Chapter 33. Using Web Service APIs

This chapter describes programming techniques for writing standalone programs that employ Web-based service APIs. As a test-bed, I'll implement an application which sends a book title to a service, and gets back a PNG image of the book's cover. I'll write five versions: one each for Google, Amazon, and eBay, and two aimed at Yahoo. There's a high degree of commonality between them, since they all pass through five stages:

1. A search query is *built* as a URL.
2. The query is *sent* to the server-side where it's processed as a HTTP Get request. In the meantime, the client application waits for an answer.
3. A response is eventually *received* back at the client as a string.
4. The string is *parsed* into structured data, typically XML or JSON.
5. Relevant information is *extracted* from the data structure, often using a path expressions language such as XPath.

These stages are shown in Figure 1, which I'll refer to using the mnemonic BSRPE.

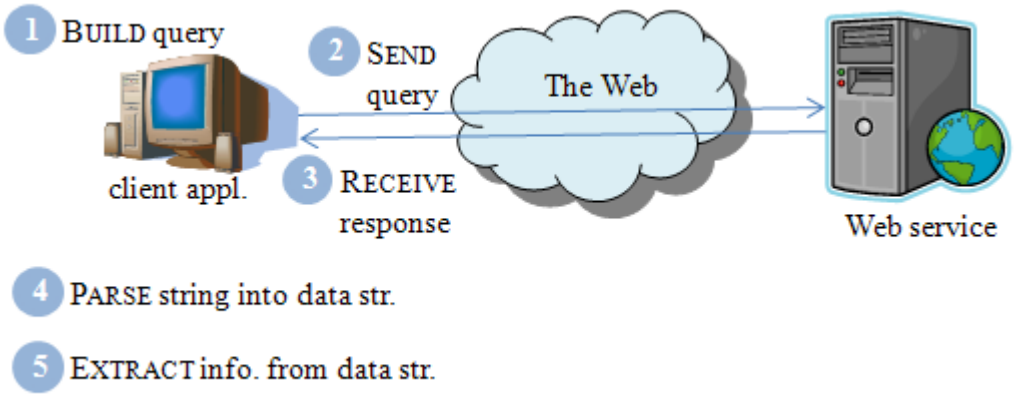

Figure 1. BSRPE: Using a Web Service in Five Stages.

There's usually also a step 0 – the programmer signs up with the Web service to obtain an application ID, and perhaps a password or secret key. The ID and password are included in the URL created during stage 1.

For more complicated searches, the five stages frequently occur twice. The first time they comprise a *search* phase where the user supplies a list of criteria in the URL, and the resulting data structure contains possible matches. A *lookup* phase follows, which also has five stages, where more information is obtained for a particular match.

The service APIs that I employ are:

- Google's AJAX multimedia Search API (http://code.google.com/apis/ajaxsearch/)
- Amazon's Advertising API, formerly known as Amazon Associates Web Services (http://docs.amazonwebservices.com/AWSECommerceService/latest/DG/)

- eBay's Shopping API (http://developer.ebay.com/developercenter/java/shopping/)
- Yahoo's Product Search API for its Yahoo! Shopping site (http://developer.yahoo.com/shopping/V3/productSearch.html)

These services offer multiple language interfaces, including JavaScript, PHP, and Flash, but I'll utilize their Java versions so I can write standalone applications that doesn't require a Web browser.

1. Getting Images from Google

The first version of my book cover image searcher uses Google's AJAX Search API (http://code.google.com/apis/ajaxsearch/) which returns results in the JSON format. The API can be applied in three main areas: the Web, location-based queries based on Google Maps, and for retrieving multimedia (image and video).

Unlike the other Web services I'll be using, Google doesn't require an application ID, although you can register for one if you wish (see http://code.google.com/apis/ajaxsearch/key.html).

The API's documentation is excellent, although somewhat slanted towards JavaScript (see http://code.google.com/apis/ajaxsearch/documentation/). Java and other languages are discussed in the "Flash and other Non-JavaScript Environments" subsection of the documentation (http://code.google.com/apis/ajaxsearch/documentation/reference.html#_intro_fonje). More examples can be found in the article:

> *Using Google's AJAX Search API with Java*
> `http://www.ajaxlines.com/ajax/stuff/article/using_google_is_ajax_search_api_with_java.php`

My SearchGoogle application begins by reading a book title string from the command line, which it passes to bookCoverSearch() for processing.

```
public static void main(String args[])
{
  if (args.length == 0) {
    System.err.println("Supply a title");
    System.exit(1);
  }
  String imUrlStr = bookCoverSearch(args[0]);
  WebUtils.downloadImage(imUrlStr);
}  // end of main()
```

bookCoverSearch() implements the five BSRPE stages:

```
private static String bookCoverSearch(String title)
{
  System.out.println("Searching Google Image for \"" + title + "\"");
  try {
    // Convert spaces to +, etc. to make a valid URL
    String ueTitle = URLEncoder.encode("\"" + title + "\"" +
                                       "\"book cover\"", "UTF-8");
    String urlStr =
```

```
          "http://ajax.googleapis.com/ajax/services/search/images" +
                "?v=1.0&q=" + ueTitle;
    String jsonStr = WebUtils.webGetString(urlStr);

    JSONObject json = new JSONObject(jsonStr);
    WebUtils.saveString("temp.json", json.toString(2) );
                                        // indent the string
    return getCoverURL(json);
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of bookCoverSearch()
```

Image searches are carried out at http://ajax.googleapis.com/ajax/services/search/images; the addresses for Web, local, video, blog, news, book, and patent searches are listed at http://code.google.com/apis/ajaxsearch/documentation/reference.html#_restUrlBase.

The URL arguments (the name/value pairs following the '?') are "v" for the search engine version, and "q" for the URL-encoded query. The query is a combination of the user's supplied title (e.g. "Killer Game Programming in Java", placed in double quotes) and the phrase "book cover", which helps to reduce the number of matches.

The user's query is converted to URL-encoded form using URLEncoder.encode(), which replaces spaces and special characters by UTF-8 characters. The construction of the urlStr string is stage 1 of BSRPE.

WebUtils.webGetString() transmits the query to the service and returns the reply as a string (these tasks correspond to stages 2 and 3 of BSRPE):

```
public static String webGetString(String urlStr)
// contact the specified URL, and return the response as a string
{
  System.out.println("Contacting \"" + urlStr + "\"");
  try {
    URL url = new URL(urlStr);

    String line;
    StringBuilder sb = new StringBuilder();
    BufferedReader reader = new BufferedReader(
                    new InputStreamReader(url.openStream()));
    while((line = reader.readLine()) != null)
      sb.append(line);
    reader.close();
    return sb.toString();
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }

  return null;
}  // end of webGetString()
```


WebUtils is my own class, holding a range of useful methods for Web querying, DOM parsing, and image downloading.

The string returned by Google will usually be very lengthy, consisting of multiple JSON structures (name/value pairs and lists). To be more easily readable, the string is parsed into a JSON data structure, which requires a third-party library since the JDK doesn't have any built-in JSON support.

There are almost twenty JSON libraries listed at http://www.json.org/, with the simplest probably being the org.json package, available from http://www.json.org/java/. An interesting comparison of five popular JSON libraries can be found at http://www.rojotek.com/blog/2009/05/07/a-review-of-5-java-json-libraries/.

The string returned from Google is converted to a JSON data structure using:

```
JSONObject json = new JSONObject(jsonStr);
```

It's very useful to write this structure out to a text file, suitable indented:

```
WebUtils.saveString("temp.json", json.toString(2) );
```

Storing the structure in a file makes it much easier to study, and determine what needs to be extracted in the final BSRPE stage.

Google's JSON format is explained at http://code.google.com/apis/ajaxsearch/documentation/reference.html#_restUrlBase. The structure is essentially the following:

```
{
  "responseData" : {
    "cursor" : {. . .}    // the useful stuff is in here
    "results" : [. . .],  // and here
  },
  "responseDetails" : null,
  "responseStatus" : 200
}
```

By default, the results list will only contain at most five matches, but the actual total number is stored in "estimatedResultCount" inside "cursor":

```
    "cursor": {
      "currentPageIndex": 0,
      "estimatedResultCount": "25",
       :  // more key/value pairs
    }
```

Each matching result consists of many key/value pairs, including the image's title, dimensions, and its URL. For example:

```
    "results": [
      {
        "contentNoFormatting": "Killer Game Programming in Java",
        "height": "500",
        "title": "5147ezBIJxL.jpg",
        "url":
           "http://ecx.images-amazon.com/images/I/5147ezBIJxL.jpg",
        "width": "377"
          :     // more key/value pairs
      },
```



```
      :  // more results tuples {. . .}
    ]
```

The tuples and lists can be examined using the org.json get methods: getJSONObject(), getJSONArray(), and getString(). However, more powerful search methods are often useful, as we'll see when dealing with Yahoo.

getCoverURL() reports the total number of matches (by accessing the "estimatedResultCount" field), prints the contents headers and URLs of the matches, and returns the first URL. getCoverURL() corresponds to BSRPE stage 5 (extraction).

```
private static String getCoverURL(JSONObject json)
{
  try {
    System.out.println("\nTotal no. of possible results: " +
                    json.getJSONObject("responseData")
                        .getJSONObject("cursor")
                        .getString("estimatedResultCount") + "\n");

    // list the content headers and URLs
    JSONArray ja = json.getJSONObject("responseData").
                        getJSONArray("results");
    for (int i = 0; i < ja.length(); i++) {
      System.out.print((i+1) + ". ");
      JSONObject j = ja.getJSONObject(i);
      System.out.println("Content: " +
                              j.getString("contentNoFormatting"));
      System.out.println("       URL: "+ j.getString("url") + "\n");
    }
    if (ja.length() > 0)
      return ja.getJSONObject(0).getString("url");  //return 1st URL
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
} // end of getCoverURL()
```

For instance, getCoverURL() prints the following when SearchGoogle is asked to look for "killer game programming in java":

```
Total no. of possible results: 25

1. Content: Killer Game Programming in Java
       URL: http://ecx.images-amazon.com/images/I/5147ezBIJxL.jpg

2. Content: Killer Game Programming in Java by ...
       URL: http://images.barnesandnoble.com/images/
                                            15710000/15711693.JPG

3. Content: Killer Game Programming in Java
       URL: http://ecx.images-amazon.com/images/
                                         I/5147ezBIJxL._SL160_.jpg

4. Content: Book Cover
       URL: http://images.barnesandnoble.com/images/
```



```
                                        15710000/15711692.jpg
```

getCoverURL() returns the first URL (http://ecx.images-amazon.com/images/I/5147ezBIJxL.jpg).

WebUtils.downloadImage() retrieves the image stored at the specified URL:

```
public static void downloadImage(String urlStr)
{
  System.out.println("Downloading image at:\n\t" + urlStr);
  URL url = null;
  BufferedImage image = null;
  try {
    url = new URL(urlStr);
    image = ImageIO.read(url);
  }
  catch (IOException e)
  {  System.out.println("Problem downloading");  }

  if (image != null) {
    String fnm = urlStr.substring( urlStr.lastIndexOf("/")+1 );
    savePNG(fnm, image);
  }
}  // end of downloadImage()


private static void savePNG(String fnm, BufferedImage im)
// save the image as a PNG in <fnm>Cover.png
{
  int extPosn = fnm.lastIndexOf(".");
  String pngFnm;
  if (extPosn == -1)
     pngFnm = "cover.png";
  else
    pngFnm = fnm.substring(0, extPosn) + "Cover.png";
  System.out.println("Saving image to " + pngFnm);

  try {
    ImageIO.write(im, "png", new File(pngFnm));
  }
  catch (IOException e)
  {  System.out.println("Could not save file");  }
}  // end of savePNG()
```

The name of the file uses the URL filename, and appends "Cover.png" to it.

2. Getting Images from Amazon

Amazon is a natural place to search since I'm interested in finding book cover images. Its Product Advertising API, formerly known as the Amazon Associates Web Service, offers a way to access item descriptions and images, and also customer reviews and third-party seller content. The API is described in the online developer guide at http://docs.amazonwebservices.com/AWSECommerceService/latest/DG/

The extensive documentation can be a bit daunting at first, but a good place to start is the "Programming Guide » Getting Set Up" subsection which explains how you must first register as an Amazon Associate, and then obtain an access Key ID (sometimes called a AWS Access Key ID) and a secret key. Until August 15th 2009, only a key ID is needed to use the API, but I'll explain the post-August process that involves digital signing of queries.

The best overview of the API's capabilities can be found in the "Visual Introduction to Product Advertising API" section, which provides a series of screenshots from http://www.amazon.com that show how its Web page elements map onto API operations. They fall into four groups:

- operations for finding items (which are explained via a screenshot of Amazon's main search page);
- operations for finding out more about an item (tied to a picture of an Amazon listings page);
- operations to retrieve seller information;
- operations for implementing a shopping cart.

I'll only need two operations: ItemSearch for finding items, and ItemLookup for retrieving more details about a particular item. These functions are described in detail in the API reference part of the developers guide at:

Programming Guide » API Reference » Operations » ItemSearch

and Programming Guide » API Reference » Operations » ItemLookup

Each page includes small examples showing how to formulate the operation calls as URLs.

My search approach is to use ItemSearch to look for all books with a given name. A list of matches is returned, ending a *search* BSRPE phase, and an Amazon item ID number (an ASIN) is selected. The ASIN is employed in a *lookup* BSRPE phase to obtain more details about a chosen book, including its cover image URL, which is used to download the cover.

2.1. XPath

Amazon's API differs from Google's is that information is returned in XML form, and is considerably more structured (since an Amazon item can be so many different things). Directly searching through the XML data using get() methods can be quite tedious, and a lot of coding effort can be saved if XPath is used instead.

XPath is a query language for selecting nodes from a tree representation of an XML document. A path through the tree is defined using a UNIX-style directory path notation which makes it very easy to gather information.

Three good introductions to XPath can be found at:

- The XPath tutorial at http://www.w3schools.com/xpath/
- "The Java XPath API", by Elliotte Rusty Harold, at http://www.ibm.com/developerworks/library/x-javaxpathapi.html

- "Pro XML development with Java technology" by Deepak Vohra at http://www.onjava.com/pub/a/onjava/2005/01/12/xpath.html

Java contains good support for XPath except in the area of namespace contexts. The articles by Elliotte Rusty Harold and Deepak Vohra go into the details, and Vohra offers a useful helper class, NamespaceContextImpl, in the comments section of his article; I reuse that class here.

2.2. Signing Your Queries

As mentioned above, after August 15th 2009 Amazon service queries will need to be digitally signed. The signing documentation can be found in the developers guide at:

> Programming Guide » Requests » Request Authentication

When a request is sent to Amazon it must include the programmer's Access Key ID to identify the sender, and be digitally signed with the programmer's secret Access Key. The request will only be processed if the digital signature matches the sender's ID.

The code for creating a signature is somewhat involved (ten distinct coding steps are identified), but fortunately Amazon supplies a Java implementation (in SignedRequestsHelper.java), which can be downloaded from:

> Programming Guide » Requests » Request Authentication » Authenticating REST Requests » Java Sample Code for Calculating Signature Version 2 Signatures

This utilizes Apache Commons Codec, which is a separate download from http://commons.apache.org/codec/

My version of SignedRequestsHelper is a slight modification which supplies the Access Key ID and secret Access Key as arguments to its constructor.

2.3. The SignSearchAmazon Class

The main() function for my Amazon search code:

```
// globals
// CHANGE THESE 2 LINES
private static final String KEY_ID = "????";
private static final String SECRET_KEY = "????";

private static XPath xpath;
private static SignedRequestsHelper signer;


public static void main(String args[])
{
  if (args.length == 0) {
    System.err.println("Supply a title");
    System.exit(1);
  }

  xpath = createXPath();
  signer = new SignedRequestsHelper(KEY_ID, SECRET_KEY);

  // get the Amazon ASIN for a matching book title
```

```
  String asin = bookTitleSearch(args[0]);    // search phase

  // get the URL of the image used with that ASIN listing
  String urlStr = imageLookup(asin);        // lookup phase

  WebUtils.downloadImage(urlStr);
}  // end of main()
```

The XPath namespace and the signer code are initialized first, then the search BSRPE phase is carried out by bookTitleSearch(). The lookup BSRPE phase is handled by imageLookup(), and the image is downloaded and saved with WebUtils.downloadImage().

The values for the key ID and secret key will depend on the programmer's registration at Amazon.

Although createXPath() appears at the beginning of main(), it can't (easily) be written until a preliminary version of bookTitleSearch() has been executed, so I'll look at that method first.

2.4. The Amazon Search Phase

bookTitleSearch() queries Amazon for all the book items which match the specified title string. Depending on the generality of the title, many items may match, but only the ASIN of the first is returned.

```
private static String bookTitleSearch(String title)
{
  System.out.println("Searching Amazon for the book title \"" +
                                                    title + "\"");
  try {
    String ueTitle = URLEncoder.encode(title, "UTF-8");

    HashMap<String, String> urlParams =
                                new HashMap<String, String>();
    urlParams.put("Service", "AWSECommerceService");
    urlParams.put("Operation", "ItemSearch");   // use ItemSearch
    urlParams.put("SearchIndex", "Books");   // limit search to books
    urlParams.put("Power", "title:\"" + title + "\"");
                             // place double quotes around title

    String urlStr = signer.sign(urlParams);

    Document doc = WebUtils.parse( WebUtils.webGet(urlStr) );
    WebUtils.saveDoc(doc, "temp.xml");

    return getASIN(doc);
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }

  return null;
}  // end of bookTitleSearch()
```


bookTitleSearch() creates a HashMap for passing URL query arguments to the SignedRequestsHelper object, signer.

The two essential parameters are "Service" and "Operation"– the service is called "AWSECommerceService", which I discovered by looking at the examples for ItemSearch in the developers guide. The "SearchIndex" parameter limits the search to a particular category of item, a list of which can be found by following the links for that parameter in the ItemSearch documentation. The "Power" parameter is also explained in the documentation.

bookTitleSearch() employs WebUtils. webGet() to obtain an input stream for receiving the response from Amazon:

```
public static InputStream webGet(String urlStr)
{
  System.out.println("Contacting \"" + urlStr + "\"");
  try {
    URL url = new URL(urlStr);
    return url.openStream();
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of webGet()
```

WebUtils.parse() parses the stream's incoming XML into a DOM document:

```
public static Document parse(InputStream is)
{
  Document document = null;
  DocumentBuilderFactory factory =
              DocumentBuilderFactory.newInstance();
  factory.setValidating(false);
  factory.setNamespaceAware(true);

  try {
    DocumentBuilder builder = factory.newDocumentBuilder();
    document = builder.parse(is);
    return document;
  }
  catch (SAXParseException spe) {    // Error generated by the parser
    System.out.println("\n** Parsing error , line " +
            spe.getLineNumber() + ", uri " + spe.getSystemId());
    System.out.println(" " + spe.getMessage());
    // Use the contained exception, if any
    Exception x = spe;
    if (spe.getException() != null)
      x = spe.getException();
    x.printStackTrace();
  }
  catch (SAXException sxe) {  // Error generated during parsing
    Exception x = sxe;
    if (sxe.getException() != null)
      x = sxe.getException();
    x.printStackTrace();
  }
```

```
  catch (ParserConfigurationException pce) {
    // Parser with specified options can't be built
    pce.printStackTrace();
  }
  catch (IOException ioe)
  {ioe.printStackTrace();  }

  return null;
}  // end of parse()
```

Most of parse() is spent dealing with different kinds of exceptions.

A document is usually so large that it's best to save it to a file:

```
public static void saveDoc(Document doc, String fnm)
{
  try {
    Transformer xformer =
          TransformerFactory.newInstance().newTransformer();

    // setup indenting to "pretty print"
    xformer.setOutputProperty(OutputKeys.INDENT, "yes");
    xformer.setOutputProperty(
            "{http://xml.apache.org/xslt}indent-amount", "2");

    // write document to the file
    System.out.println("Saving document in " + fnm);
    xformer.transform(new DOMSource(doc),
                      new StreamResult(new File(fnm)));
  }
  catch (TransformerConfigurationException e)
  { System.out.println("TransformerConfigurationException: " + e); }
  catch (TransformerException e)
  {  System.out.println("TransformerException: " + e);  }
}  // end of saveDoc()
```

An XML transformer ensures that the document is indented as it's written out, making it much easier to read. For example, when Amazon is searched for the title "killer game", the file has the form:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ItemSearchResponse xmlns="http://webservices.amazon.com/
                           AWSECommerceService/2005-10-05">
  <OperationRequest>
    // various operation details
  </OperationRequest>

  <Items>
    <Request>
      // ItemSearch Request details
    </Request>
    <TotalResults>69</TotalResults>
    <TotalPages>7</TotalPages>

    <Item>
      <ASIN>0596007302</ASIN>
      <DetailPageURL>http://www.amazon.com/
           Killer-Game-Programming-Andrew-Davison/...</DetailPageURL>
```



```
      <ItemAttributes>
        <Author>Andrew Davison</Author>
        <Manufacturer>O'Reilly Media, Inc.</Manufacturer>
        <ProductGroup>Book</ProductGroup>
        <Title>Killer Game Programming in Java</Title>
      </ItemAttributes>
    </Item>

    <Item>
      <ASIN>0818406623</ASIN>
      <DetailPageURL>http://www.amazon.com/
               Killer-Poker-No-Limit-Tournaments/...</DetailPageURL>
      <ItemAttributes>
        <Author>John Vorhaus</Author>
        <Manufacturer>Lyle Stuart</Manufacturer>
        <ProductGroup>Book</ProductGroup>
        <Title>Killer Poker No Limit Games and Tournaments</Title>
      </ItemAttributes>
    </Item>

      :  // more Items
  </Items>
</ItemSearchResponse>
```

The matching items are represented by a series of Item elements, nested inside a Items element, in a ItemSearchResponse element. Each item contains several sub-elements holding its title, author, and other details, but the most important value is the item's ASIN, which uniquely identifies it to Amazon.

The "xmlns" attribute gives the XML namespace used by the elements (http://webservices.amazon.com/AWSECommerceService/2005-10-05). This namespace is required in XPath queries that search the document, and is set up in createXPath(), called from main():

```
private static XPath createXPath()
// create a XPath object that understands the Amazon Namespace
{
  XPath xpath = XPathFactory.newInstance().newXPath();
  NamespaceContextImpl ns = new NamespaceContextImpl("amz",
    "http://webservices.amazon.com/AWSECommerceService/2005-10-05");
  xpath.setNamespaceContext(ns);
  return xpath;
}  // end of createXPath()
```

The NamespaceContextImpl class, developed by Deepak Vohra, is a convenient way to assign a short name to a namespace (in this case "amz"). This short form can be used instead of the URL when referring to a namespace inside an XPath expression.

Extracting Item Search Information

I developed getASIN() after studying the file output from bookTitleSearch(). getASIN() prints out all the returned matching titles and ASINs, and returns the first ASIN.

```
private static String getASIN(Document doc)
{
  try {
    String totalStr =      // get the total no. of results
       (String) xpath.evaluate("//amz:TotalResults",
                               doc, XPathConstants.STRING);
    System.out.println("Total no. of item matches: " + totalStr);

    // get the titles and their ASINs
    NodeList nodes = doc.getElementsByTagName("Item"); //get items
    String firstAsin = null;
    for (int i = 0; i < nodes.getLength(); i++) {
      System.out.print( (i+1) + ". ");
      Element elem = (Element) nodes.item(i);
      System.out.println("Title: " +
                         WebUtils.getElemText(elem, "Title"));
      String asin = WebUtils.getElemText(elem, "ASIN");  // get ASI
      System.out.println("   ASIN: " + asin);
      if (i == 0)  // store first match
        firstAsin = asin;
    }
    return firstAsin;
  }
  catch(Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
 }  // end of getASIN()
```

getASIN() includes an XPath query for extracting the total number of matching items, but mostly uses Document methods, including Document.getElementsByTagName() which returns a list of all the elements with the specified tag name.

The XPath expression is:

```
String totalStr =
    xpath.evaluate("//amz:TotalResults", doc, XPathConstants.STRING);
```

"//amz:TotalResults" denotes all elements with the tag name "TotalResults". Its evaluation is in the document's Amazon namespace, and the result is returned as a string.

The rest of getASIN() is a loop which iterates through a list of Item elements, collected by a call to:

```
NodeList nodes = doc.getElementsByTagName("Item"); //get items
```

For each item, its title and ASIN elements are printed out, with the help of WebUtils.getElemText():

```
public static String getElemText(Element elem, String tagName)
```



```
{
  NodeList nodeList = elem.getElementsByTagName(tagName);
  if (nodeList.getLength() > 0)
    return nodeList.item(0).getFirstChild().getNodeValue();
  return null;
}  // end of getElemText()
```

getElemText() calls Document.getElementsByTagName() to get a list of matching elements, and the first is returned.

The output of getASIN() for a "killer game" title query is:

```
Total no. of item matches: 69
1. Title: Killer Game Programming in Java
   ASIN: 0596007302
2. Title: Killer Poker No Limit Games and Tournaments
   ASIN: 0818406623
3. Title: Killer Poker Online, Vol. 2: Advanced Strategies for
Crushing the Internet Game
   ASIN: 0818406615
4. Title: Killer Poker Online: Crushing the Internet Game
   ASIN: 0818406313
     :   // more matches
10. Title: Terminate with Extreme Prejudice: Inside the Assassination
Game - First-hand Stories from Hired Killers and Their Paymasters
   ASIN: 1841199478
```

getASIN() returns the first ASIN that it encounters, which for the example above is 0596007302.

2.5. The Amazon Lookup Phase

bookTitleSearch() finishes its BSRPE search by returning an ASIN to main(). imageLookup() uses this value for a BSRPE lookup: Amazon is queried for more details on the specific ASIN, so its cover image URL can be obtained. The following code fragment from main() shows the two phases:

```
// get the Amazon ASIN for a matching book title
String asin = bookTitleSearch(args[0]);    // search phase

// get the URL of the image used with that ASIN listing
String urlStr = imageLookup(asin);        // lookup phase
```

imageLookup() bears a striking resemblance to bookTitleSearch() since it consists of similar BSRPE stages:

```
private static String imageLookup(String asin)
{
  System.out.println("Retrieving large image for Amazon
                                        item with ASIN: " + asin);
  try {
    HashMap<String, String> urlParams =
                        new HashMap<String, String>();
    urlParams.put("Service", "AWSECommerceService");
    urlParams.put("Operation", "ItemLookup");  // use ItemLookup
```

```
    urlParams.put("ItemId", asin);             //use asin to get item
    urlParams.put("ResponseGroup", "Images");  // limit to images

    String urlStr = signer.sign(urlParams);

    Document doc = WebUtils.parse( WebUtils.webGet(urlStr) );
    WebUtils.saveDoc(doc, "temp.xml");

    // return the URL of the large image for this item
    return (String) xpath.evaluate(
                "//amz:Item/amz:LargeImage/amz:URL",
                              doc, XPathConstants.STRING);
  }
  catch (Exception e)
  { System.out.println( e );
    System.exit(1);
  }
  return null;
}  // end of imageLookup()
```

This time around, the ItemLookup operation is employed, whose parameters are explained in the API reference:

> Programming Guide » API Reference » Operations » ItemLookup

The item is specified by its ASIN value, and the response details are limited to image information by setting the "ResponseGroup" parameter to "Images".

The response is parsed as a DOM document, and saved to a file for study. The contents when ASIN is 0596007302 are:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ItemLookupResponse xmlns="http://webservices.amazon.com/
                                  AWSECommerceService/2005-10-05">
  <OperationRequest>
    // various operation details
  </OperationRequest>
  <Items>
    <Request>
      // ItemLookup Request details
    </Request>
    <Item>
      <ASIN>0596007302</ASIN>
      <SmallImage>
        <URL>http://ecx.images-amazon.com/images/
                         I/5147ezBIJxL._SL75_.jpg</URL>
        <Height Units="pixels">75</Height>
        <Width Units="pixels">57</Width>
      </SmallImage>
      <MediumImage>
        <URL>http://ecx.images-amazon.com/images/
                        I/5147ezBIJxL._SL160_.jpg</URL>
        <Height Units="pixels">160</Height>
        <Width Units="pixels">121</Width>
      </MediumImage>
      <LargeImage>
        <URL>http://ecx.images-amazon.com/images/
                                I/5147ezBIJxL.jpg</URL>
        <Height Units="pixels">500</Height>
        <Width Units="pixels">377</Width>
```



```
      </LargeImage>
      <ImageSets>
        // Image sets details
      </ImageSets>
    </Item>
  </Items>
</ItemLookupResponse>
```

Even restricting the output to image information for one item still produces voluminous details (due to the many different sized images for the item).

I want the URL of the largest image, which is the URL element inside the LargeImage element in the Item element. This can be succinctly represented by the XPath expression:

```
return (String) xpath.evaluate(
                 "//amz:Item/amz:LargeImage/amz:URL",
                              doc, XPathConstants.STRING);
```

The expression retrieves the "Item/LargeImage/URL" value in the Amazon namespace of the document. For the example above, this will be http://ecx.images-amazon.com/images/I/5147ezBIJxL.jpg.

3. Getting Images from eBay

Amazon is a great way to find covers for current books, but isn't so useful for older, out-of-print texts. One way of finding those is to look through the large collection of books for sale at eBay.

I joined eBay's developers programme at https://developer.ebay.com/, by clicking on the "Join Now" button. After entering my details, several keys were generated, including an AppID which I need for querying eBay.

eBay offers a variety of service APIs, including ones focused on shopping, merchandising, feedback, and trading. The Java versions of these API can be accessed at http://developer.ebay.com/developercenter/java/.

I'm utilizing the shopping API, which parallels the eBay shopping website, and is explained at http://developer.ebay.com/developercenter/java/shopping/. It includes links to a Call Reference guide (i.e. operation descriptions with examples), and a getting started guide.

I'll only need two operations: FindItemsAdvanced for finding items, and GetSingleItem for listing more details about a particular item. These methods are described in the API reference part of the developers guide at:

http://developer.ebay.com/DevZone/shopping/docs/CallRef/FindItemsAdvanced.html

and

http://developer.ebay.com/DevZone/shopping/docs/CallRef/GetSingleItem.html

Each page includes small examples showing how to formulate queries as URLs.

The basic approach is similar to the one I used with Amazon: utilize FindItemsAdvanced to search for all books with a given name. A list of matches is returned, ending a *search* BSRPE phase, and a item ID number is selected. The ID is

employed with GetSingleItem in a *lookup* BSRPE phase to obtain more details about the chosen book, including its cover image URL, which is downloaded.

eBay is simpler than Amazon in that there's no need to digitally sign URL requests; only the AppID must be included with a query. eBay can return its results in XML or JSON form, and I'll use XML since I can then employ XPath to extract information from the document.

3.1. The SearchEbay Class

The main() method of SearchEBay has four parts: the initialization of XPath, a search BSRPE phase, a lookup BSRPE phase, and the downloading of the cover image:

```
// global
private static XPath xpath;


public static void main(String args[])
{
  if (args.length == 0) {
    System.err.println("Supply a title");
    System.exit(1);
  }

  xpath = createXPath();
  String itemID = bookTitleSearch(args[0]);  // search phase
  String imUrlStr = pictureLookup(itemID);   // lookup phase
  WebUtils.downloadImage(imUrlStr);
}  // end of main()
```

createXPath()'s allocates a short name to the eBay XML namespace, and was written after I'd got bookTitleSearch() working, and so knew eBay's namespace.

3.2. The eBay Search Phase

bookTitleSearch() queries eBay for a book with a specified title, and prints a list of matching items. The first item ID is returned to main().

```
// global
private static String APP_ID = "????";    // CHANGE THIS


private static String bookTitleSearch(String title)
{
  System.out.println("Searching eBay for \"" + title + "\"");
  try {
    String ueTitle = URLEncoder.encode(title, "UTF-8");

    String urlStr = "http://open.api.ebay.com/shopping?" +
           "callname=FindItemsAdvanced&" +   // FindItemsAdvanced op.
           "responseencoding=XML&" +   // could be JSON
           "appid=" + APP_ID + "&" +
           "siteid=0&" +
           "version=525&" +
           "CategoryID=267&" +    // restrict to books
```

```
            "QueryKeywords=" + ueTitle + "&" +
            "MaxEntries=5" ;   // at most 5 matches returned

    Document doc = WebUtils.parse( WebUtils.webGet(urlStr) );
    WebUtils.saveDoc(doc, "temp.xml");
    return getItemID(doc);
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of bookTitleSearch()
```

I formulated the URL by looking at samples in the FindItemsAdvanced operation documentation at http://developer.ebay.com/DevZone/shopping/docs/CallRef/FindItemsAdvanced.html#Samples. The "CategoryID" argument restricts the search to books. Category IDs can be obtained by sending a GetCategoryInfo request to eBay, as explained at http://developer.ebay.com/devzone/shopping/docs/callref/GetCategoryInfo.html. If a simpler search based only on keywords is sufficient, then the FindItems operation can be used instead (see http://developer.ebay.com/DevZone/shopping/docs/CallRef/FindItems.html).

The XML data is parsed into a DOM document, and saved to a temporary file. After studying the file, I wrote the getItemID() and createXPath() methods.

A search for "gunpowder plot" results in the following XML being saved:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<FindItemsAdvancedResponse xmlns="urn:ebay:apis:eBLBaseComponents">
   <Timestamp>2009-06-27T15:29:47.604Z</Timestamp>
   // other result elements
   <SearchResult>
    <ItemArray>
     <Item>
      <ItemID>110405406027</ItemID>
      <GalleryURL>http://thumbs1.ebaystatic.com/
                    pict/1104054060278080_1.jpg</GalleryURL>
      <Title>Gunpowder Plot (Daisy Dalrymple), Carola Dunn</Title>
      // other item elements
     </Item>
     <Item>
      <ItemID>120439365464</ItemID>
      <GalleryURL>http://thumbs3.ebaystatic.com/
                   pict/1204393654648080_1.jpg</GalleryURL>
      <Title>GUNPOWDER PLOT BY CAROLA DUNN ~ HC/DJ ~ </Title>
      // other item elements
     </Item>
     <Item>
       :  // more Items
    </ItemArray>
   </SearchResult>
   <PageNumber>1</PageNumber>
   <TotalPages>5</TotalPages>
   <TotalItems>25</TotalItems>
 </FindItemsAdvancedResponse>
```

The eBay namespace is given at the start (the "xmlns" attribute), which I added to createXPath():

```
// global
private static XPath xpath;

private static XPath createXPath()
// create a XPath object that understands the ebay Namespace
{
  XPath xpath = XPathFactory.newInstance().newXPath();
  NamespaceContextImpl ns =
         new NamespaceContextImpl("ebay",
                   "urn:ebay:apis:eBLBaseComponents");
  xpath.setNamespaceContext(ns);
  return xpath;
}  // end of createXPath()
```

createXPath() defines "eBay" to be the namespace's short form.

The rest of the file's XML shows that a series of Item elements are stored inside an ItemArray element. Each Item element includes an ID, title, and gallery image URL. I don't use the gallery image as my search result since it's a thumbnail, so too small. I need the FindItem operation to get a URL for a bigger version of the image.

getItemID() prints all the matching item's titles and IDs, and returns the first item ID.

```
private static String getItemID(Document doc)
{
  try {
    String totalStr =      // get the total no. of results
       (String) xpath.evaluate("//ebay:TotalItems", doc,
                                    XPathConstants.STRING);
    System.out.println("Total no. of item matches: " + totalStr);

    // get the titles and their item IDs
    NodeList nodes = doc.getElementsByTagName("Item");//get all items
    String firstItemID = null;
    for (int i = 0; i < nodes.getLength(); i++) {
      System.out.print( (i+1) + ". ");
      Element elem = (Element) nodes.item(i);
      System.out.println("Title: " +
                      WebUtils.getElemText(elem, "Title"));
      String itemID = WebUtils.getElemText(elem, "ItemID");
      System.out.println("   ItemID: " + itemID);
      if (i == 0)  // store first match
        firstItemID = itemID;
    }
    return firstItemID;
  }
  catch(Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of getItemID()
```


The code is very similar to getASIN() in the Amazon search code.

getItemID() includes a simple XPath query for extracting the total number of matching items, but mostly uses Document methods.

The XPath expression is:

```
String totalStr = (String)
    xpath.evaluate("//ebay:TotalItems", doc, XPathConstants.STRING);
```

"//ebay:TotalItems" denotes all the TotalItems elements in the eBay namespace.

The rest of getItemID() is a loop which iterates through a list of Item elements, collected by a call to:

```
NodeList nodes = doc.getElementsByTagName("Item");
```

Each item's title and ID are printed out with the help of WebUtils.getElemText(). For example, the output for a "gunpowder plot" query is:

```
Total no. of item matches: 25
1. Title: Gunpowder Plot (Daisy Dalrymple), Carola Dunn
   ItemID: 110405406027
2. Title: GUNPOWDER PLOT BY CAROLA DUNN ~ HC/DJ ~
   ItemID: 120439365464
3. Title: The Gunpowder Plot: Terror and Faith in 1605, Antonia F
   ItemID: 400057090383
4. Title: James I VI England Scotland Gunpowder Plot History 1605
   ItemID: 310151190068
5. Title: Walking Notorious London : From Gunpowder Plot to Gangl
   ItemID: 120440071073
```

getItemID() returns the first item ID which, for the example above, is 110405406027.

3.3. The eBay Lookup Phase

bookTitleSearch() finishes its BSRPE search phase by returning an item ID to main(). It passes the number to pictureLookup(), which commences a BSRPE lookup: eBay is queried for more details on the ID, so its large image URL can be obtained. The following code fragment from main() shows the two phases:

```
String itemID = bookTitleSearch(args[0]);  // search phase
String imUrlStr = pictureLookup(itemID);   // lookup phase
```

pictureLookup() is quite similar to bookTitleSearch() since it implements BSRPE:

```
private static String pictureLookup(String itemID)
{
  System.out.println("Retrieving picture for eBay item: " + itemID);
  try {
    String urlStr = "http://open.api.ebay.com/shopping?"
             "callname=GetSingleItem&" +    // GetSingleItem op.
             "responseencoding=XML&" +      // could be JSON
             "appid=" + APP_ID + "&" +
             "siteid=0&" +
```

```
              "version=525&" +
              "ItemID=" + itemID + "&" ;

    Document doc = WebUtils.parse( WebUtils.webGet(urlStr) );
    WebUtils.saveDoc(doc, "temp.xml");

    return getPictureURL(doc);
  }
  catch (Exception e)
  { System.out.println( e );
    System.exit(1);
  }
  return null;
}  // end of pictureLookup()
```

The GetSingleItem operation is explained in the API reference at http://developer.ebay.com/DevZone/shopping/docs/CallRef/GetSingleItem.html.

The item of interest is specified with an ID. The eBay response is parsed into a DOM document, and saved to a file for further study. The contents when the ID is 110405406027 are:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<GetSingleItemResponse xmlns="urn:ebay:apis:eBLBaseComponents">
   <Timestamp>2009-06-27T16:01:06.124Z</Timestamp>
   // other result elements
   <Item>
    <ItemID>110405406027</ItemID>
    <GalleryURL>http://thumbs1.ebaystatic.com/
                      pict/1104054060278080_1.jpg</GalleryURL>
    <PictureURL>http://i24.ebayimg.com/01/c/000/77/
                     3e/3c20_6.JPG</PictureURL>
    <Title>Gunpowder Plot (Daisy Dalrymple), Carola Dunn</Title>
    // other item elements
   </Item>
  </GetSingleItemResponse>
```

I only want the PictureURL element in the Item element, but some care must be taken with the XPath expression, since it's possible for an eBay item to include multiple images, or even no image. These cases are dealt with by getPictureURL():

```
private static String getPictureURL(Document doc)
{
  try {
    NodeList nodes = (NodeList)
             xpath.evaluate("//ebay:PictureURL/text()",
                            doc, XPathConstants.NODESET);
    if (nodes.getLength() == 0)
      return null;
    if (nodes.getLength() > 1)
      System.out.println("No. of pictures: " + nodes.getLength());

    return (String) nodes.item(0).getNodeValue();
  }
  catch(Exception e)
  {  System.out.println(e);  }
  return null;
```



```
}  // end of getPictureURL()
```

The expression retrieves all the PictureURL values in the eBay namespace of the document. The size of the list is checked, and the textual value of the first element is returned. For the example above, this will be http://i24.ebayimg.com/01/c/000/77/3e/3c20_6.JPG.

4. Getting Images from Yahoo

Another source of book images is the Yahoo! Shopping site (http://shopping.yahoo.com/), which can be restricted to only search its books 'department'.

I joined Yahoo's shopping Web services at http://developer.yahoo.com/shopping/ by clicking on the "Get an App ID" button. After entering my details, an Application ID was generated.

Yahoo! Shopping is supported by several APIs, focusing on products, merchants, and a buyer's catalog. I utilized the Product Search API to find books, which is explained at http://developer.yahoo.com/shopping/V3/productSearch.html; there are details on request parameters, response elements, and example URLs.

Yahoo's Java developer center is at http://developer.yahoo.com/java/, and includes 'how to' articles and code samples. An example fairly close to mine is http://developer.yahoo.com/java/samples/YahooWebServiceParseResults.java which carries out a general web search (not a product search), and examines the XML response with XPath.

The Product Search API can return its results as XML or JSON data, so I'll use XML first, and then repeat the task with JSON in the next section so I can compare the two. The coding is fairly similar in both cases: a single BSRPE search retrieves a GridImage for a matching book cover.

4.1. Yahoo Product Search using XML and XPath

The main() method of SearchYahoo.java initializes the XPath namespace, carries out the book search in bookTitleSearch(), and saves the cover image.

```
// global
private static XPath xpath;

public static void main(String args[])
{
  if (args.length == 0) {
    System.err.println("Supply a title");
    System.exit(1);
  }

  xpath = createXPath();
  String imUrlStr = bookTitleSearch(args[0]);
  WebUtils.downloadImage(imUrlStr);
}  // end of main()
```

As in previous search code, I didn't write createXPath() until I'd got back a response from bookTitleSearch() which queries Yahoo! Shopping for a book with the specified title, and returns an image URL.

```
private static String bookTitleSearch(String title)
{
  System.out.println("Searching Yahoo for \"" + title + "\"");
  try {
    String ueTitle = URLEncoder.encode(title, "UTF-8");

    String urlStr =
"http://shopping.yahooapis.com/ShoppingService/V3/productSearch?" +
            "appid=" + APP_ID + "&" +
            "department=56&" +       // limit search to book dept
            "query=" + ueTitle + "&" +
            "results=5" ;            // at most 5 results

    Document doc = WebUtils.parse( WebUtils.webGet(urlStr) );
    WebUtils.saveDoc(doc, "temp.xml");
    return getPictureURL(doc);
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of bookTitleSearch()
```

The book department code is 56, which I found by looking through a list at http://developer.yahoo.com/shopping/departments.html.

The XML data is parsed into a DOM document, and saved to a file, so I could use the information to write getPictureURL() and createXPath().

A search for "gunpowder plot", results in the following being saved:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<ProductSearch xmlns="urn:yahoo:prods"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:yahoo:prods
http://shopping.yahooapis.com/shoppingservice/v3/productsearch.xsd">
<Products firstResultPosition="1"
          totalResultsAvailable="38"
          totalResultsReturned="5">

<Product type="Catalog">
  <Catalog ID="3004752223">
    <ProductName>&lt;b&gt;Gunpowder&lt;/b&gt;
                &lt;b&gt;Plot&lt;/b&gt;:</ProductName>
    <GridImage>
      <Url>http://a367.yahoofs.com/shopping/3096215
                /simg_t_t0758215975gif110?rm_____DJ4LBfgCu</Url>
      <Height>110</Height>
      <Width>68</Width>
    </GridImage>
    <Department ID="56">
      <Name>Books</Name>
    </Department>
    // other Product elements
```



```
  </Catalog>
</Product>

<Product type="Catalog">
  <Catalog ID="3005640541">
    <ProductName>The Enigma of &lt;b&gt;Gunpowder&lt;/b&gt;
     &lt;b&gt;Plot&lt;/b&gt;, 1605: The Third Solution</ProductName>
    <GridImage>
      <Url>http://a367.yahoofs.com/shopping/
            3145909/simg_t_t1846820928gif110?rm_____DW7e2Zmr5</Url>
      <Height>110</Height>
      <Width>72</Width>
    </GridImage>
    <Description>Author: Francis, S. J. Edwards. </Description>
    <Department ID="56">
      <Name>Books</Name>
    </Department>
    // other Product elements
  </Catalog>
</Product>

       :  // more Products
</Products>
</ProductSearch>
```

The matching products are represented by Product elements, which contain numerous sub-elements, including ProductName and GridImage values.

The "xmlns" attribute at the head of the file specifies the XML namespace, which I use in createXPath():

```
private static XPath createXPath()
{
  XPath xpath = XPathFactory.newInstance().newXPath();
  NamespaceContextImpl ns =
         new NamespaceContextImpl("yah", "urn:yahoo:prods");
  xpath.setNamespaceContext(ns);
  return xpath;
}  // end of createXPath()
```

The short name defined by createXPath() is "yah".

getPictureURL() prints the matching products' titles and GridImage URLs, and returns the first of those URLs.

```
private static String getPictureURL(Document doc)
{
  try {
    String totalStr =   // get @totalResultsAvailable attribute
        (String) xpath.evaluate(
                        "//yah:Products/@totalResultsAvailable",
                               doc, XPathConstants.STRING);
    System.out.println("Total no. of products matches: " + totalStr);

    NodeList nodes = doc.getElementsByTagName("Product"); //all prods
    String firstURL = null;
    for (int i = 0; i < nodes.getLength(); i++) {
      System.out.print( (i+1) + ". ");
```

```
      Element elem = (Element) nodes.item(i);
      System.out.println("Title: " +
                  WebUtils.getElemText(elem, "ProductName"));
      String urlStr = getGridImageURL(elem);
      System.out.println("   URL: " + urlStr);
      if (firstURL == null)  // store first URL that isn't null
        firstURL = urlStr;
    }
    return firstURL;
  }
  catch(Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of getPictureURL()
```

The XPath expression that extracts the total number of matches is a tad more complicated than usual since the "totalResultsAvailable" value is an attribute of the Products tag rather than an element. XPath denotes an attribute with a "@" prefix:

```
"//yah:Products/@totalResultsAvailable"
```

The product titles are printed by iterating over the Product elements, but reporting the GridImage URLs is a little more tricky since a product may have more than one image, or even none. This is handled by getGridImageURL():

```
private static String getGridImageURL(Element elem)
{
  NodeList nodeList = elem.getElementsByTagName("GridImage");
  if (nodeList.getLength() > 0) {
   Element el = (Element) nodeList.item(0);
   return WebUtils.getElemText(el,"Url"); // get "Url" text
  }
  return null;
}  // end of getGridImageURL()
```

All the GridImage elements are collected, and the first one is returned if possible.

The output generated by getGridImageURL() for a "gunpowder plot" search is:

```
Total no. of products matches: 38
1. Title: <b>Gunpowder</b> <b>Plot</b>:
   URL:
http://a367.yahoofs.com/shopping/3096215/simg_t_t0758215975gif110?rm_
____DJ4LbfgCu

2. Title: The Enigma of <b>Gunpowder</b> <b>Plot</b>, 1605: The Third
Solution
   URL:
http://a367.yahoofs.com/shopping/3145909/simg_t_t1846820928gif110?rm_
____DW7e2Zmr5

3. Title: BEN JONSON, "VOLPONE" AND THE <b>GUNPOWDER</b> <b>PLOT</b>:
   URL:
http://a367.yahoofs.com/shopping/3108404/simg_t_t052187954xgif110?rm_
____D60xm18.G

4. Title: The Condition of Catholics Under James I: Father Gerard's
```

```
Narrative of the <b>Gunpowder</b> <b>Plot</b> (1871)
   URL:
http://a367.yahoofs.com/shopping/3106670/simg_t_t054874257xgif110?rm_
____DsI38nLNC

5. Title: Mammoth Book of Jacobean Whodunnits: <b>Gunpowder</b>,
Treason and <b>
Plot</b>: 25 Tales of Murder Mystery in the 17th Century
   URL:
http://a367.yahoofs.com/shopping/3075667/simg_t_t0786717300gif110?rm_
____DuniiQcPI
```

getGridImageURL() returns the first non-null image URL string, which for the example above is http://a367.yahoofs.com/shopping/3096215/simg_t_t0758215975gif110?rm_____DJ4LBfgCu.

4.2. Yahoo Product Search using JSON and JXPath

I'll now reimplment the Yahoo! shopping example so it processes a JSON response instead of XML. This means a change to the parsing and examination stages of the BSRPE search. I'll need a more feature-rich JSON library than org.json which I used when searching Google. The principle drawback of org.json is that it doesn't build a tree-like structure, which is required for navigation tools like XPath.

The Jackson Java JSON-processor (http://jackson.codehaus.org/) supports tree-building, and also has the ability to convert a JSON data structure into Java objects. A Jackson tutorial can be found at http://jackson.codehaus.org/Tutorial, and its online API documentation is at http://jackson.codehaus.org/1.0.0/javadoc/index.html (for v1.0.0).

Jackson is only part of my solution – I also want a JSON-like XPath library. Unfortunately, I couldn't find a Java package that navigates JSON data, but the org.apache.commons.jxpath library does support XPath expressions applied to graphs of Java objects (http://commons.apache.org/jxpath/). The JXPath user guide contains several examples (http://commons.apache.org/jxpath/users-guide.html), and its API documents start at http://commons.apache.org/jxpath/apidocs/index.html.

I combine Jackson and JXPath as shown in Figure 2.

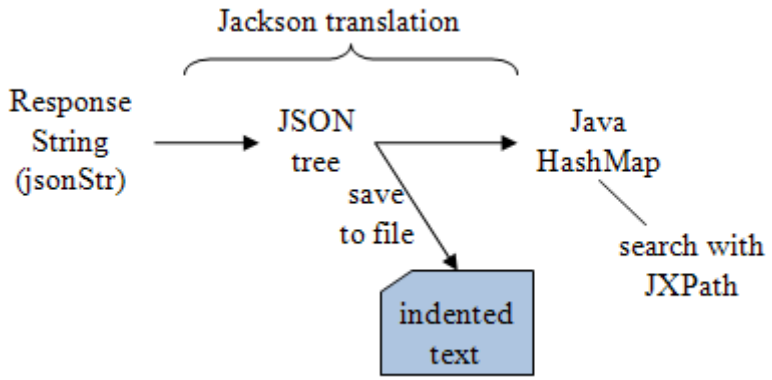

Figure 2. Using Jackson and JXPath Together.

Yahoo! Shopping returns a response string (jsonStr) which Jackson translates into a JSON tree structure. This is saved as nicely formatted text, and also converted into a Java HashMap which can be navigated by JXPath.

I downloaded the core and mapper Jackson libraries, version 1.0.0, from http://jackson.codehaus.org/, and version 1.3 of JXPath from http://commons.apache.org/jxpath/. After unzipping, I had three JAR files to add to my application directory (jackson-core-lgpl-1.0.0.jar, jackson-mapper-lgpl-1.0.0.jar, and commons-jxpath-1.3.jar).

SearchJYahoo's main() carries out the book search in bookTitleSearch(), and then saves the cover image:

```
public static void main(String args[])
{
  if (args.length == 0) {
    System.err.println("Supply a title");
    System.exit(1);
  }
  String imUrlStr = bookTitleSearch(args[0]);
  WebUtils.downloadImage(imUrlStr);
}  // end of main()
```

bookTitleSearch() does the same tasks as previously – it queries Yahoo! Shopping for a book with the specified title, and returns an image URL.

```
private static String bookTitleSearch(String title)
{
  System.out.println("Searching Yahoo for \"" + title + "\"");
  try {
    String ueTitle = URLEncoder.encode(title, "UTF-8");
    String urlStr =
"http://shopping.yahooapis.com/ShoppingService/V3/productSearch?" +
                  "appid=" + APP_ID + "&" +
                  "department=56&" +     // book department
                  "query=" + ueTitle + "&" +
                  "results=5&" +
                  "output=json" ;  // JSON output
    String jsonStr = WebUtils.webGetString(urlStr);

    // convert response string to Java HashMap
    ObjectMapper mapper = new ObjectMapper();
    HashMap<String,Object> jsonMap =
        mapper.readValue(jsonStr, HashMap.class);

    saveJSON("temp.json", mapper, jsonStr);  // save to file

    return getPictureURL(jsonMap);
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of bookTitleSearch()
```

The only change to the transmitted URL is that the "output" argument value is now "json".

ObjectMapper is a versatile Jackson class which translates data into different forms. In this case, I use it to convert the response string into a Java HashMap (after being converted to a JSON structure).

saveJSON() converts the response string into formatted text, and saves it to a file.

```
private static void saveJSON(String fnm, ObjectMapper mapper,
                                                  String jsonStr)
{
  SerializationConfig config = mapper.getSerializationConfig();
  config.set(SerializationConfig.Feature.INDENT_OUTPUT, true);
                    // pretty print output
  try {
    JsonNode rootNode = mapper.readValue(jsonStr, JsonNode.class);
    mapper.writeValue(new File(fnm), rootNode);
    System.out.println("Saved JSON struct to " + fnm);
  }
  catch(IOException e)
  {  System.out.println("Could not save JSON struct to " + fnm);  }
} // end of saveJSON()
```

ObjectMapper.readValue() translates the JSON string into a tree of JSON nodes.

For the search query "gunpowder plot", the following structure is saved:

```
{
  "Products" : {
    "Product" : [ {
      "__ATTRIBUTES" : {
        "type" : "Catalog"
      },
      "Catalog" : {
        "ProductName" : "<b>Gunpowder</b> <b>Plot</b>:",
        "GridImage" : {
          "Url" : "http://a367.yahoofs.com/shopping/3096215/
                    simg_t_t0758215975gif110?rm_____DJ4LBfgCu",
          "Height" : "110",
          "Width" : "68"
        },
        "Department" : {
          "__ATTRIBUTES" : {
            "ID" : "56"
          },
          "Name" : "Books"
        },
        "__ATTRIBUTES" : {
          "ID" : "3004752223"
        },
        // other Product elements
      }
    },
    // more Products
    ],
    "__ATTRIBUTES" : {
      "totalResultsAvailable" : "38",
      "totalResultsReturned" : "5",
      "firstResultPosition" : "1"
```

```
    }
  }
}
```

This structure contains the same information as the XML shown in the previous section, but as a series of maps and lists. XML attributes, such as the Product type and Catalog and Department IDs, are translated into maps using a "__ATTRIBUTES" key.

getPictureURL() prints the matching products' titles and GridImage URLs, and returns the first GridImage URL. It differs from the XML version of the method by using JXPath to traverse the Java HashMap.

```
private static String getPictureURL(HashMap<String,Object> jsonMap)
{
  try {
    JXPathContext context = JXPathContext.newContext(jsonMap);

    // get the @totalResultsAvailable attribute
    String totalStr = (String) context.getValue(
            "/Products/__ATTRIBUTES/totalResultsAvailable");
    System.out.println("Total no. of products matches: " + totalStr);

    Iterator prodIter = context.iterate("//Product");   // all prods
    JXPathContext prodContext;
    String firstURL = null;
    int i=0;
    while(prodIter.hasNext()){
      System.out.print( (i+1) + ". ");
      prodContext = JXPathContext.newContext( prodIter.next() );
                                          // next product
      System.out.println("Title: " +
                      prodContext.getValue("//ProductName") );
      String urlStr = null;
      try {  // Grid Image may not be present
        urlStr = (String) prodContext.getValue("//GridImage/Url");
      }
      catch(JXPathNotFoundException e) {}
      System.out.println("   URL: " + urlStr);
      if (firstURL == null)  // store first URL match that isn't null
        firstURL = urlStr;
      i++;
    }
    return firstURL;
  }
  catch (Exception e)
  { System.out.println(e); }

  return null;
} // end of getPictureURL()
```

JXPath expressions are evaluated relative to a context defined using a JXPathContext object. Initially this is the entire HashMap, and so the expression "/Products/__ATTRIBUTES/totalResultsAvailable" searches the map from the top looking for the "totalResultsAvailable" attribute beneath Products.

An iterator traverses over the Products subtree, and each product is assigned its own context:

```
prodContext = JXPathContext.newContext( prodIter.next() );
```

This product context is used for finding the product's name (//ProductName") and the GridImage URL ("//GridImage/Url").

As in the XML version, I must deal with the case when no GridImage is found, this time by using a try-catch block to catch JXPathNotFoundException.

The output generated by getPictureURL() for a "gunpowder plot" search is:

```
Total no. of products matches: 38
1. Title: <b>Gunpowder</b> <b>Plot</b>:
   URL:
http://a367.yahoofs.com/shopping/3096215/simg_t_t0758215975gif110?rm_
____DJ4LbfgCu

2. Title: The Enigma of <b>Gunpowder</b> <b>Plot</b>, 1605: The Third
Solution
   URL:
http://a367.yahoofs.com/shopping/3145909/simg_t_t1846820928gif110?rm_
____DW7e2Zmr5

3. Title: BEN JONSON, "VOLPONE" AND THE <b>GUNPOWDER</b> <b>PLOT</b>:
   URL:
http://a367.yahoofs.com/shopping/3108404/simg_t_t052187954xgif110?rm_
____D60xm18.G

4. Title: The Condition of Catholics Under James I: Father Gerard's
Narrative of the <b>Gunpowder</b> <b>Plot</b> (1871)
   URL:
http://a367.yahoofs.com/shopping/3106670/simg_t_t054874257xgif110?rm_
____DsI38nLNC

5. Title: Mammoth Book of Jacobean Whodunnits: <b>Gunpowder</b>,
Treason and <b>
Plot</b>: 25 Tales of Murder Mystery in the 17th Century
   URL:
http://a367.yahoofs.com/shopping/3075667/simg_t_t0786717300gif110?rm_
____DuniiQcPI
```

This is identical to the XML output, as you'd expect. getPictureURL() returns the image URL string, http://a367.yahoofs.com/shopping/3096215/simg_t_t0758215975gif110?rm_____DJ4LbfgCu, which is downloaded in main().

A comparison of the XML and JSON versions of the Yahoo search application show them to be functionally the same, although it requires a little more coding to use the Jackson and JXPath libraries. Also, almost all the necessary XML and XPath functionality is present in the JDK, while JSON and JXPath require third-party extensions to be installed.