($squidvariations as $variation) {
	echo 'Squid '.$variation['name'].' costs $'.$variation->price."<br />";
}
```

Note that even when using XPath, attributes and elements are referred to in the same way as in the iterative approach. The code above displays:

```
Squid With Wasabi costs $8.50
Squid Whole costs $12.20
```

Using Sessions

HTTP is a "stateless" protocol, meaning that each and every request to a web server is separate from every other request. This would make implementing, say, an online shopping site very difficult; that is why HTTP has cookies. Cookies allow certain data (usually a large random number) to be given to the web browser, which the browser then stores and sends to the server in subsequent requests. This allows the web server to keep track of users, and allows for the implementation of shopping carts, logins, and other "stateful" websites.

PHP allows developers to use cookies seamlessly with a feature called "sessions." This allows each user to have a "session" to themselves, in which a special superglobal, $_SESSION, is defined and retained across subsequent requests. $_SESSION is an associative array (hash table) that can contain anything (XML, strings, integers, etc.)

Using sessions is simple. It involves the following two steps:

1. Make a call to the session_start() function at the top of the script.
2. Read or write to $_SESSION in the script.

That's all that's necessary – and the contents of $_SESSION will be automatically saved, as long as you make a call to session_start() at the top of any script in which you want to refer to $_SESSION. Know that the contents of $_SESSION you define in, say, page1.php are even available in, say, page2.php.

Shopping Cart example

The Shopping Cart example (http://cs75.net/sections/2/shopping/; source at http://cs75.net/sections/2/src/shopping/) is implemented in three files. index.php is a simple HTML form that passes data to add.php, add.php adds an item to the shopping cart, and cart.php displays the current state of the cart.

<u>add.php</u>

Note the call to session_start() at the top of the script. This initializes the session for this script. The script then performs some simple validation on the data passed into the script (from index.php), and assigns the data to an associative array, $orel. $orel is then appended to another associative array, $_SESSION['cart'] (If $_SESSION['cart'] hasn't yet been defined, we define it as an array). Since $_SESSION['cart'] is part of the session variable, we can expect this data to be preserved for subsequent requests to the web server.

<u>cart.php</u>

Again note that this page too calls session_start(). This ensures that cart.php runs in the same session as add.php, so that our data in $_SESSION from add.php can be accessed again. Since there is a possibility

that the $_SESSION['cart'] variable is not defined in the current session (if the user accessed cart.php before add.php, for example), we first check that $_SESSION['cart'] is defined. Then, we simply iterate over the $_SESSION['cart'] variable to read out any items that we have added in add.php.

You will need to implement a similar, albeit more sophisticated, shopping cart for Project 1. Think about how to store items in the $_SESSION variable (we used associative arrays here, but you can also define and use a PHP class for each item), and how to remove and modify items already in the shopping cart. You will also want to make a more advanced version of index.php that uses the XML menu you created to read in prices and other data; letting users pick their own prices probably isn't such a good idea!