Spring, 2010 CIS 455 / 555

Internet and Web Systems

Assignment 2

Milestone 1 Due March 1, 2010; Milestone 2 Due March 15, 2010

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 as Assignment 1, which 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 XML document matches one of a set of XPath expressions.
- Writing a crawler that traverses the Web, looking for XML documents that match one of the XPath expressions.
- Building a persistent data store, using Oracle Berkeley DB, to hold retrieved 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.

1 Milestone 1: Due March 1

As with Assignment 1, you are required to implement your assignment in Java. You will submit an archive containing the complete source code to your program, a README file containing any assumptions you made, extra features you added, or anything else you want to tell us.

The first milestone will consist of an XPath evaluator. You will write a simple servlet application, hosted on your application server from Assignment 1[1], that takes an XPath and a URL to an

[1] If your application server does not work, please use Jetty, www.mortbay.org, to test your servlet.

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, you will return an HTML message indicating success; otherwise you will return an HTML message indicating that the XML document failed the test.

1.1 Servlet User Interface

Your servlet should be called **edu.upenn.cis555.XPathServlet**. 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 XML document URL. It should return 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.

1.2 XPath Engine

You are to write a class **ed.upenn.cis555.XPathEngine** that evaluates a simple XPath expression on the specified XML document. The constructor for this class should take the XPath expression as a string; the class will have a method called `evaluate`, which returns a `boolean` and takes one parameter, a representation of the XML document (either its DOM root node or a URL). It should return `true` if the document matches the XPath expression, and `false` if it does not.

To make things simpler, we are supporting a very limited subset of XPath, as specified by the following 'grammar:'

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

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.

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 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 XML document. From there, it is relatively easy to evaluate an XPath expression recursively. Sample code using a DOM parser is available on the course web page. You do not need to use a SAX parser as with XFilter.

1.3 Unit Tests

In addition to the basic code specified above, you must implement at least 5 *unit tests*, using the JUnit 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 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.

2 Milestone 2: Due March 15

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

2.1 Store

You will use Berkeley DB Java Edition (http://www.oracle.com/technology/products/berkeley-db/je/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[2].

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 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**.

2.2 Crawler

Your web crawler will be a Java application that can be run from the command line, as **edu.upenn.cis555.XPathCrawler**. It will take the following command line arguments:

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.
3. The maximum size, in megabytes, of a document to be retrieved from a Web server

[2] The *Getting Started Guide* is available at http://www.oracle.com/technology/documentation/berkeley-db/je/GettingStartedGuide/index.html

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. 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 links by using a HTML parser such as 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. If an HTML or XML document matches one of the XPath expressions, store the document in the store. The crawler must be careful not to search the same page multiple times during a given crawl. The crawler should exit when it has no more pages to crawl.

Your crawler should be a considerate Web citizen. First, it will respect the `robots.txt` file, as described in *A Standard for Robot Exclusion* (http://www.robotstxt.org/wc/norobots.html). Second, it will 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[3]

- `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. It should also not retrieve the file if it has not been modified since the last time it was crawled.

Finally, we expect you to implement your own HTTP client; you may not use the one that comes as part of the Java standard library or any other code that you did not personally write.

2.3 Servlet-based Web Interface

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

- Creation of new accounts
- Login to an existing account
- Log out of the logged in account
- List all channels available on the system
- Create a new channel by specifying its name, the set of XPath expressions, and the URL of the XSL stylesheet
- Delete a channel created by the logged in user

[3] For more details see RFC3023 (http://www.rfc-editor.org/rfc/rfc3023.txt)

- Display a channel

How you implement most of the functionality of the Web 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 XPath expressions, the `<documentcollection>` tag should contain a `<document>` tag, which 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, with any headers, i.e. the starting `<?xml` tag and any other tags that begin with `<?xml`, transcoded into UTF-16.

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, let us know and we *may* be able to find someone who is willing to share his or her code with you.

2.4 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).

2.5 Test cases

You must develop at least 2 JUnit tests for each of the storage system and the crawler.

3 Testing

3.1 ‘Sandbox’

We will implement a small sandbox for you to test your code on. It will run on machines in the Engineering Department, so it will be fast to access, and it will not contain any links out of itself. We will make the address of the sandbox available to you as soon as it is up and running.

3.2 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 on the following pages:

- http://clarkware.com/articles/JUnitPrimer.html
- 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.

4 Version control

We highly recommend that you use a *version control system* for maintaining your project code. Version control offers a convenient way of preserving and managing previous revisions of your source files, so that you may incrementally make modifications and switch back if necessary. It will also become very useful for your final project when all your team members will be modifying code at the same time. Useful functionalities include merging code, logging changes and differencing revision files.

Subversion (http://subversion.tigris.org/) is a popularly used VCS. You may create and use subversion repositories on eniac (or mod) by doing the following:

1. Create a directory for your repository, say `cis455`
2. Execute the command:

   ```
   svnadmin create --pre-1.4-compatible {full_path_to}/cis455
   ```

   Do not modify/add/delete anything in this directory!
3. If you're developing your code on a UNIX system, execute the command (in an empty directory) :

   ```
   svn co svn+ssh://{username}@eniac.seas.upenn.edu/{full_path_to}/cis455
   ```

   This will give you a fresh working copy of your repository. You can develop your code in this directory, and commit any changes using the commands `svn add, svn delete, svn commit`.

   If you're using Windows, you can use a subversion client like TortoiseSVN (http://tortoisesvn.tigris.org/). Your URL will be:

   ```
   svn+ssh://{username}@eniac.seas.upenn.edu/{full_path_to}/cis455
   ```

Other resources:

- Subversion on eniac: http://www.seas.upenn.edu/cets/answers/subversion.html
- Subclipse (Eclipse plug-in) : http://subclipse.tigris.org/

5 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.

5.1 Crawler

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.

5.2 XPath engine

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.

5.3 Channel Subscriptions

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).

5.4 Crawler Web Interface

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
- 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.

6 Submission Instructions

You will be using the turnin feature available on Eniac, to submit all your code and documentation. For Milestones 1 and 2, this should include the source code and a detailed (but concise) README file. Be sure to note the relative path of your html form page, so that we may test it with a browser. For Milestone 2, you will additionally need the **warandpeace.xp** file containing your XPath expressions for the RSS aggregator, and the **rss.xsl** XSL stylesheet.

Note that we are going to be strict about whether you have adhered to the filenames, class names, command-line argument ordering, and so on in the specification: this is a large class, and deviation from the spec dramatically slows down grading, which hurts everyone. Hence a substantial penalty will be deducted in that case.

When you have all your material in a single directory on eniac, execute the following command:

```
turnin -c cis455 -p {hw2m1|hw2m2} {mydir}
```

where you would use either hw2m1 or hw2m2 for the respective milestones, and mydir is the directory containing all your files.

Note that this command bundles your directory contents up into a tar file before submitting. If you execute this command again, your previous tar file will be overwritten.