

http://java.sun.com/developer/technicalArticles/J2EE/mashup_1/index.html

Article

Mashup Styles, Part 1: Server-Side Mashups

By Ed Ort, Sean Brydon, and Mark Basler, May 2007

[Articles Index](#)

This article is the first in a series that examines some of the most common approaches, or styles, for doing mashups. The articles in the series compare and contrast these styles and discuss some of the major design considerations related to each.

Contents

- [What's a Mashup?](#)
- [What's in a Name?](#)
- [Mashup Styles](#)
- [A Brief Tour of Mashups in the Pet Store Application](#)
- [Server-Side Mashups](#)
- [Reasons for Using the Proxy Style](#)
- [An Example of a Proxy-Style Mashup](#)
- [Design Considerations in Server-Side Mashups](#)
- [Summary](#)
- [For more Information](#)
- [About the Authors](#)

What's a Mashup?

It's fairly common these days to go to a web site such as a site that displays a list of restaurants in a given city, and then request the site to display a map that highlights the location of the restaurants. [Figure 1](#) shows an example of this type of site. Although you might assume that both the list and map come from resources within the site, the two types of information often originate from different sites. More specifically, the services that generate the two types of content are in different sites. This merging of services and content from



Get Training: Java EE 5 Platform and Ajax

- [Developing Secure Web-Tier Applications \(DTJ-3109\)](#)
- [Developing Applications for the Java EE Platform \(FJ-310\)](#)
- [Developing JavaServer Faces Web Applications With Ajax Using Sun Studio Creator \(DTJ-2105\)](#)
- [Developing JavaServer Faces Components With Ajax \(DTJ-3108\)](#)

1. [Indian Oven](#)
2. [Maharaja Indian Restaurant](#)
3. [Scenic Indian Restaurant](#)
4. [Rasoi Indian Cuisine](#)
5. [New Delhi Restaurant](#)
6. [Tandoori Mahal Indian Restaurant](#)
7. [Aroma Indian-Scenic Indian](#)

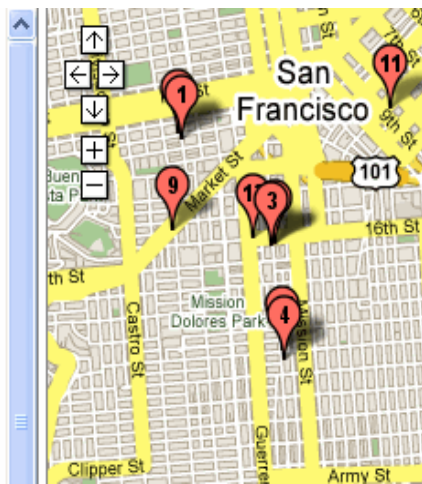


Figure 1: A Site that Highlights Restaurant Locations

multiple web sites in an integrated, coherent way is called a *mashup*.

Most mashups do more than simply integrate services and content. Sites that do mashups typically add value. They benefit users in a way that's different and better than the individual services they leverage. For example, [Delexa.org](#) brings together data from the social bookmarking site [del.icio.us](#) and the web site traffic tracker [Alexa](#). Using the topic tags that it extracts from [del.icio.us](#), [Delexa.org](#) allows users to search for top traffic sites by topic, something that you can do in only a limited way on the [Alexa](#) site. For example, a user can ask to see the most frequently visited gardening sites or the

most frequently visited karate sites.

Mashups are appearing on the web at an extremely fast rate. Three new mashups typically appear on the web each day. You can see some of the newest ones on the [ProgrammableWeb](#) site. The bulk of the mashups on the web involve the use of maps. Many of these sites use mapping services such as those provided by [Yahoo Maps](#) and [Google Maps](#). For example, [mibazaar.com](#) uses the Google Maps service in a mashup that displays the type of list and map shown in [Figure 1](#).

However, mashup sites are not limited to mapping. As the [Delexa.org](#) site illustrates, mashups can involve other types of information -- in this case, topics identified in bookmarked pages and web traffic data. In addition, a growing number of mashups involve multimedia content from sites such as [Flickr](#) and [YouTube](#), or include services from shopping sites such as [Amazon.com](#).

If you examine these mashups, you'll notice a variety of approaches. This article is the first in a series that examines some of the most common approaches, or styles, for doing mashups. The articles in the series compare and contrast these styles and discuss some of the major design considerations related to each. This first article examines a style called server-side mashups.

What's in a Name?

Before delving into mashup styles, it's important to understand that despite the definition of the term *mashup* given in this article, no single definition encompasses all mashups. This series of articles uses a rather loose definition of *mashup*. If a web site uses data or functionality from another web site and combines it in an application, it's a mashup. The application can access the data or functionality in various ways. It can use formal Representational State Transfer (REST)-based APIs provided by the other site. Or it can do some informal screen scraping, in which it extracts data from the displayed output of a program on another site. Or it can access an RSS feed or use a widget provided by another site. However, if the application simply links to another site, for instance, through an HTML `href` link, it is not a mashup. In the simplest sense, if your web application is using other web sites, it's a mashup.

There is some fuzziness in what constitutes a mashup and probably some degree of subjectivity too.

As you read the articles in this series, keep in mind that there is some fuzziness in what constitutes a mashup and probably some degree of subjectivity too.

Mashup Styles

The two primary mashup styles are server-side mashups and client-side mashups. As you might expect, server-side mashups integrate services and content on the server. The server acts as a proxy between a web application on the client, typically a browser, and the other web site that takes part in the mashup. In a server-side mashup, all the requests from the client go to the server, which acts as a proxy to make calls to the other web site. So, in a server-side mashup, the work is pushed from the web application client to the server.

In general, a server-side mashup works as illustrated in [Figure 2](#).

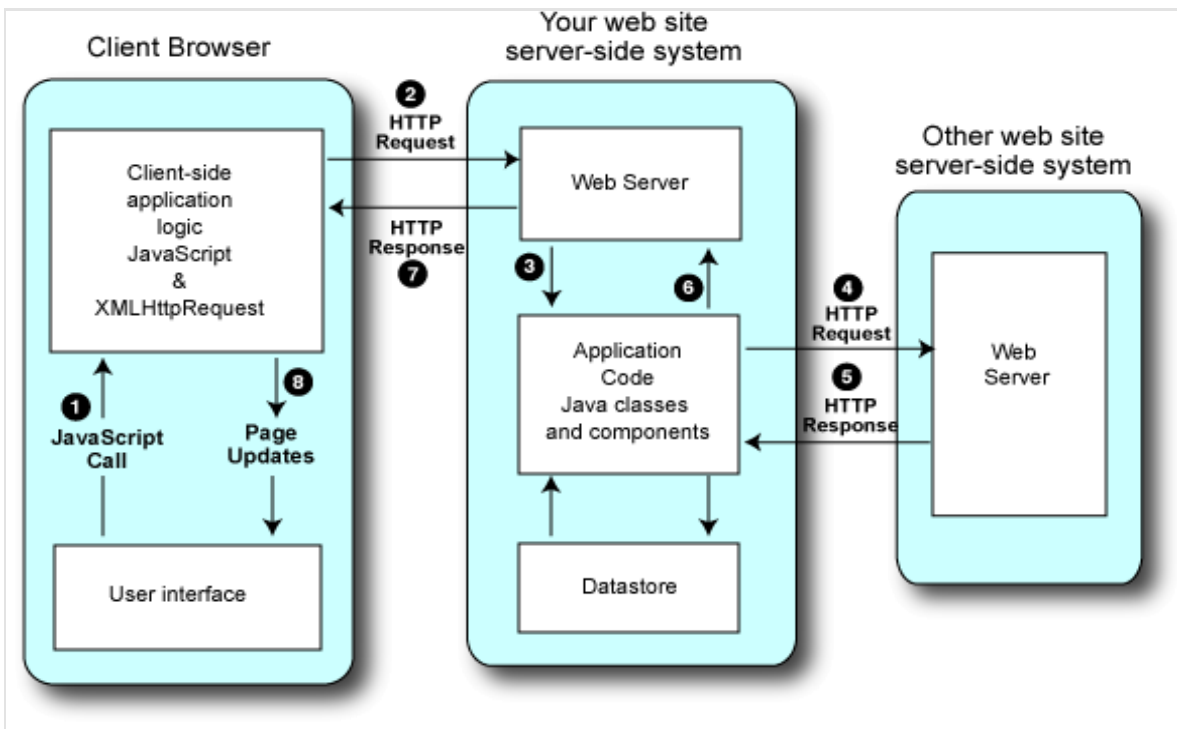


Figure 2: How a Server-Side Mashup Works

1. A user generates an event in the client, typically a web page in a browser. The event triggers a JavaScript function in the client.
2. The client makes a request to the server in your web site. The request is typically an Ajax request in the form of an `XmlHttpRequest` object.
3. A web component such as a servlet receives the request and calls a method on a Java class or on multiple Java classes. The class or classes encapsulate the code to connect and interact with the other web site in the mashup. This class, referred to here as the proxy class, could be a Java Platform Enterprise Edition (Java EE) component or just a plain Java class that you write.
4. The proxy class processes the request, augments it as needed, and opens a connection to the mashup site, that is, the web site that provides the needed service.
5. The mashup site receives the request, usually in the form of an HTTP GET or HTTP POST, processes the request, and returns data to the proxy class.
6. The proxy class receives the response and may transform it to an appropriate data format for the client. It can also cache the response for future request processing.
7. The servlet returns the response to the client.
8. A callback function exposed in the `XmlHttpRequest` updates the client view of the page by manipulating the Document Object Model (DOM) that represents the page.

Client-side mashups integrate services and content on the client. They mash up directly with the other web site's data or functionality. For example, in a client-side mashup, the client might make requests directly to the other web site.

In general, a client-side mashup works as illustrated in [Figure 3](#).

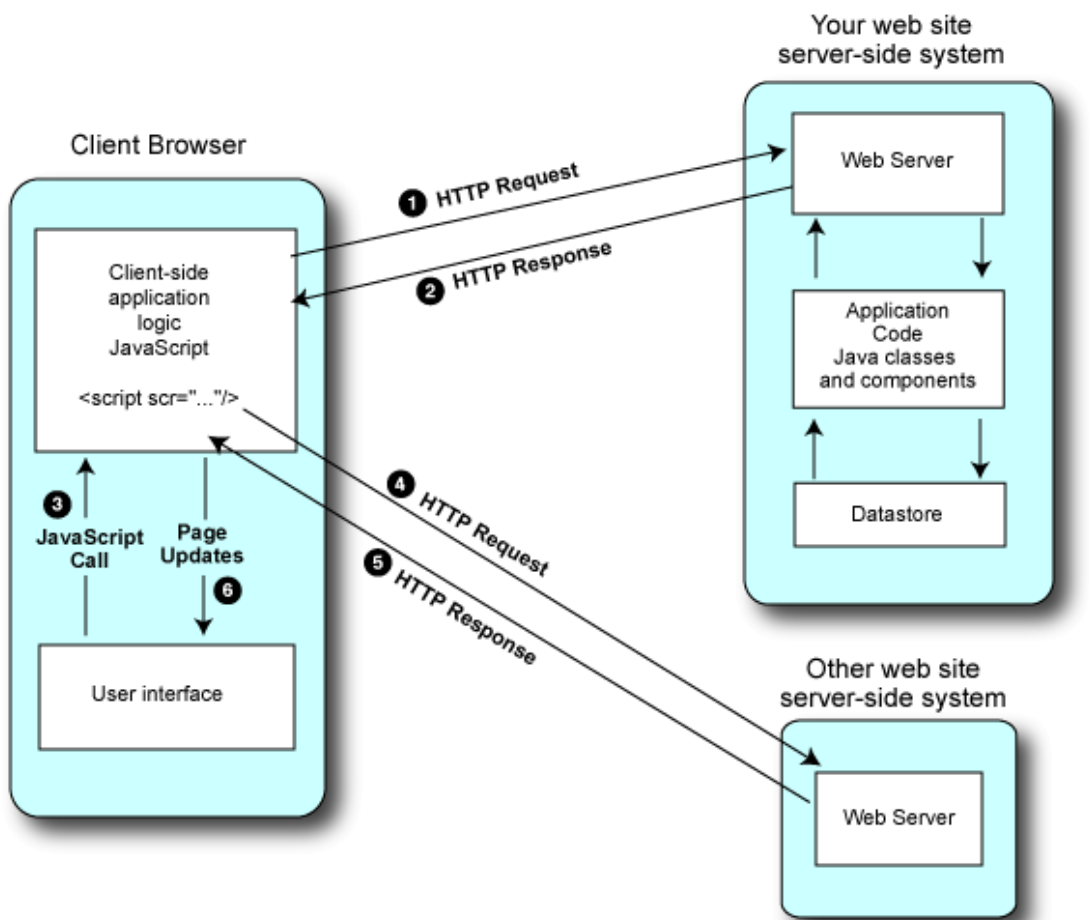


Figure 3: How a Client-Side Mashup Works

1. The browser makes a request to the server in your web site for the web page.
2. The server on your web site loads the page into the client. The page usually includes a JavaScript library from the mashup site -- for example, from a site such as Google Maps -- to enable using the mashup site's service. If the page does not include a JavaScript library, you can write a custom JavaScript function to facilitate the mashup.
3. Some action in the browser page calls a function in the JavaScript library provided by the mashup site, or it calls the custom JavaScript function that you created. The function creates a `<script>` element that points to the mashup site.
4. Based on the `<script>` element, a request is made to the mashup site to load the script.
5. The mashup site loads the script. Typically, a local callback in the browser page is then executed with a JavaScript Object Notation (JSON) object sent as a parameter.
6. The callback function updates the client view of the page by manipulating the DOM that represents the page.

However, some mashups don't correspond to either of these styles. Many see these other mashups as expanding the boundary of what constitutes a mashup.

You can find a good demonstration of these styles in the [Java PetStore 2.0 demo](#), a reference web application that is part of the [Java BluePrints Program](#). For simplicity, this article refers to the application as Pet Store. Article in this series will use the mashups in Pet Store to illustrate mashup techniques. Pet Store shows how you can use the [Java EE 5 platform](#) with Web 2.0 technologies. The application also demonstrates how you can use Java EE 5 technologies such as JavaServer Faces technology (often referred to as JSF) and the Java Persistence API in mashups. For an overview of Pet Store, see [Introducing the Java Pet Store 2.0 Application](#). You can learn more about Pet Store by [downloading it](#), examining the source code, and deploying it on a Java EE 5-compliant application server. You can also run a [live version of Pet Store](#).

A Brief Tour of Mashups in the Pet Store Application

Pet Store is a web application for selling and buying pets. If you're a seller, you can add information about your pet such as a description, price, and photo, to the application's pet catalog. You can also add information about yourself such as your address. If you're a prospective buyer, you can select one or more available pets from the

Yahoo Maps Geocoding service because it's a good example of how to design and build a server-side mashup. Pet Store's mashup with the RSS feed is also a server-side mashup, but that mashup has some unique aspects that will be covered in a subsequent article. Pet Store's mashup with the Google Maps service is a client-side mashup. Client-side mashups as well as other mashup styles will be covered in subsequent articles.

Server-Side Mashups

In a server-side mashup, the service or content integration takes place in the server. This is in contrast to a client-side mashup, where the service or content integration takes place in the client, typically a web browser. A server-side mashup is also called a *proxy-style mashup* because a component in the server acts as a proxy to the service. In general, a *proxy* is a component that acts as an intermediary between two parties. In a server-side mashup, a component in the server on your web site acts as a proxy between a page in the browser and another web site. As illustrated in [Figure 2](#), the browser client makes `HttpRequests` to your web site. Your web site then does the work of mashing up with the other web site.

Reasons for Using the Proxy Style

Perhaps the biggest challenge in doing a mashup is contending with the basic security protection that the browser security sandbox provides. The browser security sandbox is responsible for keeping personal information secure. Many mashups use Ajax functionality. An `XMLHttpRequest` is a JavaScript object that is used to exchange data asynchronously between a client and server in an Ajax transaction. To protect against possible maliciousness, most browsers allow JavaScript code that contains an `XMLHttpRequest` to communicate only with the site domain, that is, the computer system that hosts the web site, from which the page was loaded. The site from which the page is loaded is usually called the server of origin. For example, if the page containing the `XMLHttpRequest` is loaded from `mySite.com`, the `XMLHttpRequest` can only connect to `mySite.com`. It won't allow the `XMLHttpRequest` to connect to another site. If the mashup requires a service in a site that is not the server of origin, such as `yourSite.com`, there's no way to access it through an `XMLHttpRequest`. Although the server-of-origin policy adds security, that security makes the creation of mashups more difficult.

In a server-side, also called proxy-style, mashup the service or content integration takes place in the server. A significant reason for using the proxy style is to overcome the constraints of the browser security sandbox and make `XMLHttpRequests` across domains.

In a proxy-style mashup, a server-side proxy -- and not JavaScript code in the client -- accesses the service. Because of that, a server-side mashup is not subject to the browser security sandbox and can connect to the other site -- in this case, `yourSite.com` -- to access the service. Note that a Java EE application running in a server can access any other web site.

Here are some other good reasons for using proxy style in doing a mashup:

- The Java EE and Java SE platforms provide many libraries that make it easy to access other web sites from the server. These libraries also simplify various necessary tasks in processing requests related to mashups.
- The proxy used in a server-side mashup can serve as a buffer between the client and the other web site. The proxy can shield the client from problems in the other web site. For example, if the other site is slow or is responding with errors, your application can provide some default data or cached data to the client. It can also display appropriate error messages as part of a graceful degradation response.
- The other site in a mashup might return a large amount of data. In this case, the server can send data to the client in smaller chunks or send only the portion of the data that the client needs. For example, in the Pet Store mashup with the RSS feed, the server sends the client only a small portion of the data from the feed.
- You can cache data returned by the service on the server. The server can then satisfy subsequent requests for the service by returning data from the cache. This can speed up responsiveness by saving the client from making unnecessary requests to the other web site.
- You might need to transform the data returned by a service into a different format. For example, the service in a mashup might have an API that is SOAP-based or XML-based. You can make it easier for a browser to handle the data returned in the mashup by having the server transform the data into a format such as JSON before sending it to the client.
- The service might not provide exactly what your application needs. You might need to manipulate the data returned by a service or combine it with other content before returning it to the client.
- You can handle security requirements more easily on the server. For example, you can use the server to supply a key or token. Or you can use the server to authenticate to the other web site or to connect to the

other web site though a secure protocol such as HTTPS.

- You can make concurrent and asynchronous calls to many data sources at the same time from the server. The server can connect to many sites to get what it needs. Most browsers limit the number of concurrent `XMLHttpRequests` that can be outstanding at one time to only two or three. This could be fairly complicated if you tried to do it in the client.

An Example of a Proxy-Style Mashup

The Pet Store mashup with the [Yahoo Maps Geocoding service](#) is an example of a mashup that uses the proxy style. Recall that Pet Store uses the Yahoo Maps Geocoding service together with the Google Maps service to display a map of pet locations. The Google Maps service actually produces the map. However, to map a location, the Google Maps service requires the location to be specified in terms of its longitude and latitude. So to display a map as shown in [Figure 4](#), Pet Store needs a way to convert each pet address. The Yahoo Maps Geocoding service does precisely that: It converts each address to a longitude and latitude.

Rather than requesting the Yahoo Maps Geocoding service each time a map is requested, Pet Store calls the service once: when a seller adds a pet to the application's catalog of pets. As part of the information it requires for a new pet entry, Pet Store asks the seller for an address. When the seller submits the content for the pet, Pet Store calls the Yahoo Maps Geocoding service to get the latitude and longitude corresponding to the address. Pet Store then stores the longitude and latitude in a database along with the other information that the seller submits. This frees the application from having to request the Geocoding service each time a map is needed. Pet Store provides the longitude and latitude to the Google Maps service simply by accessing the database.

Using the Yahoo Maps Geocoding Service

The Yahoo Maps Geocoding service is a REST-based web service that is available for use by other web sites through a public API. To access the service through the API, you construct a URL with the required parameters and go to that URL. [Code Sample 1](#) shows an example. For formatting purposes, the URL is displayed on multiple lines. In actual code, you should specify the URL on one line.

```
http://api.local.yahoo.com/MapsService/V1/geocode
?appid=com.sun.blueprints.ui.geocoder
&location=4140%20Network%20Circle,%20Santa%20Clara,%20CA,%2095054
```

Code Sample 1: Constructing a URL for the Yahoo Geocoder Public API

The URL `http://api.local.yahoo.com/MapsService/V1/geocode` is the request URL path for the Yahoo Maps Geocoding service. The URL is appended with one or more pairs of request parameters and values. The `appid` value identifies the application, and the `location` value is the address. Notice that spaces in the address are encoded.

In response, the service returns an XML document that contains the longitude and latitude of the address. For example, the service request in the previous example returns the XML document shown in [Code Sample 2](#).

```
<ResultSet xsi:schemaLocation=
"urn:yahoo:maps http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
  <Result precision="address">
    <Latitude>37.395908</Latitude>
    <Longitude>-121.952735</Longitude>
    <Address>4140 NETWORK CIR</Address>
    <City>SANTA CLARA</City>
    <State>CA</State>
    <Zip>95054-1778</Zip>
    <Country>US</Country>
  </Result>
</ResultSet>
```

Code Sample 2: An XML Document Returned by the Yahoo Maps Geocoding Service

If there is an error, the service responds with an HTTP error code and an XML error response message.

Assembling the URL and Calling the Service From a Proxy

Let's look at the code in the proxy class that Pet Store uses in its mashup with the Yahoo Maps Geocoding service. The proxy class is named `GeoCoder`. A class such as this would usually be called from a servlet or JavaServer Faces component that processes the browser client's Ajax call, as shown in [Figure 2](#).

[Code Sample 3](#) shows a snippet of code from the `GeoCoder` class. You can look at the complete code for the class [here](#).

```
//The URL of the geocoding service we will be using
private static final String SERVICE_URL =
    "http://api.local.yahoo.com/MapsService/V1/geocode";

//The default application identifier required by the geocoding service.
//This may be overridden by setting the applicationId property
static final String APPLICATION_ID =
    "com.sun.javaee.blueprints.components.ui.geocoder";

//Now the method that does the work
public GeoPoint[] geoCode(String location) {

    ...

    // Perform the actual service call and parse the response XML document,
    // then format and return the results
    Document document = null;
    StringBuilder sb = new StringBuilder(SERVICE_URL);
    sb.append("?appid=");
    sb.append(applicationId);
    sb.append("&location=");
    sb.append(location);
    try {
        document = parseResponse(sb.toString());
        return convertResults(document);
    } catch (IllegalArgumentException e) {
        ...
    }
}
```

Code Sample 3: A Snippet of Code From the `GeoCoder` Class

Notice that the `geoCode` method in `GeoCoder` makes the call to the geocoding service. The method takes a `String location` parameter that represents an address. The `location` can be formatted in various address formats such as "city, state" or "city, state, zip". Both the application identifier (application ID) and the location parameters are also encoded, as shown in [Code Sample 6](#). Notice too that the `geoCode` method does the following:

- Identifies the URL of the Yahoo Maps geocoding service.

```
private static final String SERVICE_URL =
    "http://api.local.yahoo.com/MapsService/V1/geocode";
```

- Identifies the application. The method needs to specify an application ID in the request to the geocoding service.

```
static final String APPLICATION_ID =
    "com.sun.javaee.blueprints.components.ui.geocoder";
```

- Assembles the URL with one or more pairs of request parameters and values.

```
StringBuilder sb = new StringBuilder(SERVICE_URL);
sb.append("?appid=");
sb.append(applicationId);
sb.append("&location=");
sb.append(location);
```

- Submits the request by opening a stream using the `java.net.URL` class.

```
DocumentBuilder db =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();
InputStream stream = null;
try {
    // make call to the mashup web site and receive
    // XML document result
    stream = new URL(url).openStream();
    return db.parse(stream);
}
```

Parsing and Converting the Results

The `geoCode` method uses two private methods as helpers. One helper method parses the response from the geocoding service. The other helper method converts the results after the results are parsed. Parsing and converting results are common activities when accessing any service. Let's look at the code for the helper methods.

[Code Sample 4](#) shows a snippet of the code for the method that parses the response, `parseResponse`. You can view the complete code for the method by examining the code for the [GeoCoder class](#).

```
private Document parseResponse(String url)
    throws IOException, MalformedURLException,
        ParserConfigurationException, SAXException {

    DocumentBuilder db =
        DocumentBuilderFactory.newInstance().newDocumentBuilder();
    InputStream stream = null;
    try {
        // make call to the mashup website and receive XML document result
        stream = new URL(url).openStream();
        return db.parse(stream);
    } finally {
        ... clean up and close stream
    }
}
```

Code Sample 4: Parsing the Results from the Yahoo Geocoding Service

The `parseResponse` method takes a `String url` parameter that represents the URL of the resource to be parsed. It then parses the XML content at the specified URL into an XML `Document` object.

The `Document` object can be further processed to extract the necessary content, as shown in [Code Sample 5](#). The code sample shows a snippet of code for the method that converts the response, `convertResults`. You can view the complete code for the method by examining the code for the [GeoCoder class](#).

```
private GeoPoint[] convertResults(Document document) {

    List<GeoPoint> results = new ArrayList<GeoPoint>();
    GeoPoint point = null;

    // Acquire and validate the top level "ResultSet" element
    Element root = document.getDocumentElement();
    if (!"ResultSet".equals(root.getTagName())) {
        throw new IllegalArgumentException(root.getTagName());
    }

    // Iterate over the child "Result" components, creating a new
    // GeoPoint instance for each of them
    NodeList outerList = root.getChildNodes();
    for (int i = 0; i < outerList.getLength(); i++) {

        // Validate the outer "Result" element
        Node outer = outerList.item(i);
```

```

    if (!"Result".equals(outer.getNodeName())) {
        throw new IllegalArgumentException(outer.getNodeName());
    }

    // Create a new GeoPoint for this element
    point = new GeoPoint();

    // Iterate over the inner elements to set properties
    ...
}
// Return the accumulated point information
return (GeoPoint[]) results.toArray(new GeoPoint[results.size()]);
}

```

Code Sample 5: Converting the Results from the Yahoo Geocoding Service to a Java Object

The `convertResults` method takes a `document` parameter, which is the parsed `Document` object that represents the results from the Yahoo Geocoding service. The method converts the parsed XML results into an array of `GeoPoint` objects. The `GeoPoint` class is yet another helper class in Pet Store. You can view the code for the `GeoPoint` class [here](#). The class represents the location address and the longitude and latitude coordinates for the address. If there are no results from the Yahoo Geocoding service and no exception is thrown, the `convertResults` method returns a zero-length array.

Design Considerations in Server-Side Mashups

There are various things to consider when you do a server-side mashup:

- [Security](#)
- [Ways to call the service](#)
- [The format of the response from the service](#)
- [Caching of results](#)
- [The need to modify the response](#)
- [Ways to handle exceptions and errors](#)

There are various things you need to consider if you do a server-side mashup. These range from security issues to issues regarding the format and destination of the data returned by the service.

Security

Some of the security-related issues you need to address when you do a server-side mashup are the following:

- Does the exposed service require the user to log in, is the service certificate-based, or is there an authentication mechanism?
- When you expose the service, are you opening a tunneling mechanism that a malicious developer might abuse? For example, can another developer use the service to initiate a denial-of-service attack on the service you are accessing?
- Can the service be used to hide the identity of a malicious client that is illegally using the service?
- Can a malicious user use the service to avoid paying fees?

The Yahoo Maps Geocoding service has a number of specific security requirements. To access the service, you must first set up an account with Yahoo. You must also obtain from Yahoo an application ID, which is a string that uniquely identifies your application. Note that Yahoo limits the number of times that an account can use the geocoding service. It does this by tracking usage for each account ID. That's why you must specify an `appid` parameter in the call to the service. This identification requirement is fairly common. Many public REST-based web services require a key, token, or something similar to identify the calling application. For example, the Google Map API requires application keys, in which the value is specific to a deployed instance. One benefit of using the server as a proxy to a service is that you do not need to expose the application ID or token of a mashup service in your client code. Instead, the application ID or token is only used on the server and not passed to each client's browser.

Ways to Call the Service

The Yahoo Maps Geocoding service is a REST-based service. To call it, you specify the appropriate URL along with any parameter-value pairs. If you examine the [GeoCoder class](#), you'll also notice that it encodes the service call's URL. Encoding the URL to a generally accepted format makes the request as transportable as possible. This is shown in [Code Sample 6](#). The caller encodes, prior to calling the URL, any escape characters in parameters. The caller also decodes any escape characters in fields that are returned from the URL.

```

// URL encode the specified location
String applicationId = getApplicationId();
try {
    applicationId = URLEncoder.encode(applicationId, "ISO-8859-1");
} catch (UnsupportedEncodingException e) {
    if (logger.isLoggable(Level.WARNING)) {
        logger.log(Level.WARNING, "geoCoder.encodeApplicationId", e);
    }
}
throw new IllegalArgumentException(e.getMessage());
}

// URL encode the specified location
try {
    location = URLEncoder.encode(location, "ISO-8859-1");
} catch (UnsupportedEncodingException e) {
    if (logger.isLoggable(Level.WARNING)) {
        logger.log(Level.WARNING, "geoCoder.encodeLocation", e);
    }
}
throw new IllegalArgumentException(e.getMessage());
}

```

Code Sample 6: Encoding the URL Parameters

In addition to encoding the parameters and assembling the URL, the code in the server needs to make a request to the service at that URL. As [Code Sample 4](#) shows, Pet Store uses the `java.net.URL` class to open a connection to this URL and get an `InputStream` for reading from that connection. However, there are other ways to call a service. Java EE and Java SE technologies provide a variety of ways to make a call to a service. For example, you could use the Java API for XML-Based Services (JAX-WS), Java sockets, or many other technologies available to Java EE 5 developers. The mashup with the Yahoo Maps Geocoding service uses a URL and represents a relatively simple interaction. But if a service requires a more complex interaction -- for example, it requires a digital certificate -- you could use a method more suited to handling that type of request.

The Format of the Response

When you do a server-side mashup, you need to determine the format of the response that the service returns. A service can return a response in many different formats including XML, JSON, HTML, plain text, RSS/ATOM, and GData. In the Pet Store mashup with the Yahoo Maps Geocoding service, the client doesn't display the data returned by the service. It simply uses the data as input to the Google Maps service. The Yahoo service returns an XML document, as you can see in [Code Sample 2](#). Pet Store needs to convert the XML document to a Java object. As [Code Sample 5](#) shows, the results are parsed into an array of Java objects. The array is stored in a database for subsequent use when a user wants to see a map of pets for sale in a particular area. Because many public APIs provide the response in XML, the server-side code must often convert the response into another data type.

Other applications might need to convert the data returned by a service to another format such as JSON. Data in formats such as JSON can be easier for the browser to handle. By using the server as a proxy to a service, the server does the work of parsing an XML document and possibly converting it to a format such as JSON. In that case, your client JavaScript code can be simpler because it doesn't have to parse and convert the XML.

Caching of Results

Will the response from the service be used by the client, or will the data be stored on the server? In Pet Store's mashup with the Yahoo Maps Geocoding service, the data provided by the service is not sent back to the client. Instead, it is stored in a database on the server for subsequent use by the Google Maps component to pinpoint pets for sale on the map. If Pet Store did not store the results, it would have to make a request to the Geocoding service for each client request for a map and for each point shown on the map. This is a good reason for doing the mashup with the geocoding service as a server-side mashup rather than a client-side mashup.

Map data can change. For example, street addresses and street names change. So you might need a mechanism for updating data that might go out of date. It's relatively easy to write code on the server to reaccess the mashup service if you need to update the data cached from a service.

The RSS feed in Pet Store is another example of data that has been cached from a service for reuse. After data is retrieved from the live RSS feed on the Java Blueprints project site, the server side of the Pet Store application parses the RSS XML document and converts it to JSON. It then caches the data. The cached data in JSON format is used to fulfill Ajax requests from all Pet Store clients for display in the news bar.

The Need to Modify the Response

You must determine whether the server side of the application will need to modify the response from the service

before the client can use it or whether the application can directly pass it to the client. As mentioned in [Caching of Results](#), the RSS news feed data used by Pet Store is further processed before it's sent to the client.

Ways to Handle Exceptions and Errors

The `GeoCoder` proxy class handles exceptions that could occur when accessing the Yahoo Maps Geocoding service. In general, it's good practice to handle exceptions that could occur when accessing a service. Additionally, it's good practice to validate the input data to a service. For example, `GeoCoder` validates the address input string before sending it to the geocoding service. Doing this can help uncover error conditions before the call to another web site and it could help you respond more quickly to the user. For instance, you could prompt the user to correct the entry.

Also, web sites and public APIs used in mashups have very different mechanisms for responding to exception conditions. So your error-handling code can get convoluted. By using the server as a proxy to a service, you can shield the client JavaScript code from many of these details and keep it more loosely coupled. In this way, you can lessen the impact on your client code if the mashup service changes its API.

Summary

You can design a mashup in various ways. In one approach, called a server-side mashup, also known as a proxy-style mashup, you integrate services and content on the server. There are a number of good reasons for using the server-side mashup approach, not the least of which is that the approach overcomes browser security restrictions that constrain `XMLHttpRequests` and the ability to access web sites across domains.

The next article in this series focuses on client-side mashups, in which you integrate services and content in the client. You'll be able to compare and contrast the client-side and server-side mashup styles and identify which, if either, of these approaches better suits your needs.

For More Information

- [Introducing the Java Pet Store 2.0 Application](#)
- [Java EE 5 SDK](#)
- [Java BluePrints Solutions Catalog](#)
- [Java BluePrints Project](#)
- [JavaServer Faces Technology](#)
- [ProgrammableWeb](#)
- [Google Maps API](#)
- [Yahoo Maps Geocoding API](#)

About the Authors

[Ed Ort](#) is a staff member of [java.sun.com](#). He has written extensively about relational database technology, programming languages, and web services.

[Sean Brydon](#) is an engineer with Sun Microsystems, where he is the technical lead for the [Java BluePrints Program](#). He has been involved with the Java BluePrints Program since its inception. He is an author of the Addison-Wesley Java-series books, [Designing Enterprise Applications With the Java 2 Platform, Enterprise Edition](#) and [Designing Web Services With the J2EE 1.4 Platform](#). He helped create the [Java BluePrints Solution Catalog](#) and the [Java Pet Store Demo 2.0](#) reference applications.

[Mark Basler](#), a senior software engineer, is part of the Java BluePrints Program team. He helped create the [Java BluePrints Solution Catalog](#) and the [Java Pet Store Demo 2.0](#), reference applications that demonstrate how to design and develop Ajax-enabled Web 2.0 applications. His other contributions include the design and development of key components for Sun's Download Center, eCommerce suites, and Sun Java System Application Server.

[About Sun](#) | [About This Site](#) | [Newsletters](#) | [Contact Us](#) | [Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) | [Trademarks](#)

[A Sun Developer Network Site](#)



Copyright Sun Microsystems, Inc.

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this [License](#).

