Computer Science 75
Fall 2009
Scribe Notes

Lecture 10: November 23, 2009
Andrew Sellergren

Contents

Computer Science 75
Fall 2009
Scribe Notes

Lecture 10: November 23, 2009
Andrew Sellergren

1 Announcements (0:00–2:00)

- David is once again a walking advertisement for his other class, CS 50.
- As promised, David fixed his hack in Shuttleboy so that it now uses Ajax!

2 The HarvardFood API: Underneath the Hood (2:00–61:00)

2.1 Examining the Data

- We spoke last time on a high level about the HarvardFood API. Of course, `food.cs50.net` isn't much of an API, but rather an explanation of how the data is gathered from the Harvard University Dining Services (HUDS) menu. David managed to find several other universities who use the same third-party service to display their menu, but he found no indication that there was a standard XML file or other data source that he might directly interface with. So he turned his attention to screen-scraping. Today, we'll actually be looking underneath the hood at the PHP code that accomplishes this screen-scraping which, perhaps surprisingly, is less than 100 lines long. We'll be using some XPath and XML, including a detail of XML which we previously glossed over: namespaces.
- The purpose of this exercise is not to make you more familiar with the HUDS menu in particular, but rather to make you more familiar with external data sources in general: how to interface with them and reverse engineer them.
- One of our first discoveries from last time was that the HUDS menu page took a number of parameters, including `date`, `type`, and `meal`. as input via a GET string. This is great news for us since it makes it easy for us to replicate. `date` is pretty self-explanatory, `type` seems to return the whole menu if 30 is specified, and `meal` represents breakfast, lunch, and dinner as 0, 1, and 2, respectively. David found all this out with a little trial and error. He then made an arbitrary (but reasonable) design decision to query once a day for all the meals for the following week. This would open up some interesting possiblilties: for example, the ability to know on Monday when pizza would next to be served that week. With some more trial and error, he found that it was indeed possible to query a week in advance and even a little farther. Beyond that, HUDS's data isn't published, so David realized he would need to run a `cron` job so that he could automate the data collection in the future.
- If we return to the current day, we can begin to think about what data on the site is actually of interest to us. Obviously, the menu items. There's also nutritional information and portion sizes, but David made the assumption that those would be using this API would only care about what and when, not how much and how healthy. So for now we'll focus only on the menu items.

- To examine the actual data, we begin by right clicking on a menu item and selecting Inspect Element, a Firebug option. This will open the Firebug window and take us immediately to the DOM node that we highlighted. As we noticed last time, this DOM node is an `a` element inside a `span` element inside a `div` which has a `class` value of `item_wrap`. This is all useful information for distinguishing the item from the rest of the content on the page. If we scroll down or choose Inspect Element for any other items, we see that this pattern seems to be repeated. Of course, if this pattern were ever to change, all of our code would break.

- Question: as a sidenote, David's investigation (specifically his observation of the subdomain for the HUDS menu) led him to Google around and find that FoodPro serves a number of other universities. Unfortunately, it didn't help him much in gaining direct access to a data source.

- After identifying a standardized tree structure, David was as excited as a tick on a fat dog.[1] However, he was soon befuddled to find that the HUDS menu was not valid XHTML. Thankfully, with Tidy available to fix these validation errors, David was able to rejuvenate his hopes of screen-scraping by traversing the DOM. Regular expressions are always a possibility for gathering data, but in this case they would've been quite complicated and tedious.

2.2 Extracting the Data

- So if we want to write a PHP script which will use XPath to traverse the DOM of the HUDS menu page, we'lll probably begin by including a configuration file with constants (since we'll eventually be connecting to a database) and possibly handling some command-line arguments. Although the script will be automated, we may want the option of running it manually with hardcoded inputs in order to debug. To accomplish this, we'll check to see if command-line arguments have been provided. If they have, we'll set the start and end dates to their values using a function named `strtotime` which converts strings in numerous different formats into date objects.[2] If no command-line arguments have been provided, then by default the start date will be the current date and the end date will be the end of the week:

  ```
  $sd = (@$argv[1]) ? getdate(strtotime($argv[1])) : getdate();
  $ed = (@$argv[2]) ? sd : getdate(strtotime("+6 days", $sd[1]);
  ```

  The `? :` syntax is a ternary operator which is shorthand for if-then-else.

- Now we need to loop over the seven days:

[1] Yeah, I'm from the South, how'd you know?
[2] This will be useful because Harvard dining halls serve brunch on Sundays, so we need to be able to check if the current day is Sunday.

```
for ($date = $sd; $date[0] <= $ed[0];
        $date = getdate(strtotime("+1 day", $date[0])))
{
 ...
}
```

Notice we're using the `getdate` function to increment one day at a time so we don't have to worry about months and leap years and such.

- Within the loop, our first step is to convert today's date into two different formats, one to pass to the HUDS page as a GET parameter and one to store in our database:

```
// get today's date in M-D-YYYY format
$njY= date("n-j-Y", $date[0]);

// get today's date in YYYY-MM-DD format
$Ymd = date("Y-m-d", $date[0]);
```

- In an array, we'll store the different types of meals, accounting for Sundays when only brunch and dinner are served:

```
// determine meals
$meals = ($date["wday"] == 0) ? array("Brunch", "Dinner")
                              : array("Breakfast", "Lunch", "Dinner");
```

We'll then loop through the meals using a `foreach` statement.

- Our first order of business within the loop is to convert the menu to valid XHTML:

```
// fetch meal's menu
if (!($tidy = tidy_parse_file("http://www.foodpro.huds.harvard.edu/foodpro/" .
                    "menu_items.asp?date={$njY}&type=30&meal={$i}",
                    array("numeric-entities" => true, "output-xhtml" => true))))
    continue;

// convert menu to XHTML
$tidy->cleanRepair();
$xhtml = (string) $tidy;
```

We're passing the menu page to Tidy along with an array which we use to specify a variable number of configuration options, one of which tells Tidy to output XHTML. We'll save this XHTML in a variable named `$xhtml` after explicitly casting it to a string (since Tidy actually returns an object). The `numeric_entities` specifies that we don't use HTML entities (since they're not valid in pure XML) but rather their numeric equivalents.

- Question: Tidy does not come pre-installed on most Linux systems, but it's generally fairly easy to install. You can use an installer utility like `yum` or `apt-get`. In the case of `cs75.net`, we have our hands tied a little bit by DirectAdmin: when we need to install a new feature, we have to recompile PHP from source (which is a pain).

- As we begin the process of actually traversing the DOM, we need to step back for a moment to discuss XML namespaces. This whole semester, we've been taking for granted the `xmlns` attribute for the `html` element:

  ```
  <html xmlns="http://www.w3.org/1999/xhtml">
  ```

  `xmlns` stands for XML namespace and is specified by a URL. In programming, a namespace generally means a set of variables which are defined only for a specific context. This ensures that if there exists a separate namespace which has identically named variables, the two namespaces won't collide with each other. Here we're specifying the XHTML namespace. That means we can go on to use namespaces that have elements with names like `title` and not worry about collisions.[3] The URL value actually does nothing except to specify uniqueness. We can come up with our own namespace and specify our own domain name here since presumably no one else in the world will have that domain. We wouldn't use the `xmlns` attribute, however, but rather an attribute like `xmlns:malan`, where `malan` is then the prefix we'll have to use for all of our tags.

  So after all this talk of namespaces, we finally get to another line of code:

  ```
  // parse XHTML
  $dom = simplexml_load_string($xhtml);

  // register XHTML namespace
  $dom->registerXPathNamespace("xhtml", "http://www.w3.org/1999/xhtml");
  ```

  As we've seen before, we're loading up the XHTML string into a DOM with the SimpleXML package. Within that package, we're registering a namespace which we're going specify as `xhtml`. We could've put anything as this first argument, so long as we then used it in our XPath queries consistently, but just for clarity we'll call it `xhtml`.

- In our XPath query, we'll use `xhtml` as a prefix:

  ```
  // get menu's TRs
  $trs = $dom->xpath("//xhtml:form[@id='report_form']/xhtml:table/xhtml:tr");
  ```

  Since all the items are nested within rows of an HTML table, we're going to query for the `tr` elements first. We could've specified every piece of the

[3] So long as we use the correct syntax for prefixing tags.

hierarchy starting with `html`, but we've chosen instead to search all of the DOM nodes by beginning our query with `//`. The form's `id` value and the rest of this query were discovered just by examining the source code of the HUDS menu using Firebug. Just trial and error.

- The next few lines of code, which handle the uneven division of items into categories, we're going to skip over. It's not very interesting to examine.
- Turns out that using XPath to query for the `tr` elements was a good approach because each `tr` element encapsulates only one menu item even though it's nested within several other elements. So to retrieve that menu item, we write:

```
// get item
$a = $tr->td->div->span->a;
if (!($item = trim($a)))
    continue;
```

Our quick sanity check handles the few cases where we have blank rows in our table. Once we have our `a` element, we can grab its `href` attribute and, more specifically, a parameter within it named `recipe` which we surmise is a unique identifier for each menu item:

```
// determine recipe
if (!preg_match("/recipe=(\d+)/", $a["href"], $matches))
    continue;
$recipe = $matches[1];
```

This regular expression searches for `recipe=` and then uses parentheses to capture one or more digits within a variable named `$matches`. Previously, we were using regular expressions to match, which would return a boolean, but here we're using regular expressions to extract, which returns the matching string. `$matches` by default is an array, but the 1-index of that array will gives us the first match.

- Because we have surmised that the `recipe` parameter is a unique identifier for menu items, we're going to build a database table of menu items using it as a primary key. So once we've extracted a menu item, we'll insert it into that table:

```
// INSERT INTO into items
$sql = sprintf("INSERT IGNORE INTO items (recipe, item) VALUES('%s', '%s')",
               mysql_real_escape_string($recipe),
               mysql_real_escape_string($item));
mysql_query($sql);
```

We're escaping just in case HUDS's data has some dangerous SQL keywords in there. The `IGNORE` keyword tells MySQL to fail silently (i.e. without any warnings) when trying to insert an item with a duplicate `recipe` number. We could've used a `SELECT` query to find out if the item already existed, but this takes care of everything in a single query, which is ideal.

- Let's talk database design. What data type should our `recipe` column be? We could make it an `INT`, but what if there are leading zeroes? You might've run into this problem with Project 3—leading zeroes will be truncated for `INT` columns. So instead, we'll probably opt for a `CHAR(6)` data type. We're taking a bit of a chance in assuming that it will never be longer than 6 characters, but we could hedge our bets and use a `VARCHAR` if we really wanted to. As we said before, we're also going to make the assumption that `recipe` is a unique identifier, so we can specify it as a primary key so as to constrain our data and optimize searches.

- Next to each menu item, you may have noticed two small GIFs. These identify the items as part of a subset like "Organic" or "Vegan." If we use Inspect Element on these GIFs, we see that they are `img` elements which are slightly farther up the hierarchy. So we can use XPath to step up a few levels and grab these too:

```
// INSERT INTO legend
$a->registerXPathNamespace("xhtml", "http://www.w3.org/1999/xhtml");
foreach ($a->xpath("../../xhtml:img") as $img)
{
    $sql = sprintf("INSERT IGNORE INTO legend (recipe, `key`) VALUES('%s', '%s')",
                   mysql_real_escape_string($recipe),
                   mysql_real_escape_string($img["alt"]));
    mysql_query($sql);
}
```

  What we're actually grabbing is the `alt` text, which is the human-readable subset name like "Organic" or "Vegan."

- Why have we factored out the `legend` table? In our database design, we want to avoid repeating data as much as possible. Given that some items have two subset names associated with them, if we placed the subset name next to the item in our `menu` table, we would have to repeat the entire row twice. If we have a separate table, we are still repeating a row, but each row only contains two columns. Our `menu` table contains many more columns that we would have to repeat.

  For the same reason, we don't keep the item name as a column in `menu` because it corresponds one-to-one to the `recipe` column, which is already unique.

Another solution would have been to add several more columns as flags for each category. But then each time a category is added, we have to add another column to our table. At some point, too many columns will make a table unwieldy.

One assumption of the `legend` table is that a given `recipe` will not be removed from a given category. This is reasonable, though, as it seems unlikely that meat will be added to a Vegetarian dish without the "recipe" being changed.

- Why have we **not** factored out the `category`? David hypothesized that some items will have different `category` values at different times. Although he could've normalized the database even further, he wanted to avoid over-engineering the data to the point where he would risk breaking his model at some point based on a faulty assumption. Not to mention, this approach is more human-readable. In the case of the course website, for example, the current version factors out students' names and usernames and replaces them with an `INT`. When examining the data in phpMyAdmin, this becomes a huge pain since it's unclear which `INT` corresponds to which student. In the next version of the site, username will be the unique identifier.

- Finally, we insert into our `menu` table:

```
// INSERT INTO menu
$sql = sprintf("INSERT INTO menu (date, meal, category, recipe)"
    "VALUES('%s', '%s', '%s', '%s')",
                mysql_real_escape_string($Ymd),
                mysql_real_escape_string($meals[$i]),
                mysql_real_escape_string($category),
                mysql_real_escape_string($recipe));
mysql_query($sql);
```

At the end of our loop, we'll make a call to `sleep` so that we avoid pounding HUDS's server. Twenty-one queries in the span of a a few seconds, though, really isn't that bad.

- Question: to avoid duplicates, `recipe` and `key` are together defined as `UNIQUE` on the `legend` table.

2.3 Using the Data

- Now, if we follow the guidelines of the API, we can access the data in a more machine-friendly way. For example, if we request the data in JSON format, we might get something like the below:

```
[
   {
```

```
        "date":"2009-11-11",
        "meal":"Breakfast",
        "category":"BREAKFAST BAKERY",
        "recipe":"213012",
        "item":"Aesops Bagels",
        "Vegetarian":true,
        "Vegan":false,
        "Mollie Katzen":false,
        "Local":false,
        "Organic":false
    },
    {
        "date":"2009-11-11",
        "meal":"Breakfast",
        "category":"BREAKFAST BAKERY",
        "recipe":"213046",
        "item":"Pistachio Muffin",
        "Vegetarian":true,
        "Vegan":false,
        "Mollie Katzen":false,
        "Local":false,
        "Organic":false
    }
]
```


We can get the data in CSV, JSON, or PHP format simply by requesting a URL. CSV and JSON are familiar to you by now, but the PHP format might be new. What we do is *serialize* the data, meaning we represent it as a string even though it's actually a hierarchical object or array. Then we can send it over HTTP to someone who will then *unserialize* it to use it in its original hierarchical form.

As a sidenote, if an object is passed to PHP's `serialize` function, its methods are not by default encoded. That prevents someone from including a malicious script which a user might blindly execute by calling `unserialize`. In this case, it doesn't matter, since we're simply sending data across the wire.

3 Shuttleboy: Adding Ajax (61:00–95:00)

- We talked last time about the Shuttleboy application and its one major shortcoming: it used a `meta` tag to add new content via a page refresh every minute or less. This was a waste of resources because a lot of the content on the page is static, yet it would have to be reloaded every single time a page refresh was performed.[4]

[4] Not accounting for browser caching, anyway.

- One caveat was that David wanted users to be able to bookmark routes for easy access. So data requests are done via GET rather than POST. But more importantly, he didn't want **all** of the data to be retrieved via Ajax because this would mean the URL would never change. When searching on Google Maps, for example, the URL never changes from `maps.google.com`. You can, however, click the Link button on the upper righthand corner so as to be provided with a so-called "deep link" that saves the page state.

  What are the downsides of Google's approach? First, long URLs are unwieldy and may be broken when copying and pasting in an e-mail for example. Second, the average user may not realize that he needs to click on the Link button in order to get a URL he can bookmark. He might instead click the bookmark shortcut in his browser and thus save `maps.google.com` instead of his search.

- So to satisfy both desires—to use Ajax but to create URLs that can be bookmarked—we resort to a trick that's been around for a while. A *fragment identifier* is a parameter in a URL which is separated from the others by a `#`. If you have an element in a page `foo.php` with the `name` or `id` attribute of `bar`, then a link to `foo.php#bar` will jump you straight to that element in the page.

  In the context of Ajax, however, we don't care about this ability to jump to different content. What we care about is the fact that we can append fragment identifiers to the URL to provide extra information but without fundamentally altering the URL. And using external libraries like YUI or jQuery, we can use this extra information to maintain the page's state. Frankly, it's a pain to implement across different browsers without an external library, so perhaps it's not surprising that Google Maps has chosen not to do this yet.

  The YUI Browser History Manager listens for changes in the URL every half second or so. When a change occurs, it parses the URL to grab everything after the `#` and treat it as a GET string. It then updates the page content accordingly. One of the big problems with this approach is that the browser's back button will usually break.

- If we open up Firebug and click the Console tab while examining a schedule on Shuttleboy,[5] we see that a request is being made to `search.php` every five seconds or so. Previously, we were refreshing every 60 seconds, but David found that when he made an Ajax request only every minute, occasionally the client clock and the server clock would get out of sync and requests would be missed. Refreshing every five seconds is acceptable because eventually Google Maps will be integrated into the site to show shuttle locations, so there will soon be a need to refresh that frequently anyway.

[5] You need to choose two stops for any Ajax gets involved to refresh the schedule.

- For each of these requests, we can inspect the response headers (by clicking the Response tab) to see that the Content-type of the response is actually text/html. As empirical tests have shown, preparing XHTML server-side and inserting it all in one chunk on the client-side is generally more efficient than the standards-compliant approach of modifying and adding nodes in JavaScript.

  If we inspect the parameters that are sent with each request (by clicking the Params tab), we see three: `a` and `b`, which designate stops, and `output`, which designates the type of content to respond with. We can also inspect the actual response by clicking on the Response tab and we see that it is an XHTML table. This table is assigned as the value of the `innerHTML` of the righthand `div`.

- The major downside of adding Ajax to Shuttleboy is that our servers will experience a great deal more load per user. Each user will now be making a request to our servers every five seconds. We could, of course, limit the amount of content that each of these requests generates—not downloading the entire table each time, for example—but no matter what, we'll probably have a higher load on our servers.

  We learned this firsthand when we developed an office hours sign-up tool using Ajax which crippled our servers because the refresh rate was too high. We had to go back to a whiteboard eventually.

  To view this load, we can run the `top` Linux utility to see all the current running processes. If a number of different `httpd` processes are eating up our CPU, we have a good idea that too many requests are being funneled through too few processes, causing them to choke and back up. You should be so lucky as to have your website slashdotted someday.

- Question: to better handle the increased load and perhaps reject some users who might push your servers over the edge, you could throttle network traffic at the router level or balance it across multiple servers with a load balancer.

- Question: each of the `httpd` processes is actually a thread that forked off the original `httpd` process at startup. The maximum number of threads that are forked off is an Apache configuration option.

- Question: unfortunately, it's not possible in JavaScript to determine if a user has minimized his window. And you certainly can't tell if he's not paying attention, so it wouldn't be possible to stop the Ajax updating when the user was idle.

- Question: sure, updating the page's content automatically for a user might run contrary to the way the web used to work, but frankly the generation which is up-and-coming is already accustomed to sites like Facebook and Twitter doing so. Some users might forcibly refresh pages, but that's fine—the site's still going to work like it's supposed to.