

The ultimate mashup -- Web services and the semantic Web, Part 1: Use and combine Web services

Explore mashup concepts and build a simple mashup

Skill Level: Intermediate

[Nicholas Chase \(ibmquestions@nicholaschase.com\)](mailto:ibmquestions@nicholaschase.com)

Freelance writer
Backstop Media

22 Aug 2006

Updated 08 Mar 2007

As Web services grow in popularity, enterprising Web and application developers create new and innovative applications with their data. In addition to single-service applications, developers are creating mashups, applications that combine data from multiple services to create something new. This series chronicles the creation of the ultimate mashup, an application that not only stores data from different mashups but uses semantic technology to enable users to create their own mashups by swapping services, or even by picking and choosing data. It uses Java™ programming and a combination of servlets, JSP, software from the open source Jena project, and DB2's new native XML capabilities. In this part, Nicholas Chase introduces the concept of mashups, shows you how they work and how to build a simple version of one.

Section 1. Before you start

This tutorial is for developers who want to learn more about using and combining Web services from the XML point of view, and how to output that data to the Web. All services discussed in this tutorial are REST services, but the concepts are the

same for SOAP services. This tutorial assumes that you are familiar with Java™ programming, XML, Web development, and the basic concepts of Web services. If you need a refresher on these topics, please see [Resources](#) for more information.

About this series

It seems you can't turn around on the Web these days without running into a Web site that either offers access to its data through a Web-services-based API or uses data from another site obtained through a Web-services-based API. When you consider the advantage of using existing information in your own applications, that's probably not terribly surprising. It was also just a matter of time before someone started to combine the data from these disparate systems to create something entirely new. These applications, called mashups, are the latest rage on the Web, from community-based sites to specialized search sites to the ever-present mapping mashups.

Mashups are almost all useful, but they are all developed for a specific set of services. If one of those services changes, or if the preference for a specific service of a particular type changes, you'll have lots of work to do.

The purpose of this tutorial series is to create a mashup application so smart that users can literally add and remove services at will, and the system will know what to do with them. The series progresses as follows:

Here in Part 1, I introduce the concept of mashups, showing you how they work and building a simple version of one. You also discover serious performance problems involved in making potentially dozens of Web calls.

In Part 2, you solve some of that problem by using DB2's new pureXML capabilities to build an XML cache, which saves the results of previous requests and also enables you to retrieve specific information.

Ultimately, you will need to use *ontologies*, or vocabularies that define concepts and their relationships, so in Part 3 you start that process by learning about RDF and RDFs, two key ingredients in the Web Ontology Language (OWL), which I discuss in Part 4. In Part 5, you take the ontologies created in Part 4 and use them to enable users to change out information sources.

In Part 6, the fun increases. At this point, you have a working application and the framework in place so that the system can use semantic reasoning to understand the services at its disposal. In this part, you give the user control, enabling him or her to map new services into the ontology and to pick and choose the data that is used for a custom mashup.

About this tutorial

Over the course of this tutorial, you will learn how to create an application that retrieves and displays Web service information from a generic point of view; in other words, adding new services is practically a matter of configuration rather than programming.

This tutorial teaches you to:

- Request data from a REST Web service
- Easily serialize XML data for output or display
- Create an XML template, and then replace pre-determined elements and attributes with dynamic data
- Add XML nodes from multiple documents into a single output document
- Create a system that displays multiple Web services
- Create a system in which one Web service can provide information based on the output of a second Web service

This tutorial uses the Java language, but the concepts are the same for any programming language or operating system.

Prerequisites

To follow along with the code in this tutorial, you will need to have the following software installed and tested.

- [Apache Tomcat](#) or other servlet engine: This tutorial assumes that you will build Web applications using servlets, so you'll need a servlet engine such as Apache Tomcat. If you choose to build the application using another environment, just make sure you have the appropriate software on hand. Download [apache-tomcat-5.5.17.zip](#) and install into a directory with no spaces in the directory name.
- Java: Apache Tomcat 5.5, with which this tutorial is built, requires Java 1.5 or higher. Download the [J2SE SDK](#).

- To make things easier, you can use an IDE such as Eclipse or IBM® Rational™ Web Developer for your development. You can download Eclipse at Eclipse.org, download a trial version of [Rational Web Developer](#), or use your favorite development environment. You won't do anything fancy as far as compilation and deployment are concerned.

In Parts 2, 4, 5, and 6 of the series, you will also need:

- **IBM® DB2® 9 (formerly known as "Viper")**: This relational database also includes significant XML capabilities, which you'll need for this tutorial. You can download a trial version of DB2 9: [DB2 Enterprise 9](#) or [DB2 Express-C 9](#), a no-charge version of DB2 Express 9 data server.
-

Section 2. Overview

Before you dive into the mechanics of making this work, take a look at what it is you're trying to do.

Why Web services?

Depending on how loosely you define them, Web services have been around in one form or another for a decade or more, but now you hear an awful lot about them, especially on the Web. Why? Well, it has to do with how the Web is expanding at the moment.

Web services are a way to send information back and forth between servers in different locations. In most cases, the information going back and forth is XML, which provides an operating system and programming-language independent set of data. In other words, a server running a Perl application on Linux can send information over the Internet to a server running Java on Windows, and both of them will be able to understand it.

The reason this is important is that the generation whose mantra was "Information wants to be free" has moved from unemployed college students working away in the middle of the night to the movers and shakers in the information industry. It has now gotten to the point that if your Web application *doesn't* have a Web services-based API that enables others to use your data, you're considered backward, and perhaps a trifle suspect. Sometimes it seems that one of the worst things you can say about a Web site or application is "It's terrific, but it's a walled garden."

At the very least, an application isn't be considered cool unless other people can use it from some sort of API.

SOAP versus REST versus XML-RPC

Just because you have an API, doesn't mean that it's like everybody else's API. A wide variety of options exist, and no particular standard has been chosen, although popularity of one type of API or another waxes and wanes with popular sentiment (and developer choices) regularly. In many cases, an application provides access through more than one type of API.

At the moment, the most popular types of APIs are:

- **REST:** Although not the first to emerge, REST quickly gained popularity through its simplicity. All that's necessary to make a request is a URL; the XML is sent back in response to an HTTP request, just like a Web page.
- **SOAP:** The protocol that started it all, SOAP provides a much more robust way to make requests, but is more robust than most APIs need. It's also more complicated to use. That said, many toolkits exist to make the process easier, and tools such as Web Service Description Language-to-code converters keep this a popular option.
- **XML-RPC:** More complex than REST but simpler than SOAP, XML-RPC is still a favorite for many programmers who prefer its Remote Procedure Call style.
- **JavaScript:** The newest trend in APIs is for applications to offer a free JavaScript library that is the only way to access data. This enables anyone to place the data on their Web pages easily, but limits integration with other services. In some cases, data from other services can be fed to the JavaScript API, but feeding data from the JavaScript API to other services can be difficult or impossible. That said, look for these restrictions to be called out as the "walled gardens" they are, and for access to lighten up somewhat.

Now, with the exception of some JavaScript APIs, all of these APIs are based on the concept of sending XML in response to a request, which means that if you build your application on the idea of receiving XML, you'll ultimately be able to use just about anything.

To make things simple for now, though, let's stick with REST, though the concepts apply to any of the first three options.

What's a mashup?

The purpose behind having all of this information is for others to use it in some way. In some cases, that doesn't necessarily mean using it in the way you expect. For example, Amazon Light contains listings of products sold by Amazon.com (see [Resources](#) for a link), but it combines them with data from Google, Yahoo!, Del.icio.us, Blogger, and even DropCash. ActorTracker (see [Resources](#) for a link) combines Amazon, eBay, and other movie data. ChicagoCrime.org puts publicly available statistics on a Google map.

You can even keep up with new mashups by reading (or even subscribing to) MashupFeed.com, which lists new mashups by API and by popularity. It also has a very long list of APIs available for integration into new mashups, in case you want to make one of your own.

Basically, a mashup is a combination of information from more than one source, mixed up in such a way that you create something new, or at least useful.

The overall goal

One thing that all of these mashups have in common is that they were specifically designed for a purpose, with various choices -- such as what service to use, and how to access the data -- hard coded into the application. Your goal is to change that. You want to create an application that enables the user to make his or her own choices, to, say, switch out Google for Yahoo!, or Barnes and Noble for Amazon.com.

You also want the user to be able to create something entirely new without having to do any programming.

To make that happen, you need to do several things:

1. Create a flexible structure for your mashup that enables services to be changed and changed out easily.
2. Create an efficient storage system that enables you to easily cache and remix data.
3. Create an ontology that enables the application to understand the data it deals with. In other words, the application understands what a price or a location is, at least in relation to other, related data.
4. Find a way to programmatically use that ontology.
5. Give the user a way to interact with the data to create new mashups.

You'll be doing all of those things in the course of this series.

First steps to do right now

In this tutorial, to start that process, you will create a small mashup that enables some of these features.

First, you'll display basic information returned by a REST service. Then, you'll move on to create an XML-based template that enables you to control what each REST result will ultimately look like.

From there, you'll make it easy to extend the system to display information from multiple services. For example, you'll see results from both Yahoo! and Flickr.

Finally, you'll enable the system to perform the more complex mashup task of displaying information from one API based on the information returned by another API. For example, you'll feed URLs returned by Yahoo! to the Technorati Web service to see how many blog entries refer to each URL.

Let's get started.

Section 3. Extracting information from an XML feed

Now that you've set up everything, you can begin to create the actual mashup. You'll start by simply extracting information from an XML file residing on the local hard drive.

The sample data

I start with an XMLTV file already sitting on my hard drive. This data details television channels and programs, along with information and scheduling for each show. Because the complete file was more than 10MB, I extract just a small sample. You can see the structure in Listing 1.

Listing 1. The sample data

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tv source-info-url="http://labs.zap2it.com/"
    source-info-name="TMS Data Direct Service"
    generator-info-name="XMLTV"
    generator-info-url="http://www.xmltv.org/">

  <programme start="20060714043000 -0400"
    stop="20060714050000 -0400"
```

```

        channel="I21883.labs.zap2it.com">
        <title lang="en">Flintstones</title>
        <sub-title lang="en">The Gravelberry Pie
King</sub-title>
        <desc lang="en">Fred gets fired and enters
the
pie-baking business.</desc>
        <credits>
        <actor>Alan Reed</actor>
        <actor>Jean Vander Pyl</actor>
        <actor>Mel Blanc</actor>
        <actor>Gerry Johnson</actor>
        <actor>Don Messick</actor>
        </credits>
        <date>19651112</date>
        <category lang="en">Children</category>
        <category lang="en">Animated</category>
        <category lang="en">Series</category>
        <episode-num
system="dd_progid">EP001648.0149</episode-num>
        <episode-num
system="onscreen">65149</episode-num>
        <subtitles type="teletext" />
        <rating system="VCHIP">
        <value>TV-G</value>
        </rating>
        </programme>
        <programme start="20060714050000 -0400 "
        stop="20060714053000 -0400 "
        channel="I21883.labs.zap2it.com">
        <title lang="en">Dastardly &
Muttley</title>
        <sub-title lang="en">Muttley On the
Bounty</sub-title>
        <category lang="en">Children</category>
        <category lang="en">Animated</category>
        <category lang="en">Series</category>
        <episode-num
system="dd_progid">EP001168.0006</episode-num>
        <episode-num
system="onscreen">6901</episode-num>
        <rating system="VCHIP">
        <value>TV-G</value>
        </rating>
        </programme>
        <programme start="20060714053000 -0400 "
        stop="20060714060000 -0400 "
        channel="I21883.labs.zap2it.com">
        <title lang="en">Banana Splits</title>
        ...
        </programme>
</tv>

```

This is just a normal XML file, with a root element and a number of individual record elements. In this case the record elements are the programme elements. I want to loop through each of them and display specific information.

Parse the data

To start, parse the document into memory so you can work with it, as in Listing 2.

Listing 2. Parsing the document

```
import org.w3c.dom.Document;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.File;

public class SimpleClient {

    public static void main(String[] args) {

        try {

            DocumentBuilder builder =
                DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document document = builder.parse(
                new File("/SW/xmltv-0.5.44-win32/sample.xml"));

        } catch (Exception e){

            e.printStackTrace();

        }

    }

}
```

This application isn't particularly difficult. It simply creates a class that parses a file on the local operating system into a Document object you can then manipulate. Of course, in a production environment, you'd create more robust exception handling in this and later situations than I've done here, but this isn't a production environment, and isn't meant to be.

In the past, you would then work your way through that Document object looking for specific nodes. Fortunately, you don't have to do that anymore.

Introducing XPath

Every XML document has a structure, and previously to programmatically deal with one, you had to either work your way through the document in a tree-like fashion using one of the document object models, or use an event-based API such as SAX to find the information you want.

But that's always seemed kind of silly, especially for developers who were used to working with XSL transformations and their style sheets, which enabled developers to specify a node or group of nodes using an XPath expression. Consider, for example, the XPath expressions in Listing 3.

Listing 3. XPath expressions

```
/tv/programme
//rating
//programme/@start
//title[@lang='en']
```

They specify, in order, any `programme` child of the root `tv` element, any `rating` element anywhere in the document (the double slash (`//`) indicates any descendant of the current node, which in this case is the root element), the `start` attribute of any `programme` element, and any `title` element that has a `lang` attribute equal to `en`.

Fortunately, these days you don't have to resort to tricks such as creating an XSL transformation to use XPath expressions. In this application, you'll use them directly from your Java application to pull any information that you need.

Evaluating an XPath expression

Your goal is to loop through each of the `programme` elements in the document, so your first steps are to create an XPath expression that represents them and to evaluate that expression, as you can see in Listing 4.

Listing 4. Evaluating an XPath expression

```
...
import java.io.File;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPathConstants;
import org.w3c.dom.NodeList;

public class SimpleClient {

    public static void main(String[] args) {

        try {

            DocumentBuilder builder =
                DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document document = builder.parse(
                new File("/SW/xmltv-0.5.44-win32/sample.xml"));

            XPath xpath = XPathFactory.newInstance().newXPath();
            String expression = "//programme";

            NodeList recordNodes =
                (NodeList) xpath.evaluate(
                    expression, document, XPathConstants.NODESET);

            System.out.println("The result has "+
                recordNodes.getLength()+" nodes");

        } catch (Exception e){

            e.printStackTrace();

        }

    }
}
```

First, create the `XPath` class from an `XpathFactory`. Once you have the class, you can use the `evaluate()` method to extract a `NodeList` of relevant nodes by

feeding it the XPath expression that represents the nodes you want, the document in which to look for them, and the type of data you want returned.

This last parameter can be quite powerful. In this case, you return a set of nodes, but you might also tell the method that you want just a single node, or even a Boolean value that simply tells you whether or not the expression represents any actual nodes.

Once you get the data back, you can treat it as a `NodeList` object you've retrieved any other way. Here you can see that the code requests the number of nodes, just as you might do with a `NodeList` obtained with the DOM's `getChildNodes()` method.

If you execute the application, either from the command line or from within Eclipse or another IDE, you'll see output that tells you how many programme elements existed in the document, as in Listing 5.

Listing 5. Executing the application

```
The result has 12 nodes
```

Create the output document

You can simply run through each document and output individual pieces of data, but that locks you in to specific formats, display methods, and so on. Instead, you'll create an output document, an XML document to which you can add the information you want to output. You can then output that document to your chosen destination.

In Listing 6, start by creating the actual Document object.

Listing 6. Creating the output Document

```
...
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.xml.sax.InputSource;
import java.io.StringReader;

public class SimpleClient {

    public static void main(String[] args) {

        try {
            ...
            NodeList recordNodes = (NodeList) xpath.evaluate(
                expression, document, XPathConstants.NODESET);
            System.out.println("The result has "+
                recordNodes.getLength()+" nodes");

            Document hostDoc = builder.parse(
```

```

        new InputSource(new StringReader("<div />"));
        Node hostRoot = hostDoc.getDocumentElement();

    } catch (Exception e){
        e.printStackTrace();
    }
}
}

```

The first step is to create a new `Document` object, and to do that, you parse an XML document that provides a shell, of sorts -- in this case, a `div` element -- into which you can place the data. Rather than using an actual file, however, you can use a `StringReader` and an `InputSource` to make a `Document` object out of a `String`.

Once you have the `Document`, you can reference its root element in order to add data.

Transfer nodes from one document to the other

Now, as anyone who's tried to do it can tell you, you can't simply take a `Node` from one `Document` and add it to another `Document`. Well, not without getting a nasty runtime error, anyway. So, to safely transfer the information from the source document to the output document, you need to import it, as you see in Listing 7.

Listing 7. Transferring nodes from one document to another

```

...
    NodeList recordNodes = (NodeList) xpath.evaluate(
        expression, document, XPathConstants.NODESET);
    System.out.println("The result has "+
        recordNodes.getLength()+" nodes");

    Document hostDoc = builder.parse(
        new InputSource(new StringReader("<div />")));
    Node hostRoot = hostDoc.getDocumentElement();

    for (int i=0; i < recordNodes.getLength(); i++) {
        Node importedNode =
            hostDoc.importNode(recordNodes.item(i), true);
        hostRoot.appendChild(importedNode);
        hostRoot.appendChild(hostDoc.createTextNode("\n"));
    }

    } catch (Exception e){
        e.printStackTrace();
    }
}
}

```

For each `Node` in the `NodeList`, use the `hostDoc`'s `importNode()` method to

create a copy of the `Node` that belongs to `hostDoc`. The second parameter tells the application whether to do a *deep* copy. In other words, by specifying that value as `true`, you say that you want not only the `programme` element, but also all of its children and other descendants.

Once you have a copy that's imported into the `hostDoc`, you can append it to the `hostRoot`. Then, add a simple line feed text node to make the results easier to read.

Output the results

To output the host document, you'll use an old XSLT-related trick to serialize it to the command line, as in Listing 8.

Listing 8. Outputting the results

```
import java.io.StringReader;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

public class SimpleClient {
    ...
    for (int i=0; i < recordNodes.getLength(); i++) {
        Node importedNode =
            hostDoc.importNode(recordNodes.item(i), true);
        hostRoot.appendChild(importedNode);
        hostRoot.appendChild(hostDoc.createTextNode("\n"));
    }

    DOMSource source = new DOMSource(hostDoc);
    StreamResult result = new StreamResult(System.out);
    TransformerFactory transFactory =
        TransformerFactory.newInstance();
    Transformer transformer = transFactory.newTransformer();
    transformer.transform(source, result);

    } catch (Exception e){
        e.printStackTrace();
    }
}
```

First you create a `DOMSource` out of the `Document` you want to output. Then, create a `StreamResult` that represents the command line. From there, create a new `Transformer`, but don't feed it a style sheet because you want the `Document` to be unchanged. (That's one of the advantages of this output method; if you later decide to transform the data before outputting it, adding a stylesheet into the process is almost trivial.) Finally, transform the document and send the results to the command line.

The results

You can see the resulting output in Listing 9.

Listing 9. The result of outputting the document

```
The result has 12 nodes
<?xml version="1.0" encoding="UTF-8"?><div><programme
  channel="I21883.labs.zap2it.com" start="20060714043000 -0400"
  stop="20060714050000 -0400">
  <title lang="en">Flintstones</title>
  <sub-title lang="en">The Gravelberry Pie King</sub-title>
  <desc lang="en">Fred gets fired and enters the pie-baking
  business.</desc>
  <credits>
    <actor>Alan Reed</actor>
    <actor>Jean Vander Pyl</actor>
    <actor>Mel Blanc</actor>
    <actor>Gerry Johnson</actor>
    <actor>Don Messick</actor>
  </credits>
  <date>19651112</date>
  <category lang="en">Children</category>
  <category lang="en">Animated</category>
  <category lang="en">Series</category>
  <episode-num system="dd_progid">EP001648.0149</episode-num>
  <episode-num system="onscreen">65149</episode-num>
  <subtitles type="teletext"/>
  <rating system="VCHIP">
    <value>TV-G</value>
  </rating>
</programme>
<programme channel="I21883.labs.zap2it.com" start="20060714053000 -0400"
  stop="20060714053000 -0400">
  <title lang="en">Dastardly & Muttley</title>
  ...
</programme>
<programme channel="I21883.labs.zap2it.com" start="20060714053000 -0400"
  stop="20060714060000 -0400">
  <title lang="en">Banana Splits</title>
  ...
</programme>
...
</div>
```

Notice that you still have the original output statement, and that the data is essentially unchanged, except for the fact that it's now in a `div` element.

In the next section, you'll look at how to easily change the structure of the outputted data using XML-based templates.

Section 4. Making things generic using templates

So far you've looked at the mechanics of retrieving an XML document, finding individual records, and outputting them. That's all well and good, but in the long run, not much more useful than simply retrieving the document and displaying it directly. In this section, you start the process of massaging the data into a more desirable form using templates.

Using templates

The ultimate goal is to create an application that enables users to make choices about content to display, so the last thing you want to do is create a system that requires programming to change the appearance of data. To get around that problem, you can create templates into which the live data can be inserted.

For example, you can create a template such as the one in Listing 10.

Listing 10. A sample template

```
<p><b><value/></b>: <value/></p>
```

This template takes two values, which you can then replace with live data. For example, Listing 11 shows two different uses of this simple template.

Listing 11. Two different uses of the same template

```
                <p><b>The West Wing</b>: When a
uranium-bearing rig crashes in an Idaho tunnel, the White House
scrambles
to assess the potential crisis; election strategists mull dropping Vice
President Hoynes (Tim Matheson) from the Democratic ticket.</p>
                <p><b>One killed, one missing in Ohio storms; rare
twister
in New York state</b>: (Undated-AP) July 13, 2006 - At least
one person died after up to nine inches of rain brought flooding to
Indiana
and Ohio, while tornado north of New York City partly collapsed a
commercial
building and ripped the roof off a hotel.</p>
```

In the first case, this is television data and in the second, it's search results. The difference is their source documents and the XPath expressions that represent what was added. In the first case it was the `/tv/programme/title` and `/tv/programme/desc` values from the XMLTV feed. In the second it was the `/ResultSet/Result/Title` and `/ResultSet/Result/Summary` values from a Yahoo! search result. In this section, you'll look at using XPath to insert specific data into a template and then at outputting the resulting data.

Create the template document

To begin, create a `Document` object out of the template, as you can see in Listing 12.

Listing 12. Creating the template Document

```
...
    Document hostDoc = builder.parse(
        new InputSource(new StringReader("<div />"));
    Node hostRoot = hostDoc.getDocumentElement();

    for (int i=0; i < recordNodes.getLength(); i++) {
        String template =
"<p><b><value/></b>:<value/></p>";
        Document templateDoc = builder.parse(
            new InputSource(new StringReader(template)));

    }

    DOMSource source = new DOMSource(hostDoc);
    StreamResult result = new StreamResult(System.out);
...

```

Here you use the same technique used to create the host document. But instead of a single element, you seed the document with the actual template you want to use for each record in the results.

Find the nodes to replace

The next step is to use an XPath expression to locate all of the nodes in the template Document that are elements named `value`, as you can see in Listing 13.

Listing 13. Finding the nodes to replace

```
...
    for (int i=0; i < recordNodes.getLength(); i++) {
        String template =
"<p><b><value/></b>:<value/></p>";
        Document templateDoc = builder.parse(
            new InputSource(new StringReader(template)));

        expression = "//value";
        NodeList templateNodes =
            (NodeList) xpath.evaluate(
                expression, templateDoc, XPathConstants.NODESET);
        for (int j=0; j < templateNodes.getLength(); j++){

    }
}

```

```

    }
    DOMSource source = new DOMSource(hostDoc);
    ...

```

The XPath expression represents all value elements in the Document, so when you evaluate it against the `templateDoc` object, you get a `NodeList` that points to all of the value elements in that Document. You can then loop through each of those Nodes and take action.

Find the replacement data

The action that you want to take is to replace those elements with specific data, so the next step is to find that data. In Listing 14, you look for it using specific XPath expressions.

Listing 14. Finding the replacement data

```

...
    for (int i=0; i < recordNodes.getLength(); i++) {
        String template =
        "<p><b><value/></b>:<value/></p>";
        Document templateDoc = builder.parse(
            new InputSource(new StringReader(template)));

        String[] elementValues = {"title[@lang='en']",
            "desc[@lang='en']"};

        expression = "//value";
        NodeList templateNodes =
            (NodeList) xpath.evaluate(
                expression, templateDoc, XPathConstants.NODESET);
        for (int j=0; j < templateNodes.getLength(); j++){

            if ( (Boolean)xpath.evaluate(elementValues[j],
                recordNodes.item(i),
                XPathConstants.BOOLEAN)) {

                Node valueNode =(Node)xpath.evaluate(
                    elementValues[j],
                    recordNodes.item(i),
                    XPathConstants.NODE);

                String replacementValue = valueNode.getTextContent();
            }
        }
    }
    DOMSource source = new DOMSource(hostDoc);
    ...

```

You have two slots in the template, so you'll define two different XPath expressions

to fill them. For each record node, first check to see whether the record node (`recordNodes.item(i)`) contains the target expression (`elementValues[j]`). (The context node is `/tv/programme`, so the XPath expression is relative to that.)

If the node exists, you can then use the `evaluate()` method, along with the `XPathConstants.NODE` constant, to retrieve the node itself. You can then extract the actual text of the Node so you can add it to the template.

Add the replacements to the template

Now that you have the replacement text, you can add it to the template, as in Listing 15.

Listing 15. Adding replacement values to the template

```
...
    expression = "//value";
    NodeList templateNodes = (NodeList) xpath.evaluate(
        expression, templateDoc, XPathConstants.NODESET);
    for (int j=0; j < templateNodes.getLength(); j++){

        Node thisTemplateNode = templateNodes.item(j);
        Node parentNode = thisTemplateNode.getParentNode();

        if ( (Boolean)xpath.evaluate(elementValues[j],
            recordNodes.item(i), XPathConstants.BOOLEAN)) {

            Node valueNode = (Node)xpath.evaluate(
                elementValues[j], recordNodes.item(i),
                XPathConstants.NODE);
            String replacementValue = valueNode.getTextContent();

            Node replacementNode =
                templateDoc.createTextNode(replacementValue);
            parentNode.replaceChild(replacementNode,
                thisTemplateNode);

        }
    }
...

```

For each template node to be replaced, first obtain a reference to that specific node. Then get a reference to that node's parent, which will ultimately enable you to replace the template node.

Once you determine that a replacement value exists and retrieve that value, you first create a new text node within the `templateDoc`. That node contains the replacement value. You can then use the `replaceChild()` method to replace the original value element (`thisTemplateNode`) with the new replacement text node. The result is a template that contains the new information.

Add the new template to the output document

Once you have the completed template, add it to the output document, just as you added the raw data in the previous section, as in Listing 16.

Listing 16. Adding the completed template to the output document

```

...
        for (int j=0; j < templateNodes.getLength(); j++){
...
            parentNode.replaceChild(replacementNode,
                                   thisTemplateName);

        }

    }
    Node importedNode = hostDoc.importNode(
        templateDoc.getDocumentElement(), true);
    hostRoot.appendChild(importedNode);
    hostRoot.appendChild(hostDoc.createTextNode("\n"));
}

DOMSource source = new DOMSource(hostDoc);
StreamResult result = new StreamResult(System.out);
...

```

For each record, process the template and then add it to the host document just as you previously added the raw document. Once that process is complete, output the host document as usual. For the abbreviated TV listings document, the results look like Listing 17.

Listing 17. The final results

```

<?xml version="1.0"
  encoding="UTF-8"?><div><p><b>Flintstones</b>
: Fred gets fired and enters the pie-baking business.</p>
<p><b>Dastardly & Muttley</b>:<value/></p>
<p><b>Banana Splits</b>:<value/></p>
<p><b>Project Runway</b>:<value/></p>
<p><b>Project Runway</b>:<value/></p>
<p><b>Celebrity Poker Showdown</b>: Angie Dickinson, Penn
  Jillette, Jeff Gordon, Kathy Griffin and Ron Livingston compete on behalf
  of their favorite charities.</p>
<p><b>Celebrity Poker Showdown</b>: Willie Garson, Jennie
  Garth, Richard Kind, Dave Navarro and Jerry O'Connell compete on behalf of
  their favorite charities.</p>
<p><b>Kathy Griffin: My Life on the D-List</b>: Kathy's
  trip to entertain the troops is winding down.</p>
<p><b>Kathy Griffin: My Life on the D-List</b>: Kathy has
  problems with her new puppy; the Learning Annex hires Kathy to teach a
  class.</p>
<p><b>Project Runway</b>:<value/></p>
<p><b>Project Runway</b>:<value/></p>
<p><b>The West Wing</b>: When a uranium-bearing rig
  crashes in an Idaho tunnel, the White House scrambles to assess the
  potential crisis; election strategists mull dropping Vice President

```

```
Hoynes (Tim Matheson) from the Democratic ticket.</p>
</div>
```

Notice that the template still contains value elements in cases in which the data wasn't found in the source document. Browsers will ignore them, but you also have the option to put them into separate namespaces, or even to remove them altogether before output. (For example, that might be a good application for XSL transformations.)

Section 5. Even more generic: Multiple services

At this point, you can essentially control the output of the application through a series of strings, whether they represent the template or XPath expressions to fill the template. In this section, you'll genericize the application even further, enabling you to automatically add services to and remove services from the output by controlling the contents of an array.

Creating the service array

The first step is to create an array of Service objects that include information necessary for processing them. In the final application, you'll create this array dynamically, perhaps pulling service information from a database, but for now you can arbitrarily create the Service class and its members, as seen in Listing 18.

Listing 18. Creating the service array

```
public class Service {
    String name = "";
    String baseURL = "";
    String template = "";
    String[] elementValues = {"title", "desc"};
    String recordExp = "";

    public static Service[] getServices(){
        Service[] services = new Service[2];

        Service thisService = new Service();

        thisService.name = "Yahoo! Search";
        thisService.baseURL =
"http://api.search.yahoo.com/NewsSearchService/V1/newsSearch?appid=ma
shupid&type=all&query=New+York";
        thisService.template =
"<p><b><value/></b>:<value/></p>";
        thisService.elementValues[0] = "Title";
        thisService.elementValues[1] = "Summary";
    }
}
```

```

        thisService.recordExp =
"/ResultSet/Result";
        services[0] = thisService;

        Service thisService2 = new Service();

        thisService2.name = "YouTube";
        thisService2.baseURL =
"http://www.youtube.com/api2_rest?method=youtube.videos.list_by_tag&d
ev_id=s2gNEM-7qoU&tag=New+York";
        thisService2.template =
"<p><value/>:<value/></p>";
        thisService2.elementValues[0] = "title";
        thisService2.elementValues[1] =
"description";
        thisService2.recordExp =
"/ut_response/video_list/video";
        services[1] = thisService2;

        return services;
    }
}

```

This is not a paragon of Java design, but it does what it needs to do. It creates a class that includes as attributes the values that you previously hard coded into the application, such as the template and the XPath expression that represents an individual record. It also includes attributes that are specific to each service, such as the URL and name.

The class includes a static method, `getServices()`, which arbitrarily creates two `Service` objects and adds them to an array.

You'll use that array within the application.

Looping through each service

To add each service is fairly straightforward, as you can see in Listing 19.

Listing 19. Adding multiple services

```

...
public class SimpleClient {
    public static void main(String[] args) {
        try {
            Service[] svcs = Service.getServices();
            DocumentBuilder builder =
                DocumentBuilderFactory.newInstance().newDocumentBuilder();

            Document hostDoc = builder.parse(
                new InputSource(new StringReader("<div />")));
            Node hostRoot = hostDoc.getDocumentElement();

            for (int s=0; s < svcs.length; s++){

```

```

Service svc = svcs[s];
Document document = builder.parse(svc.baseURL);

XPath xpath = XPathFactory.newInstance().newXPath();
String expression = svc.recordExp;
NodeList recordNodes = (NodeList) xpath.evaluate(
    expression, document, XPathConstants.NODESET);

Node titleNode = hostDoc.createTextNode(svc.name);
Element titleElement = hostDoc.createElement("h1");
titleElement.appendChild(titleNode);

hostRoot.appendChild(titleElement);

for (int i=0; i < recordNodes.getLength(); i++) {

    String template = svc.template;
    Document templateDoc = builder.parse(
        new InputSource(new StringReader(template)));

    String[] elementValues = svc.elementValues;

    expression = "//value";
    NodeList templateNodes = (NodeList) xpath.evaluate(
        expression,
        templateDoc,
        XPathConstants.NODESET);
    for (int j=0; j < templateNodes.getLength(); j++){

        Node thisTemplateNode = templateNodes.item(j);
        Node parentNode = thisTemplateNode.getParentNode();

        if ( (Boolean)xpath.evaluate(elementValues[j],
            recordNodes.item(i),
            XPathConstants.BOOLEAN)) {

            Node valueNode = (Node)xpath.evaluate(
                elementValues[j],
                recordNodes.item(i),
                XPathConstants.NODE);
            String replacementValue =
                valueNode.getTextContent();
            Node replacementNode =
                templateDoc.createTextNode(replacementValue);
            parentNode.replaceChild(replacementNode,
                thisTemplateNode);

        }

    }
    Node importedNode = hostDoc.importNode(
        templateDoc.getDocumentElement(), true);
    hostRoot.appendChild(importedNode);
    hostRoot.appendChild(hostDoc.createTextNode("\n"));
}

}

DOMSource source = new DOMSource(hostDoc);
StreamResult result = new StreamResult(System.out);
TransformerFactory transFactory =
    TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
transformer.transform(source, result);

} catch (Exception e){

```

```

        e.printStackTrace();
    }
}

```

First, get the array of services. Then loop through the array and treat each service just as you did before, with the values from the Service object replacing the hard-coded values previously used. The code also adds a heading with the name of each service to the output document.

The result is a document that includes all of the output of both services, as seen in Listing 20.

Listing 20. The output of both services

```

<?xml version="1.0" encoding="UTF-8"?><div><h1>Yahoo!
Search</h1><p>
<b>Severe Weather Hits Ohio, New York</b>:Severe
weather hit New York
and the Ohio region. Up to 9 inches of rain brought flooding to
Indiana and Ohio on Wednesday, killing a woman. North of New York
City, a tornado partly collapsed a commercial building and ripped the
roof off a hotel.</p>
<p><b>Musei: a New York ticket di 20 dlr</b>:(ANSA) -
NEW YORK, 13 LUG
- Il Metropolitan Museum di New York ha alzato il prezzo suggerito per
l'ingresso di un adulto da 15 a 20 dollari.</p>
<p><b>One killed, one missing in Ohio storms; rare twister in
New York ...
Europe through Vodafone Group in hopes of resolving technical problems
with a previous model</p>
<h1>YouTube</h1><p>New York Dolls-Jet Boy live:NEW
YORK DOLLS =D YAY</p>
<p>Rolling Stones-Dead Flowers live:Rolling stones perform dead
...
<p>maurice:New york</p>
</div>

```

It's a little messy, but all of the points of interest are here, including the main `div` element, the service names, and the fact that each service has a slightly different template.

Section 6. Moving the application to the Web

At this point the application is complete, but it still needs to be transferred to the Web. To do that, you need to add the code to a servlet. The servlet has two main methods: `doGet()` and `doPost()`. When the browser requests the servlet using the `GET` method, it displays a simple form that enables the user to enter a search

query. When the user submits the form, it submits it through the POST method, and the servlet processes the data as before.

Creating the servlet

First, create the actual servlet, as in Listing 21.

Listing 21. Creating the actual servlet

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MashupClientServlet
    extends
    javax.servlet.http.HttpServlet
    implements
    javax.servlet.Servlet {

    public MashupClientServlet() {
        super();
    }

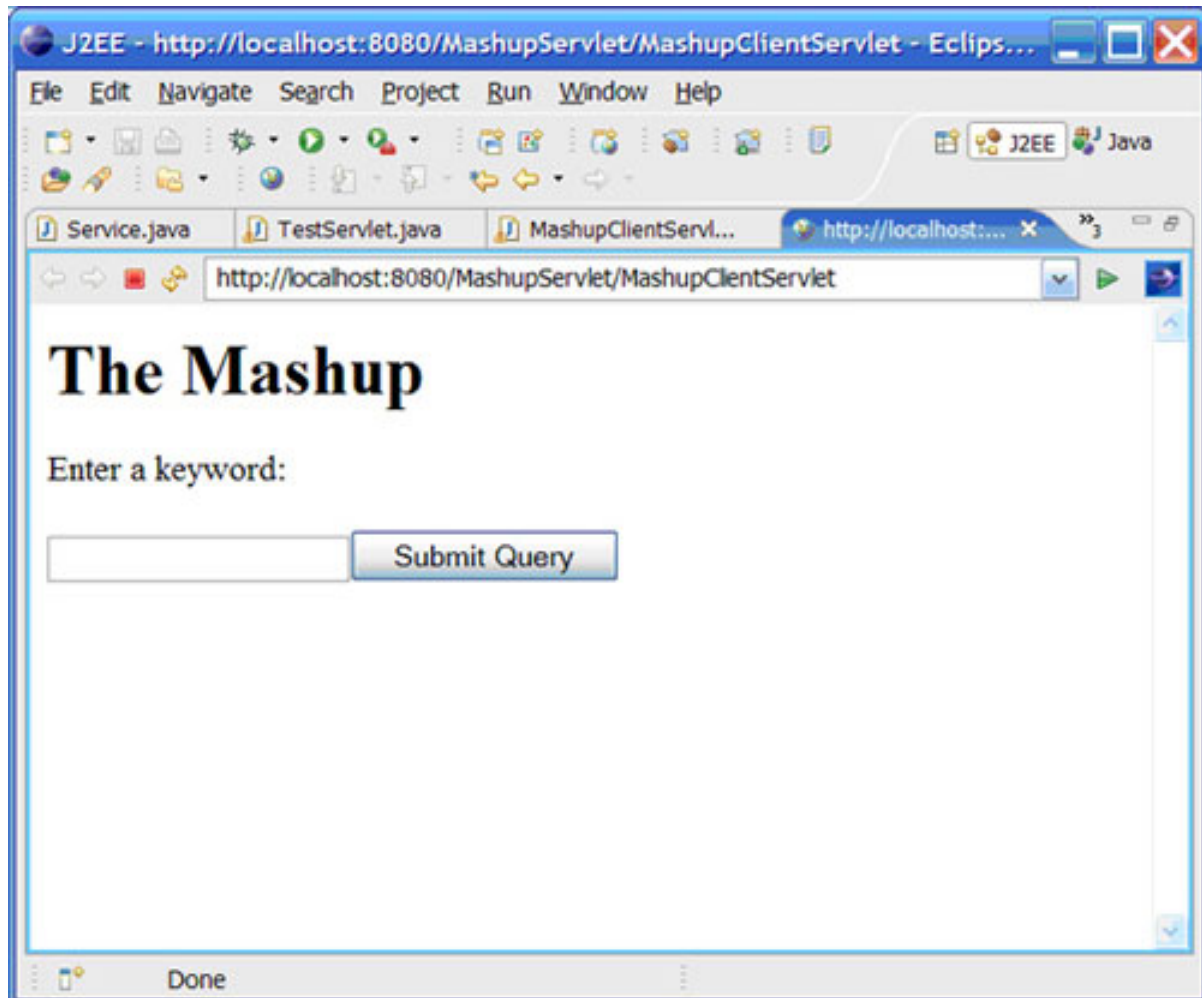
    protected void doGet(HttpServletRequest
        request,
                           HttpServletResponse
        response)
        throws
        ServletException, IOException {
        response.getWriter().print("<html>");
        response.getWriter().print("<head></head>");
        response.getWriter().print("<body>");

        response.getWriter().print("<h1>The
Mashup</h1>");
        response.getWriter().print("<p>Enter a
keyword:</p>");
        response.getWriter().print(
            "<form action='MashupClientServlet
method='post'>");
        response.getWriter().print(
            "<input type='text'
name='query' />");
        response.getWriter().print("<input
type='submit' />");
        response.getWriter().print("</form>");
        response.getWriter().print("</body>");
        response.getWriter().print("</html>");
    }

    protected void doPost(HttpServletRequest
        request,
                           HttpServletResponse
        response)
        throws
        ServletException, IOException {
        // Actual processing goes here
    }
}
```

This is just a standard servlet. If you save it and deploy it to your server, and then call it from the browser, you will see something similar to Figure 1.

Figure 1. The doGet() method



Adjusting the Service information

Because you now pull keywords from the form, you need to make a slight change so that the keywords are no longer embedded in the URL that represents the REST requests, as show in Listing 22.

Listing 22. Adjusting the Service information

```
...
    thisService.name = "Yahoo! News Search";
    thisService.baseURL =
"http://api.search.yahoo.com/NewsSearchService/V1/newsSearch?appid=ma
shupid&type=all&query=";
    thisService.template =
```

```

" <p><b><value/></b>:<value/></p>" ;
...
    thisService2.name = "YouTube" ;
    thisService2.baseURL =
"http://www.youtube.com/api2_rest?method=youtube.videos.list_by_tag&de
v_id=s2gNEM-7qoU&tag=" ;
    thisService2.template =
" <p><value/>:<value/></p>" ;
...

```

Otherwise, the information is unchanged. Add the keywords back on before you call the service.

The complete servlet

Completing the servlet is a straightforward matter of transferring the functionality you've already built to the `doPost()` method, as you can see in Listing 23.

Listing 23. The full servlet

```

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;
import java.io.IOException;
import java.io.StringReader;

public class MashupClientServlet extends javax.servlet.http.HttpServlet
implements javax.servlet.Servlet {

    public MashupClientServlet() {
        super();
    }

    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        response.getWriter().print("<html>");
        ...
        response.getWriter().print("</html>");
    }

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        try {

```

```
String query = request.getParameter("query").toString();

Service[] svcs = Service.getServices();
DocumentBuilder builder =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();

Document hostDoc = builder.parse(
    new InputSource(new StringReader("<div />")));
Node hostRoot = hostDoc.getDocumentElement();

for (int s=0; s < svcs.length; s++){

    Service svc = svcs[s];
    Document document = builder.parse(svc.baseURL+query);

    XPath xpath = XPathFactory.newInstance().newXPath();
...

}

DOMSource source = new DOMSource(hostDoc);
StreamResult result = new StreamResult(response.getWriter());
TransformerFactory transFactory =
    TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
transformer.transform(source, result);

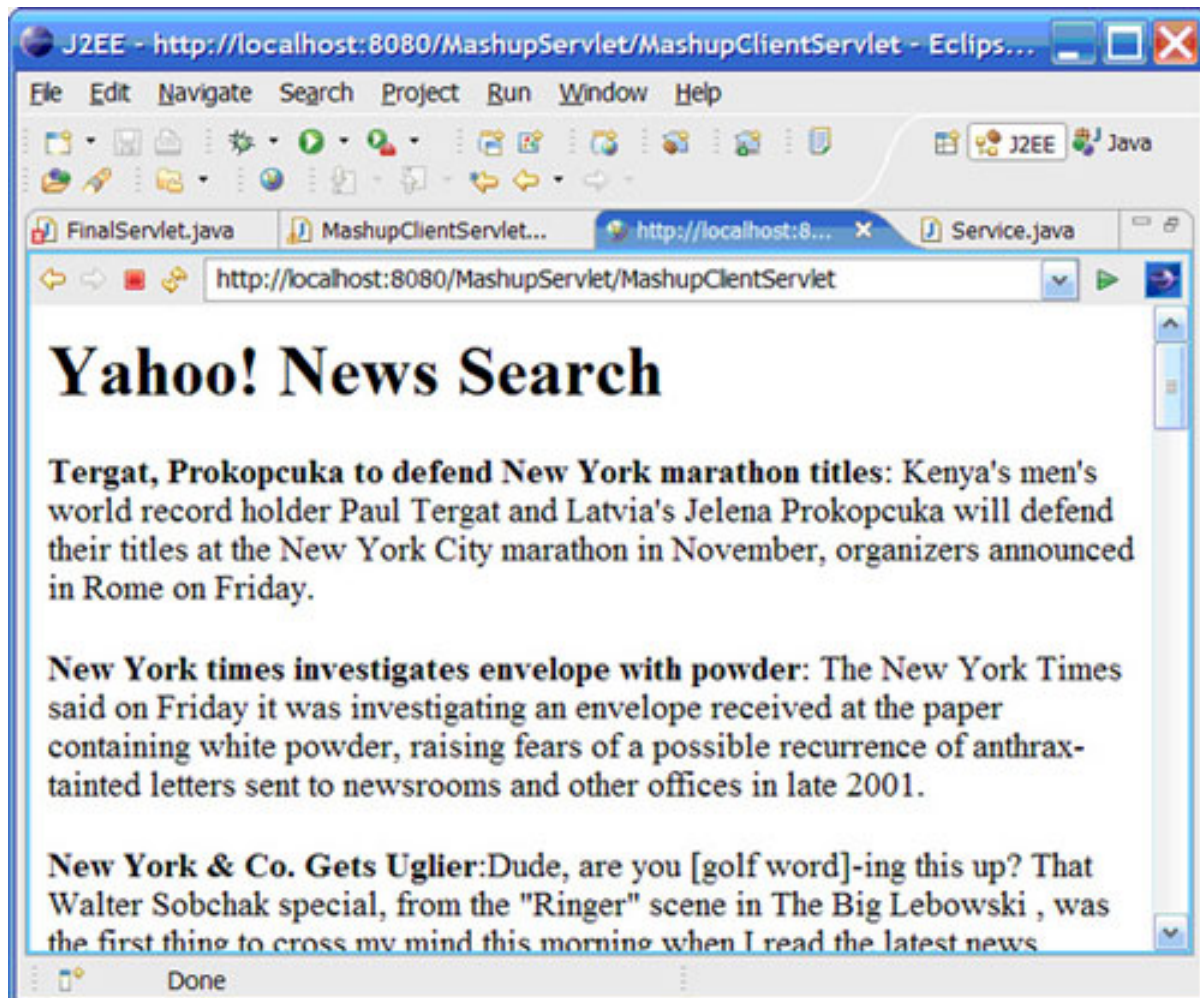
} catch (Exception e){
    e.printStackTrace();
}
}
```

All you really did here is to take the contents of the `main()` method of the standalone class and add it as the contents of the `doPost()` method, with three small changes. First, you retrieved the query entered by the user, and then you added the query to the end of the URLs to be requested.

Finally, rather than output the results to the command line, you output them to the Web page.

The results should look something like Figure 2.

Figure 2. Results of running the servlet



Adding attribute templates

If you scroll down the page, you'll find that YouTube results are also on the page, but they don't look very nice, because they don't have any images. Similarly, it would be nice to be able to provide links to the news items Yahoo! presents. To do that, however, you need to provide a way to specify attributes in your templates, and not just elements.

To do that, you can add a new attribute to the `Service` class, and alter the templates somewhat, as you can see in Listing 24.

Listing 24. Adding attributes to the `Service` class

```
public class Service {  
    String name = "";  
    String baseUrl = "";  
    String template = "";
```

```

String[] elementValues = {"title",
                          "desc"};
String[] attributeValues = {"src"};
String recordExp = "";

public static Service[] getServices(){
    Service[] services = new Service[2];

    Service thisService = new Service();
    thisService.name = "Yahoo! Search";
    thisService.baseURL =
"http://api.search.yahoo.com/NewsSearchService/V1/newsSearch?appid=ma
shupid&type=all&query=";
    thisService.template =
        "<p><b><a
value='href'><value/></a></b>:<value/>
</p>";
    thisService.elementValues[0] = "Title";
    thisService.elementValues[1] = "Summary";
    thisService.attributeValues[0] = "ClickUrl";
    thisService.recordExp = "/ResultSet/Result";
    services[0] = thisService;

    Service thisService2 = new Service();
    thisService2.name = "YouTube";
    thisService2.baseURL =
"http://www.youtube.com/api2_rest?method=youtube.videos.list_by_tag&de
v_id=s2gNEM-7qoU&tag=";
    thisService2.template =
        "<p><img value='src' />
<value/>:<value/></p>";
    thisService2.elementValues[0] = "title";
    thisService2.elementValues[1] = "description";
    thisService2.attributeValues[0] = "thumbnail_url";
    thisService2.recordExp = "/ut_response/video_list/video";
    services[1] = thisService2;

    return services;
}
}

```

Here you added an `attributeValues` array for each object, but you also added new attributes to the templates. The form is slightly different from the value elements. The value attribute determines the element to carry the attribute, as well as the name of the attribute to appear in the final document. For example, the template element `` gets translated into an element of ``.

Let's see how that works.

Processing the attribute templates

Processing the attribute values is similar to processing the element values, as you can see in Listing 25.

Listing 25. Processing attribute templates

```

...
        String replacementValue =
            valueNode.getTextContent();
        Node replacementNode =
            templateDoc.createTextNode(replacementValue);
        parentNode.replaceChild(replacementNode,
            thisTemplateNode);
    }
}

expression = "@value";
templateNodes = (NodeList) xpath.evaluate(
    expression, templateDoc, XPathConstants.NODESET);
for (int j=0; j < templateNodes.getLength(); j++){

    Attr thisNode = (Attr) templateNodes.item(j);
    Element parentElement =
        (Element)thisNode.getOwnerElement();

    parentElement.removeAttribute("value");
    if ( (Boolean)xpath.evaluate(svc.attributeValues[j],
        recordNodes.item(i),
        XPathConstants.BOOLEAN) ) {
        Node valueNode = (Node)xpath.evaluate(
            svc.attributeValues[j],
            recordNodes.item(i),
            XPathConstants.NODE);
        parentElement.setAttribute(
            thisNode.getNodeValue(),
            valueNode.getTextContent());
    }
}

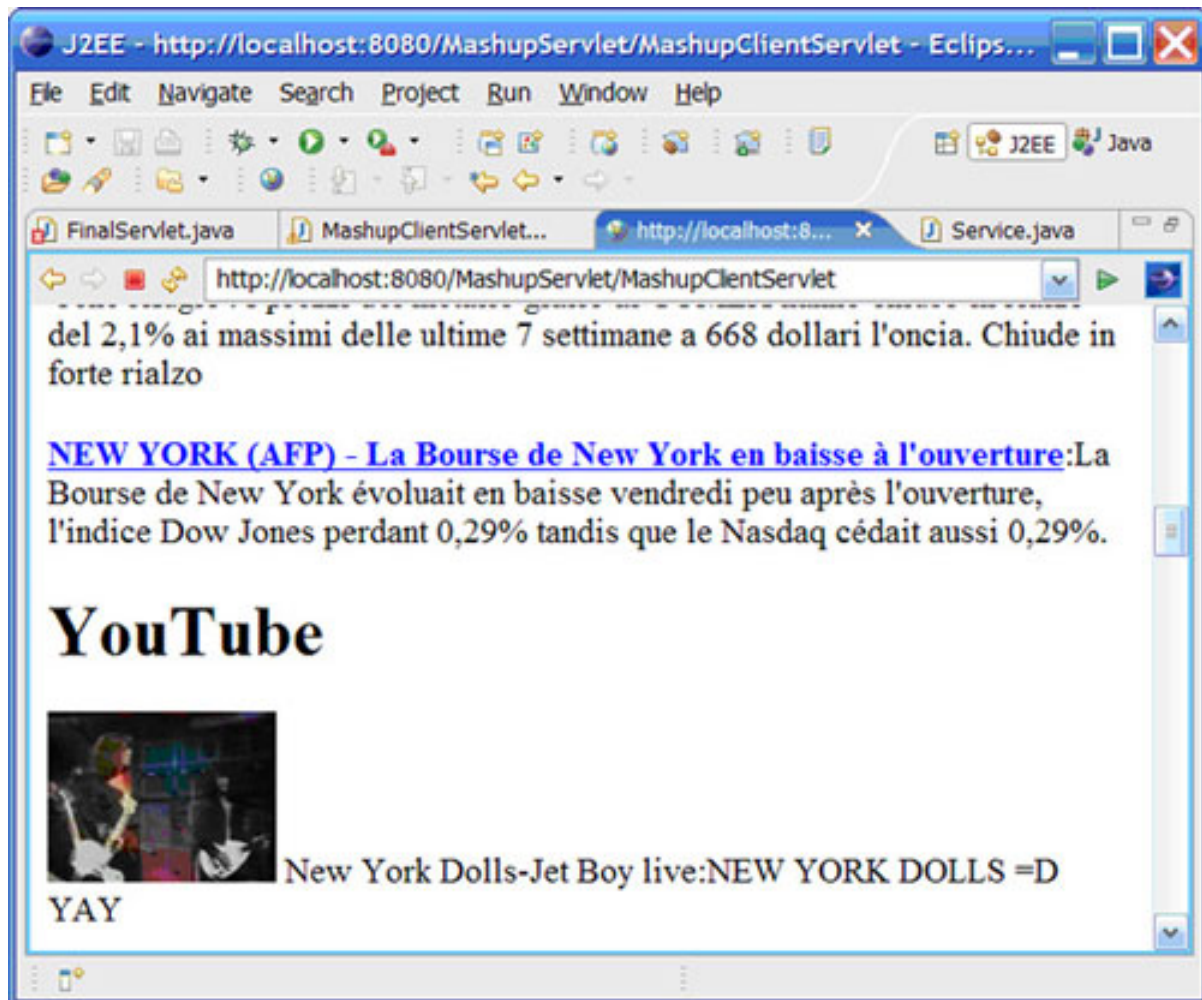
Node importedNode = hostDoc.importNode(
    templateDoc.getDocumentElement(), true);
hostRoot.appendChild(importedNode);
hostRoot.appendChild(hostDoc.createTextNode("\n"));
}
}
...

```

First, define the XPath expression for attributes called value, and get a list of Nodes to which it applies. For each one, get a reference to the Node itself, and then the element carrying it. Once you have that, you can remove the old attribute and if the relevant information is present, add a new one with the replacement value.

The response is a page that includes links and images, as you can see in Figure 3.

Figure 3. Links and images



Section 7. Really mashing services together

Okay, up until now, you created a mashup in that you have information from different services, but they really haven't been interacting with each other. In this section, you'll change that. You'll add a third service, the Technorati cosmos query that shows the number of times bloggers have linked to a given URL, and display that value for each of the entries returned by the Yahoo! service.

Separating processing into a separate method

The first step is to take the actual processing of a service out of the main body of the `doPost()` method and make it into a method of its own so you can call it

recursively, as you can see in Listing 26.

Listing 26. Separating processing into a separate method

```
...
public class MashupClientServlet
    extends javax.servlet.http.HttpServlet
        implements
    javax.servlet.Servlet {

    public MashupClientServlet() {
        super();
    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException,
        IOException {
        ...
    }

    public static Node renderService(String query,
        Service svc,
        Document hostDoc){

        try {
            Node serviceElement =
            hostDoc.createElement("span");
            DocumentBuilder builder =
            DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document document =
            builder.parse(svc.baseURL+query);

            ...

            toImport =
            hostDoc.importNode(templateDoc.getDocumentElement(),
            true);
            serviceElement.appendChild(toImport);
        }

        return serviceElement;
    } catch (Exception e){
        e.printStackTrace();
        return null;
    }
}

protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException,
    IOException {

    String query =
    request.getParameter("query").toString();

    try {
        DocumentBuilder builder =
        DocumentBuilderFactory.newInstance().newDocumentBuilder();
```

```
        Service[] svcs = Service.getServices();
        Document hostDoc = builder.parse(
            new InputSource(new
StringReader("<div/>")));
        Node hostRoot =
hostDoc.getDocumentElement();

        for (int k=0; k < svcs.length; k++){

            Service svc = svcs[k];

            Element nameElement =
hostDoc.createElement("h1");
nameElement.appendChild(hostDoc.createTextNode(svc.name));

            Node serviceElement =
hostDoc.createElement("span");
serviceElement.appendChild(nameElement);

            serviceElement.appendChild(
                renderService(query, svc,
hostDoc));

            hostRoot.appendChild(serviceElement);
hostRoot.appendChild(hostDoc.createTextNode("\n"));

        }

        DOMSource source = new
DOMSource(hostDoc);
        ...
```

First, create a new method, `renderService()`, and feed it the query, `Service` object, and `Document`, so you can simply copy and paste the functionality right into the method with no changes. The method returns the `serviceElement`, or a collection of all of the processed templates for that service. Because it's part of the output document, you can simply append the results to the document as usual.

If you save and run the servlet, the results should be identical to what you saw before (although the results themselves might be different, of course!) because you haven't changed any of the functionality, just moved it around a little.

Adding subservices

To add subservices, you'll make another couple of changes to the `Service` class, as in Listing 27.

Listing 27. Creating subservices

```
public class Service {
    String name = "";
    String baseURL = "";
```

```

String template = "";
String[] elementValues = {"title",
                          "desc"};
String[] attributeValues = {"src"};
String recordExp = "";
Service subSvc = null;
String filterExp = "";

public static Service[] getServices(){
    Service[] services = new Service[2];

    Service subService = new Service();

    subService.name = "Technorati";
    subService.baseURL =
    "http://api.technorati.com/cosmos?key=0368d22f6a2f864dad1c59feb07153ee
&url=";
    subService.template = "<span><value/></span>";
    subService.elementValues[0] = "inboundlinks";
    subService.elementValues[1] = "bogus";
    subService.attributeValues[0] = "thumbnail_url";
    subService.recordExp = "/tapi/document/result";

    Service thisService = new Service();

    thisService.name = "Yahoo! Search";
    thisService.baseURL =
    "http://api.search.yahoo.com/NewsSearchService/V1/newsSearch?appid=ma
shupid&type=all&query=";
    thisService.template = "<p><b><a
value='href'><value/></a></b>"
        + " :<value/> (Linked <subService/>
times)</p>";
    thisService.elementValues[0] = "Title";
    thisService.elementValues[1] = "Summary";
    thisService.attributeValues[0] = "ClickUrl";

    thisService.recordExp = "/ResultSet/Result";

    thisService.subSvc = subService;
    thisService.filterExp = "ClickUrl";

    services[0] = thisService;

    ...

```

To start, modify the `Service` class to enable objects to include a subservice, as well as an expression on which to filter the subservice.

For example, you defined a `Service` object for the Technorati service, which will act as a subservice for the Yahoo! service. To call the Technorati service, feed it a URL to check. This URL is the filter expression; the program filters the results based on that URL. If you look at where you add the subservice to the Yahoo! service, you'll see that you specified that the URL on which you'll filter the Technorati service is the Yahoo! service's `ClickUrl` value.

Notice also that [Listing 27](#) adds a `subService` element to the Yahoo! service's template, which tells the application to include the output of the subservice.

Other than the way you call the service -- and the fact that it's not added to the array -- there's nothing odd about the subservice itself.

Processing subservices

Processing the subservice is a matter of recursively calling the `renderService()` method, as you can see in Listing 28.

Listing 28. Processing subservices

```

...
public static Node renderService(String query, Service svc,
                                Document hostDoc){

    try {
...
        parentElement.removeAttribute("value");
        if ( (Boolean)xpath.evaluate(svc.attributeValues[j],
recordNodes.item(i), XPathConstants.BOOLEAN)) {
            Node valueNode = (Node)xpath.evaluate(svc.attributeValues[j],
recordNodes.item(i), XPathConstants.NODE);
            parentElement.setAttribute(thisNode.getNodeValue(),
valueNode.getTextContent());
        }
    }

    expression = "//subService";
    NodeList subServiceNodes = (NodeList) xpath.evaluate(
        expression, templateDoc, XPathConstants.NODESET);
    if (subServiceNodes.getLength() > 0) {
        Node thisNode = subServiceNodes.item(0);
        Node parentNode = thisNode.getParentNode();
        if ( (Boolean)xpath.evaluate(svc.filterExp,
            recordNodes.item(i), XPathConstants.BOOLEAN)) {
            Node valueNode = (Node)xpath.evaluate(svc.filterExp,
                recordNodes.item(i), XPathConstants.NODE);
            String filter = valueNode.getTextContent();
            Node renderedSubService =
                renderService(filter, svc.subSvc, hostDoc);
            toImport = templateDoc.importNode(renderedSubService,
                true);
            parentNode.replaceChild(toImport, thisNode);
        }
    }

    toImport = hostDoc.importNode(templateDoc.getDocumentElement(),
true);
    serviceElement.appendChild(toImport);

    return serviceElement;
    } catch (Exception e){
        e.printStackTrace();
        return null;
    }
}
...

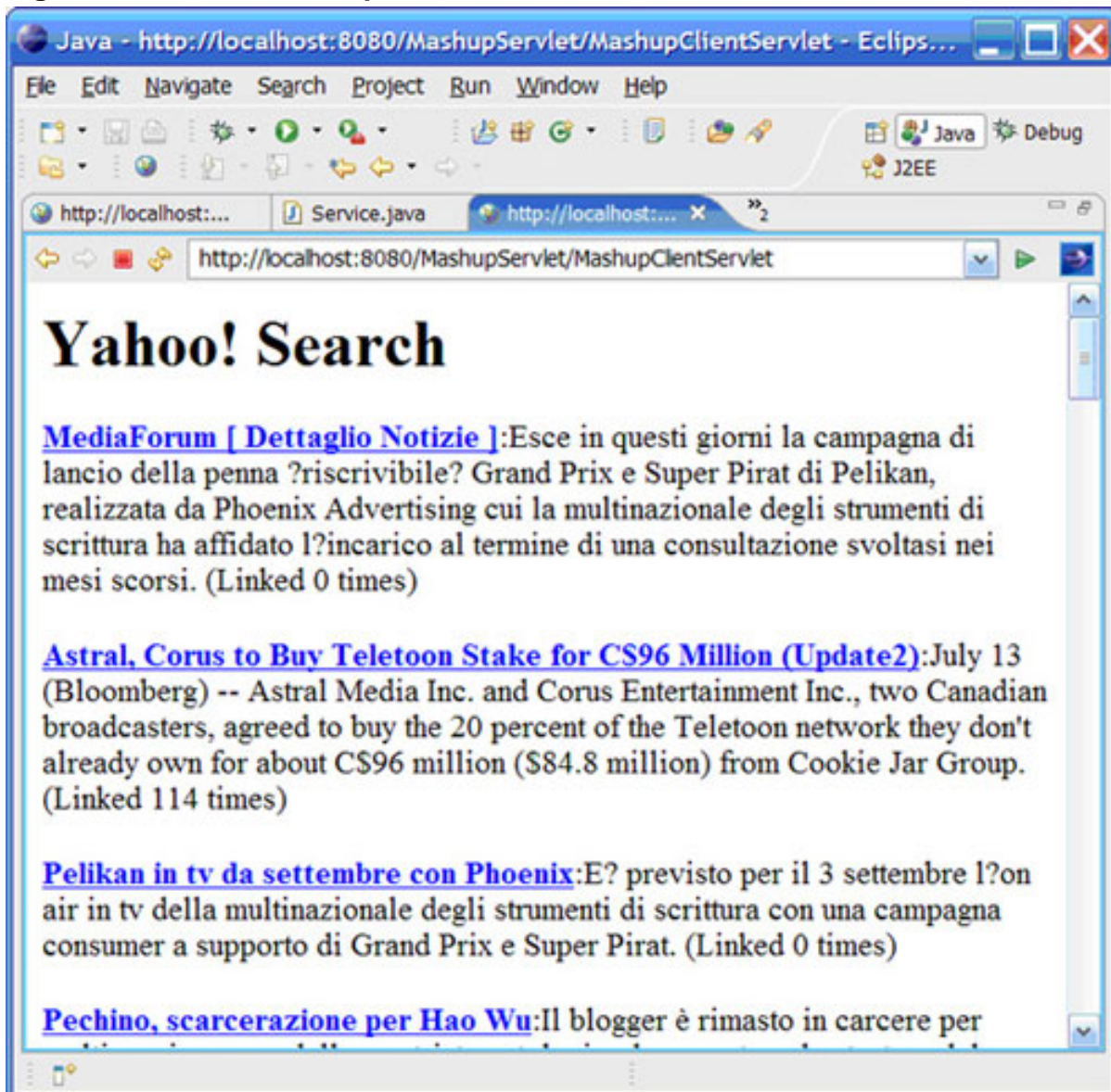
```

After the code checks the template for value elements and attributes, it looks for `subService` elements. The process is virtually identical to replacing value elements, except that if the application finds a `subService` element, rather than

looking for the data from an XPath expression, it feeds the service's subservice and filter to the `renderService()` method, which runs recursively and provides a node or nodes to replace the `subService` element.

The result is that the servlet runs and for each entry returned by the Yahoo! service, you get a notation of how many links Technorati knows about, as you can see in Figure 4, a search for "boing boing".

Figure 4. The final mashup



Note that performance is not stellar, because you make not 2, not 3, but 12 different Web requests. But I'll deal with performance, or at least caching in Part 2.

Section 8. Summary

Wrap up and looking ahead

In this tutorial, you looked at utilizing data from REST-based Web service APIs using XML techniques such as DOM manipulation and XPath. You created a mashup of different services first by simply showing more than one service on a page, and later by integrating one service with another. You did this by following a template approach, which will enable you to add additional services and to change appearances at will.

This will come in handy in future installments, when you programmatically add and remove services.

In Part 2, you'll look at how to store XML data natively in DB2 to create a source of cached data.

Downloads

Description	Name	Size	Download method
Part 1 code source	x-ultimashup1/mashup1_source.zip	8KB	HTTP

[Information about download methods](#)

Resources

Learn

- [The ultimate mashup -- Web services and the semantic Web](#) tutorial series: Take the other tutorials and create a custom mashup.
- [Programmable Web](#): Stay up to date with the latest on mashups and the new Web 2.0 APIs.
- [Amazon Light](#): Look at this page with listings of products sold by Amazon.com.
- [ActorTracker](#): Check out this combination of Amazon, eBay, and other movie data.
- [Considering Ajax, Part 1: Cut through the hype](#) (Chris Laffra, developerWorks, May 2006): Consider this set of discussion points for every developer before you use Ajax techniques for a Web site.
- [Ajax page](#): Visit this page sponsored by the [Mozilla Development Center](#).
- [DB2 and open source: Put yourself on the map with Google Maps API, DB2/Informix, and PHP on Linux](#) (Marty Lurie and Aron Y. Lurie, developerWorks, March 2006): Create an easy-to-use map with your data on it.
- [Building Web service applications with the Google API](#) (Nicholas Chase, developerWorks, May 2002): Learn to embed Google search results and other information in your Java applications in this tutorial.
- [Second Generation Web Services](#): On XML.com, find coverage of the REST architecture.
- [REST and the Real World](#): Read more on REST (Representational State Transfer) from XML.com.
- The [W3C Semantic Web Activity](#) site: Read about the Semantic Web.
- [W3C RDF Activity](#): Visit this site for the latest on Resource Description Framework.
- [Introduction to Jena: Use RDF models in your Java applications with the Jena Semantic Web Framework](#)(Philip McCarthy, developerWorks, June 2004): Find out how to use the Jena Semantic Web Toolkit to exploit RDF data models in your Java applications.
- [What is RSS?](#): From XML.com, learn about this syndication format for news, content, and personal weblogs.
- [Introduction to XML](#) (Doug Tidwell, developerWorks, August 2002): Take this tutorial for a good grounding in XML.
- [Understanding DOM](#) (Nicholas Chase, developerWorks, July 2003): Get a

better understanding of dealing with XML as an object in this tutorial.

- [developerWorks Java zone](#): Learn more about Java programming with articles, tutorials, forums, and more.
- [developerWorks XML zone](#): Find a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks about XML.

Get products and technologies

- [W3C SOAP Specification](#): Get the latest version.
- [Scraping with style: scrAPI toolkit for Ruby](#): Try this technology for your mashups.
- [Rational Web Developer](#): Download a trial version and make your development easier.
- [DB2 Enterprise 9](#): Download a trial version of DB2 9 or [DB2 Express-C 9](#), a no-charge version of DB2 Express 9 data server.

Discuss

- [developerWorks XML forums](#): Communicate with other XML developers trying to solve the same problems you are.
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

About the author

Nicholas Chase

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the Chief Technology Officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams).

Trademarks

IBM, DB2, pureXML, and Rational are trademarks of IBM Corporation in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.