CIS 455/555: Internet and Web Systems

Spring 2013

Assignment 2: Web Crawling and XPath

Milestone 1 due March 15, 2013, at 10:00pm
Milestone 2 due March 26, 2013, at 10:00pm

1 Introduction

In this assignment, you will explore several important Web technologies, and continue to build components useful towards your course project, by building a topic-specific Web crawler. A topic-specific crawler looks for documents or data matching a particular category – here, the category will be specified as an XPath expression.

This assignment will entail:

- writing a servlet-based Web application that runs on the application server you built for Assignment 1 and allows users to create topic-specific "channels" defined by a set of XPath expressions, and to display documents that match a channel using a referenced XSL stylesheet;
- writing an XPath evaluation engine that determines if an HTML or XML document matches one of a set of XPath expressions;
- writing a crawler that traverses the Web, looking for HTML and XML documents that match one of the XPath expressions; and
- building a persistent data store, using Oracle Berkeley DB, to hold retrieved HTML/XML documents and channel definitions.

The resulting application will be very versatile; one use of it could be to write an RSS aggregator with keyword-defined channels. In this case, the XPath expressions would find RSS feeds (which are just XML documents) that contain articles with the specified keywords, and the stylesheet would select the matching articles from the feeds and format them for the user. In Milestone 2, you will provide an XSL stylesheet and XPath expressions for a sample query over RSS documents. However, your application will have many other uses as well.

Note that Asssignment 2 builds on your application server from Assignment 1. If your application server does not work, please use Jetty (http://jetty.codehaus.org/jetty/) or Tomcat (http://tomcat.apache.org/), to test your servlet.

2 Developing and running your code

We strongly recommend that you do the following before you start writing code:

1. **Carefully read the entire assignment** (both milestones) from front to back and make a list of the features you need to implement. There are many important details that are easily overlooked!

2. Spend at least some time thinking about the *design* of your solution. What classes will you need? How many threads will there be? What will their interfaces look like? Which data structures need synchronization? And so on.
3. Check in your changes into svn regularly.

We recommend that you continue using the VM image we have provided for HW0 and HW1. This image should already contain all the tools you will need for HW2. You can check out the framework code for HW2 using the same process as for HW0 (there should now be an additonal "HW2" folder).

Of course, you are free to use any other Java IDE you like, or no IDE at all, and you do not have to use any of the scripts we provide. However, to ensure efficient grading, your submission **must** meet the requirements specified below – in particular, it must build and run correctly in the original VM image and have an `ant` build script. If you are using our scripts, this should be the case automatically.

We strongly recommend that you regularly check the discussions on Piazza for clarifications and solutions to common problems.

3 Milestone 1: XPath engine (due March 15)

The first milestone will consist of an XPath evaluator. You will write a simple servlet application, hosted on your application server from Assignment 1, that takes an XPath and a URL to an HTML or XML document. You can implement XPath matching in any (correct!) way of your preference: one option is to follow the model established in XFilter, but one can also use other schemes involving recursive traversal and matching on the XPath.

If the document passes the filter, your servlet should return an HTML message indicating success; otherwise it should return an HTML message indicating that the document failed the test.

3.1 Servlet user interface

Your servlet class should be called `edu.upenn.cis455.servlet.XPathServlet`. When the servlet's root URL is opened, it should present to the Web browser an HTML form (see the HTML forms interface). This form should have (at least) input boxes for the XPath and HTML/XML document URL. When the form is submitted, it should generate a `POST` message that invokes a handler servlet. In turn, this should trigger a parse and a scan for XPath matches. The servlet should then return success or failure.

3.2 XPath Engine

You need to write a class `edu.upenn.cis455.xpathengine.XPathEngineImpl` that implements `edu.upenn.cis455.xpathengine.XPathEngine` (included with the code in `svn`) and evaluates a set of XPath expressions on the specified HTML or XML document. The constructor for `XPathEngineImpl` must not have any arguments. The setXPaths method gives the class a number of XPaths to evaluate. The `isValid(i)` method should return `false` if the i.th XPath was invalid, and `true` otherwise. The `evaluate(d)` method takes a DOM root node as an argument, which contains the representation of a HTML or XML document, and it returns an array with the i.th element set to `true` if the document matches the i.th XPath expression, and `false` otherwise.

To make things simpler, we are supporting a very limited subset of XPath, as specified by the following 'grammar' (modulo white space, which your engine should ignore):

$$
\begin{aligned}
\textit{XPath} &\rightarrow \textit{axis step} \\
\textit{axis} &\rightarrow \texttt{/} \\
\textit{step} &\rightarrow \textit{nodename}\ (\texttt{[}\ \textit{test}\ \texttt{]})^*\ (\textit{axis step})? \\
\textit{test} &\rightarrow \textit{step} \\
&\rightarrow \texttt{text() = "..."} \\
&\rightarrow \texttt{contains(text(), "...")} \\
&\rightarrow \texttt{@attname = "..."}
\end{aligned}
$$

where `nodename` and `attname` are valid XML identifiers, and "..." indicates a quoted string. This means that a query like `/db/record/name[text() = "Alice"]` is valid but the similar (and valid XPath) query `/db/record/child::name[text() = "Bob"]` is not. Recall that if two separate bracketed conditions are imposed at the same step in a query, both must apply for a node to be in the answer set.

Below are some examples of valid XPaths that you need to support (not an exhaustive list):

```
/foo/bar/xyz
/foo/bar[@att="123"]
/xyz/abc[contains(text(),"someSubstring")]
/a/b/c[text()="theEntireText"]
/blah[anotherElement]
/this/that[something/else]
/d/e/f[foo[text()="something"]][bar]
/a/b/c[text() =   "whiteSpacesShouldNotMatter"]
```

You will want to think of the XPath match as a recursive process, where an XPath is matched if its step node-test matches and all of its tests (recursively) match and its next step matches. The easiest HTML/XML parser to use in Java is probably a DOM (Document Object Model) parser. Such a parser builds an in-memory data structure holding an entire HTML or XML document. From there, it is relatively easy to evaluate an XPath expression recursively. Sample code using a DOM parser is available in the `examples` directory in `svn`. You do not need to use a SAX parser as with XFilter, but you can do so for extra credit (see Section 6.2).

3.3 Unit Tests

In addition to the basic code specified above, you must implement at least 5 *unit tests*, using the JUnit 3 package (see the section on Testing below for helpful Web page references). JUnit provides automated test scaffolding for your code: you can set up a set of basic objects for all of the test cases, then have the test cases run one-by-one.

Your JUnit test suite should instantiate any necessary objects with some test data (e.g., parse a given HTML or XML document or build a DOM tree), then run a series of unit tests that validate your servlet and your XPath matcher. In total you must have at least 5 unit tests (perhaps each designed to exercise some particular functionality) and at least one must be for the servlet and one for the XPath evaluator.

Note that you should use JUnit 3 for this assignment, not any other version.

3.4 Requirements

Your solution must meet the following requirements (please read carefully!):

1. Your servlet class must be called `edu.upenn.cis455.servlet.XPathServlet`; your XPath engine class must be `edu.upenn.cis455.xpathengine.XPathEngineImpl`. Please check the capitalization – XPath is not the same as Xpath! The XPath engine class must implement the `edu.upenn.cis455.xpathengine.XPathEngine` interface.
2. Your submission must contain a) the entire source code, as well as any supplementary files needed to build your solution, b) a working `ant` build script called `build.xml` (such as the one included in the `svn` repository), and c) a `README` file. The `README` file must contain 1) your full name and SEAS login name, 2) a description of features implemented, 3) any extra credit claimed, 4) a list of source files included, and 5) any special instructions for building or running. You must also complete all the yes/no questions in the file.
3. When your submission is unpacked in the original VM image and the `ant` build script is run, your solution must compile correctly. Please test this before submitting!
4. Your server must implement the user interface specified in 3.1, and the interface must display your full name and SEAS login name along with the web form.
5. You must implement your own HTTP client (open socket, send headers, parse response, etc.).
6. Your solutions for MS1 and MS2 must be submitted via `turnin` (!) before the deadlines stated on the first page, unless you have used jokers to extend these deadlines. The project name for `turnin` should be `hw2ms1` for the first milesone, and `hw2ms2` for the second milestone.
7. Your code must contain a reasonable amount of useful documentation.

Reminder: All the code you submit (other than the Mozilla/JTidy/TagSoup parser, the standard Java libraries, and any code we have provided) must have been written by you personally, and you may not collaborate with anyone else on this assignment. Copying code from the web is considered plagiarism.

4 Milestone 2: Storage and crawler (due March 26)

For the second milestone, you will add (1) a Web crawler that will look for relevant XML documents, and (2) a storage capability for HTML and XML documents that the crawler has retrieved.

4.1 Store

You will use Berkeley DB Java Edition (http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html), which may be downloaded freely from their website, to implement a disk-backed data store. Berkeley DB is a popular embedded database, and it is relatively easy to use; there is ample documentation and reference information available on its Web site, e.g., the Getting Started Guide at http://docs.oracle.com/cd/E17277_02/html/GettingStartedGuide/BerkeleyDB-JE-GSG.pdf.

Your store will hold (at least) the usernames and passwords of registered users, the XPath expressions and URLs for XSL stylesheets for the user-defined channels and the name of the user that created them, and the raw content of HTML or XML files retrieved from the Web ~~that match one of the user-supplied XPath expressions~~ as well as the time the file was last checked by the crawler. We expect that you will implement a easy-to-use wrapper class around the store which will provide all of the operations you need to perform on the database as methods.

The path to your Berkeley DB data store will be specified as the `BDBstore` context parameter in your `web.xml` file.

4.2 Crawler

Your web crawler will be a Java application that can be run from the command line, as `edu.upenn.cis455.crawler.XPathCrawler`. It will take the following command-line arguments (in this specific order, and the first three are required):

1. The URL of the Web page at which to start.
2. The directory containing the BerkeleyDB database environment that holds your store. The directory should be created if it does not already exist. Your crawler should recursively follow links from the page it starts on. (Note: the store servlet takes the path from `web.xml` while the crawler expects it as a command-line argument. You may assume that these paths are the same.)
3. The maximum size, in megabytes, of a document to be retrieved from a Web server
4. An optional argument indicating the number of files (HTML and XML) to retrieve before stopping. This will be useful for testing!

It is intended that the crawler be run periodically, either by hand or from an automated system like `cron` command. So there is therefore no need to build a connection from the Web interface to the crawler.

The crawler traverses links in HTML documents. You can extract these using a HTML parser, such as the Mozilla parser (http://mozillaparser.sourceforge.net/), TagSoup (http://ccil.org/~cowan/XML/tagsoup/), JTidy (http://jtidy.sourceforge.net/) or simply by searching the HTML document for occurrences of the pattern `href="URL"` and its subtle variations. If a link points to another HTML document, it should be retrieved and scanned for links as well. If it points to an XML or RSS document, it should be retrieved as well. Don't bother crawling images, or trying to extract links from XML files. All retrieved HTML and XML documents should be stored in the database (so that the crawler does not have to retrieve them again if they do not change before the next crawl), but only the XML documents that match one of the XPath expressions should be added to the corresponding channels. The crawler must be careful not to search the same page multiple times during a given crawl, and it should exit when it has no more pages to crawl.

When your crawler is processing a new HTML or XML page, it should print a short status report to `System.out`. Example: "http://xyz.com/index.html: Downloading" (if the page is actually downloaded) or "http://abc.com/def.html: Not modified" (if the page is not downloaded because it has not changed).

4.3 Politeness

Your crawler must be a considerate Web citizen. First, it must respect the `robots.txt` file, as described in A Standard for Robot Exclusion (http://www.robotstxt.org/wc/norobots.html). It must support the Crawl-Delay directive and "User-agent: *", but it need not support wildcards in the Disallow: paths. Second, it must always send a HEAD request first to determine the type and size of a file on a Web server. If the file has the type `text/html` or one of the XML MIME types

- `text/xml`
- `application/xml`
- Any mime types that ends with `+xml`

and if the file is at most as large as the specified maximum size, then the crawler should retrieve the file and process it; otherwise it should ignore it and move on. For more details on XML media types, see RFC 3023 (http://www.ietf.org/rfc/rfc3023.txt). Your crawler should also not retrieve the file if it has not been modified since the last time it was crawled, but it should still process unchanged files (i.e., match them against XPaths and extract links from them) using the copy in its local database.

We expect you to implement your own HTTP client, i.e., you need to write code for opening a TCP connection, sending the request, and parsing the response. You may not use the HTTP client that comes as part of the Java standard library, or any other code that you did not personally write.
Certain web content, such as the papers in ACM's Digital Library, normally costs money to download but is free from Penn's campus network. If your crawler accidentally downloads a lot of this content, this will cause a lot of trouble. To prevent this, you **must** send the header `User-Agent: cis455crawler` in each request. Since this is very important, solutions that omit or misspell this header will receive a zero.

4.4 Servlet-based Web Interface

For Milestone 2, you will also enhance the servlet to support the following functions:

- Create a new account
- Log into an existing account
- Log out of the account that is currently logged in
- List all channels available on the system (does not require login)
- Create a new channel by specifying its name, the set of XPath expressions, and the URL of the XSL stylesheet (requires login)
- Delete a channel created by the logged in user
- Display a channel

How you implement most of the functionality of theWeb interface is entirely up to you. However, in order to make things consistent across assignments, we are specifying how the channel must be displayed and the schema that the XSL stylesheet should expect. The channel must be displayed by sending an XML document to the client with an embedded link to the XSL stylesheet for the channel. The XML document should follow the following structure.

1. It should have as its outermost tag a `<documentcollection>` tag.
2. For each XML document that matched one of the XPaths, the `<documentcollection>` tag should contain a `<document>` tag that has the following attributes:
    - `crawled`, defined to be the time the XML document was last visited by the crawler, in the same format as `2002-10-31T17:45:48`, i.e. `YYYY-MM-DDThh:mm:ss`, where the `T` is a separator between the day and the time.
    - `location`, defined to be the URL of the document

    and has as its content the verbatim matching document, except for the processing instruction (`<?xml ... ?>`). Any XSL stylesheet should therefore work relative to this schema.

We expect this application to run on your application server from the first assignment. If you did not complete the first assignment, or for some other reason do not want to continue to use the application server that you wrote, you may Jetty (http://www.eclipse.org/jetty/) or Tomcat (http://tomcat.apache.org/) to test your servlet. There will not be any penalties for using Jetty or Tomcat.

4.5 RSS 2.0 aggregator

You are to supply XPath rules to find RSS 2.0 documents which contain items whose title or description contains the character strings `war` or `peace`. This should be in a file called `warandpeace.xp`. You are also to supply a XSL stylesheet `rss.xsl` that displays this channel by showing, for each matching RSS document, its title (as a link to the website specified by the feed), and for each of its matching items, the title and description (each if they exist) and providing the link (if it exists).

4.6 Test cases

You must develop at least two JUnit tests for storage system and two more the crawler. This is in addition to the five XPath unit tests described in Section 3.3 (which are due with MS1).

4.7 Requirements

Your solution must meet the same requirements as MS1 (see 3.4 above), with two exceptions: You should also, in the `rss/` subdirectory, include the `warandpeace.xp` file that contains your XPath expressions for the RSS aggregator, as well as the `rss.xsl` XSL stylesheet. Be sure to note the relative path of your HTML form page in the `README` file, so that we can test it with a browser.

5 Testing

5.1 'Sandbox'

We have implemented a small sandbox for you to test your code on. It runs on machines in the Engineering Department, so it will be fast to access, and it will not contain any links out of itself. The address of the sandbox is `http://crawltest.cis.upenn.edu/`.

5.2 JUnit

In order to encourage modularization and test driven development, you will be required to code test cases using the JUnit package (http://www.junit.org/) – a framework that allows you to write and run tests over your modules. A single test case consists of a class of methods, each of which (usually) tests one of your source classes and its methods. A test suite is a class that allows you to run all your test cases as a single program.You can get more information here: http://www.onjava.com/pub/a/onjava/2004/02/04/juie.html.

For Milestone 1, you must include 5 test cases and for Milestone 2, a test suite consisting of these 5 and at least 2 more for each new component. If your test suite uses any files (e.g., test inputs), please put them into your project folder and use a relative path, so your tests will run correctly on the graders' machines.

6 Extra credit

There are several enhancements you can add to your assignment for extra credit. In all cases, if you implement an improved component, you do not need to implement the simpler version described above; however, your improved component must still pass our test suite for the basic version, and you will lose points if it does not. A safer approach is to implement the basic version first and to extend it later; if you choose to do this, and if you submit both versions, please document in your `README` file how to enable the extra functionality.

6.1 Advanced crawler design (+5%/+15%)

You can implement a multi-threaded crawler for 5% extra credit. You can implement a crawler based on the Mercator design, namely supporting their innovations in the 'URL Frontier' and 'Content-Seen Test' sections, for 15% extra credit.

6.2 DFA-based XPath engine (+20%)

You can implement a DFA-based XPath engine for 20% extra credit; see the XFilter paper for a starting point. This will entail using a SAX parser; sample SAX parser code is also available on the course webpage. Please ask on Piazza about the appropriate method signature for `evaluate` in this case.

6.3 Channel subscriptions (+5%)

For 5% extra credit, allow users to choose which channels to subscribe to from a list of available channels. Show them a list of the channels they're subscribed instead of all channels, and indicate which channels may have updated content since they last viewed it (i.e., for which channels has a matching XML document been updated, or has a new XML document matched that didn't match before).

6.4 Crawler web interface (+15%)

For 15% extra credit, provide a Web interface for the crawler. An admin user (not all users) should be able to start the crawler at a specific page, set crawler parameters, stop the crawler if it is running, and display statistics about the crawler's execution, such as

- the number of HTML pages scanned for links,
- the number of XML documents retrieved,
- the amount of data downloaded,
- the number of servers visited,
- the number of XML documents that match each channel, and
- the servers with the most XML documents that match one of the channels.

This will entail using some sort of interprocess communication, so it's not for the faint of heart.