MASHUPS: Remixing the Web

H79.2802

ASSIGNMENT #4: BRINGING IT ALL TOGETHER

Due November 17, 2010 (in lecture)

Reflection Ideation Exercise Bonus Challenge

Collective Intelligence vs. the Cult of the Amateur (*10 Points*)

Two of the course readings discuss websites that attempt to harness the "wisdom of crowds." Jeff Howe's WIRED piece, "The Rise of Crowdsourcing," is largely a collection of success stories demonstrating how the power of collective intelligence can be applied to problems in many different domains.

But the wisdom of the crowds can just as easily become the madness of mobs or the tragedy of the commons. In "Infotopia: How Many Minds Produce Knowledge," Cass Sunstein gives a more nuanced portrayal of crowdsourcing, discussing its limitations as well as its strengths.

For all its flaws, Wikipedia is a wonderful thing, but for every Wikipedia-like success story there is a site that has failed to successfully take advantage of user-generated content. We run into the negative aspects of crowds every day, from bad behavior in crowded subway cars to poor decisions in financial markets. Furthermore, some argue that innovation is more likely to come from individuals than from groups, and the democratization of content contribution can lead to a situation in which knowledgeable experts are shouted down by a less capable majority, a situation sometimes described as "the cult of the amateur."

What do you think are the primary factors leading to the success or failure of a crowd-sourced website? In your opinion, how do successful user-contributed websites like Wikipedia provide the right system of structure, controls, and incentives that encourage participation, keep communication open, and result in generally high-quality content? How can we open websites up to user submissions in a way that raises us up to the sum of our capabilities instead of dooming us to the mediocrity of the masses?

The Future of Mashup Development (*10 Points*)

In “Hacking, Mashing, Gluing: Understanding Opportunistic Design,” Hartmann et al. describe a development methodology they call *opportunistic programming*. Based on their interviews with web programmers, the authors conclude that this model of development is common among mashup creators, and that it is not particularly well served by traditional design tools and software development frameworks.

You have spent the semester becoming a Web 2.0 programmer, acquainting yourself with a new set of tools and APIs and learning to build your own mashups. What aspects of mashup development did you find the most challenging? What did you find to be the major stumbling blocks? What do you see as the biggest limitations of the existing set of tools used to build Web 2.0 websites? Do you see yourself as an opportunistic programmer, and if so, do you agree with the assessment that existing development tools do not adequately support your typical programming habits?

In the course readings and lectures, you have seen examples of various systems designed to address the needs of mashup creators: Greasemonkey, which allows website customization through the injection of arbitrary JavaScript code; Marmite, which lets users extract content from web pages and process it in a dataflow manner; Chickenfoot, which supports the automation of web tasks using simple natural language commands; and d.mix, which leverages site-to-service correspondences to extract sample code from sites using an API. Which of these tools best reflect your own approach to building mashups? Can you think of a different tool that might simplify your own mashup-building process?

txting 2 sms gtwy (*25 Points*)

In this exercise, you will write a JavaScript-powered web page that connects to an SMS gateway and displays the SMS text messages received by the gateway. An SMS gateway is a server that receives the text messages sent to a particular phone number and makes them available elsewhere.

1. Let’s begin by learning how to read messages from the Gateway server. You can access the latest messages sent to the Gateway Server using the following URL:

   http://webremix.org/assignments/sms/messages.php

   Verify that the Gateway is working by using your cell phone to send a text message to the following phone number: **650-451-1228**. Wait for 5-10 seconds for the message to be received and processed by the gateway, and then reload the page. You should see your message appear in the output.

2. The gateway script outputs messages in a simple XML format. By default, the script will output only the messages received in the last five minutes. If no messages were received in the last five minutes, the output will be empty. Here is an example of what the XML output of the script looks like:

   ```
   <messageset>
   <message timestamp="1246555031" number="5552293067">Living in space</message>
   <message timestamp="1246555033" number="5553952178">Still paying my mortgage</message>
   <message timestamp="1246555035" number="5555454956">Eating beef jerky</message>
   </messageset>
   ```

   Notice that the entire set of messages is enclosed in a `<messageset>` element, and each text message is enclosed in a `<message>` element. The `<message>` elements have two attributes: a `timestamp` specifying the time that the message was received by the gateway, and the phone `number` from which the message originated. The timestamp is a typical Unix timestamp, which represents the time as the number of seconds since January 1, 1970.

   In addition to the default behavior of the script, which queries for messages sent in the last five minutes, you can query for messages in a particular time range by specifying `starttime` and `endtime` URL parameters. The `starttime` parameter specifies the earliest message timestamp you would like to see, and the `endtime` parameter specifies the latest timestamp. Here is a URL that requests messages stamped between the timestamps 1288161134 and 1288161160:

   http://webremix.org/assignments/sms/messages.php?starttime=1288161134&endtime=1288161160

   If you don't specify a `starttime`, the script will default to a starting time of five minutes ago. If you don't specify an `endtime`, the script will assume the current time (all messages up to the present). So the URL
   http://webremix.org/assignments/sms/messages.php?starttime=0
   would give you all of the messages ever collected by the gateway. (It's probably not a good idea to make queries like this once the number of messages stored on the gateway becomes large.)

   For testing purposes, the gateway server has been populated with several hundred sample messages, spaced in quick succession, in the timestamp range 1288161134-1288161532. Try making a few queries in this range until you understand how to make requests to the gateway.

3. Before you can write AJAX code to read the incoming messages, you will need to set up a PHP *proxy* to query the gateway. Recall that browser security restrictions will prevent you from making AJAX queries across domains. Since you will not be hosting your JavaScript code on the same domain as the SMS gateway, you cannot query the gateway

directly from JavaScript. Instead, you will need to create a simple PHP script known as a *proxy* that receives requests from your JavaScript code, forwards the requests with their parameters to the gateway server, and returns the responses to your JavaScript code. Here is a working example of a simple proxy that you can use for this assignment:

```
<?php
// A simple PHP proxy to query the SMS Gateway Server
header('Content-Type: text/xml');
$url = "http://webremix.org/assignments/sms/messages.php?";
if (isset($_GET["starttime"])) {
  $url .= "starttime=" . $_GET["starttime"];
}
if (isset($_GET["endtime"])) {
  $url .= "&endtime=" . $_GET["endtime"];
}
echo file_get_contents($url);
?>
```

You can also find this proxy example code here:
http://webremix.org/assignments/simple-proxy.txt

Copy the proxy code to the server space that you will be using for this assignment. Try querying the proxy the same way that you queried the gateway directly, using `starttime` and `endtime` parameters, and verify that the output is the same.

4. Your goal is to create a nicely animated JavaScript visualization of the latest messages to be received by the gateway. When people visit your page, they should be able to send a text message to the gateway phone number, and see it animate onto the page after a short delay.

   To accomplish this, you will need to write JavaScript code that:
   a. Makes an AJAX call to retrieve the latest messages from the gateway, by way of your proxy script.
   b. Only retrieves the latest messages, to avoid displaying the same message more than once. You can accomplish this by specifying new non-overlapping time intervals each time you make a query to the gateway.
   c. Displays the newest messages in a compelling animated fashion, using fades, color, motion, or some combination thereof.
   d. Makes older messages animate out, fade away, or transition into new messages after they have been onscreen for a while.

   When testing your script, you probably won't want to constantly use your phone to send new SMS messages. Instead, you can simulate new messages being received by starting your script at an older timestamp and moving the time forward from there. When the script starts, pretend that the current time is 1288161134, and store this in a variable called `currentTime`. Declare another variable called `lastUpdateTime` that stores the last time that your script checked for new message. Every three seconds, add three to the `currentTime` variable, and make a request to retrieve the messages received by the gateway between `lastUpdateTime` and `currentTime`:

http://webremix.org/assignments/sms/messages.php?starttime=[`lastUpdateTime`]&endtime=[`currentTime`]

Finally, after making this request, update the time at which the last update was received by setting `lastUpdateTime` equal to `currentTime`.

SMS Polling (*10 Points*)

The SMS gateway server also supports simple *polling*, to collect answers to multiple choice questions from large groups of people via SMS messages.

1. To vote in a poll, you simply send a text message to the gateway phone number that contains the number of the question followed by the letter of your response, which must be A, B, C, or D. For example, to vote for choice "B" on question 2, you would text "2B" to **650-451-1228**. To prevent ballot stuffing, you can only vote once on a particular question from a particular phone number; after that your vote will be ignored.

   To access the vote counts, make an HTTP request like this:

   http://webremix.org/assignments/sms/poll.php?question=2

   You can replace the `question=2` parameter with a different question number to retrieve the counts for a different question. The server will return an XML-formatted response that looks like this:

   ```
   <pollresult question="2">
   <votecount choice="a">2</votecount>
   <votecount choice="b">2</votecount>
   <votecount choice="c">0</votecount>
   <votecount choice="d">0</votecount>
   </pollresult>
   ```

   As in the previous exercise, you will need to set up a PHP proxy to access the vote counts from the gateway. You can modify the proxy script from the previous exercise to pass along the `question` parameter instead of the `starttime` and `endtime` parameters.

2. Once you have the proxy working, your goal is to create a web page that does the following:
    a. Prompts visitors with a series of numbered questions, each with four possible choices labeled A, B, C, and D.
    b. Continually updates the live count of responses for all of the questions.
    c. Displays the poll counts in a continually-updating pie chart or bar chart. You can draw the charts using a chart-drawing tool like Flot (http://code.google.com/p/flot/), PlotKit (http://www.liquidx.net/plotkit/), JSCharts (http://www.jscharts.com/), or the Google Chart API (http://code.google.com/apis/chart/).

Data-Driven Fitness: Plotting Runs on Google Maps (*25 Points*)

This exercise uses the Google Maps API. You can read more about the Maps API in the Maps API Developer's Guide:
http://code.google.com/apis/maps/documentation/index.html

You can also find many useful examples in the Code Playground:
http://code.google.com/apis/ajax/playground/?exp=maps#map_simple

1. You will begin this exercise with a template:
   http://webremix.org/assignments/maps-template.html

   The template downloads an XML file containing the data from a single run, and draws the entire running route on the map as a polyline. You will improve upon this basic template to create a system for viewing data about past runs.

2. First, register for a Google Maps API Key, and add it to the template. You can sign up for an API key here:
   http://code.google.com/apis/maps/signup.html

   Note that the API key is limited to a particular domain, so you will need to decide which site you will be using to host your code before registering for a key. If you attempt to use your key from a page hosted on a different domain, you will see an error message. If you are hosting you code on your ITP web space, you would apply for a key using the URL `http://itp.nyu.edu/~yournetid`.

3. For security reasons, browsers do not allow sites to make asynchronous download requests to an external domain. This domain restriction means that if you are hosting your code on `itp.nyu.edu`, you will not be able to download an XML file from `webremix.org` (or even from `www.nyu.edu`, since it is considered a different top-level domain). Because of this rule, you will need to download the GPX run data files for this exercise and host them locally on your site. You can find all of the data files here:

http://webremix.org/assignments/runs/

Download the GPX files and copy them to the same location in which you are hosting your code for this exercise. Then modify the template to point to the new location of the `atlanta.gpx` data file.

The running data is in GPX format, a common lightweight XML format for the exchange of GPS data. The format is fairly straightforward, but if you would like to learn more about GPX or study the official schema, visit http://www.topografix.com/gpx.asp.

4. If you have completed the preceding steps correctly, you should be able to load the page in your browser and see a red running path traced around Chastain Park in Atlanta.

5. The example always starts with the map in Atlanta, and doesn't move the map when loading data from a different location. Modify the code so that when a new GPX file is loaded, the map center is moved to the center of the run path, and zoomed so that the path just fits inside the window. To do this, you will need to keep track of maximum and minimum latitude and longitude values you observe when stepping through the data.

6. Seeing the whole polyline drawn at once isn't very interesting – it would be more fun to view a run progressing over time. Modify the code to animate the run by tracing the path gradually. As before, you should store the trackpoints in an array when the XML document is loaded. However, instead of drawing the whole polyline at once, draw it in increments, setting a timed delay in between each increment. You can set a timer using `window.setTimeout()`.

7. Finally, add a drop-down menu that allows users to select among multiple running data files. You can create the menu using a `<select>` form element, with one `<option>` for each filename. When a file is selected from the dropdown menu, your code should clear the map, load the new file, jump to the location of the new run, and animate the run.

Advanced Mapping Fun (*15 Points*)

In this bonus exercise, you will add some additional enhancements to the running map you created in the previous exercise.

1. GPS devices don't typically store trackpoints at regular time intervals when recording a path. Instead, devices will save space by drawing trackpoints far apart on straight-aways and closer together on curves. You can determine the exact time that each trackpoint was recorded by looking at the `<time>` field inside the `<trkpt>` element. Change your code so that the time between drawing each trackpoint reflects the actual time elapsed. Naturally, you do not want to playback the run in real-time, or you would have to sit at your browser for hours to watch a long run! Instead, scale the time so that one minute of

running takes one second of playback time.

2. Add mile marker "checkpoints" to the map at one-mile increments along the polyline. They don't have to be exactly a mile apart – they can be at the trackpoint that is closest to the mile mark, or the first trackpoint encountered after the mile mark has been passed.

3. Add a statistics table below the map. Each time your code draws a checkpoint marker, print out the distance traveled, elapsed time, and pace for the previous one-mile segment expressed in minutes per mile.

4. GPS devices can also record elevation. In the data files provided, you can find the elevation measurements in an element called `<ele>` inside each `<trkpt>` element. Using a JavaScript charting library such as Flot (http://code.google.com/p/flot/), PlotKit (http://www.liquidx.net/plotkit/), or JSCharts (http://www.jscharts.com/), draw a graph of the elevation over time that updates in sync with the route as it is being drawn.