

API Basics

Introducing Bing API Version 2.0

This whitepaper introduces Bing API Version 2.0 and provides the information you need to begin developing applications with the API. The paper consists of these sections:

- About Bing API Version 2.0
- Getting Started: Using Bing API from a Browser
- Incorporating Advertisements into Results
- Using the API from Programmable Environments
- Appendix: Terms of Use Overview

About Bing API Version 2.0

Bing API Version 2.0 enables you to embed a flexible and powerful search engine as a custom search component in your sites and applications. The new version includes:

- HTTP endpoints that can provide results in either XML or JSON media formats
- Enhanced support for SOAP
- The ability to monetize your applications with advertisements
- A fully [OpenSearch](#) compliant RSS interface that offers access to many of the API's information sources

Both the HTTP endpoints and the OpenSearch RSS interface are new in this version. With the addition of these interfaces and enhanced SOAP support, you are now able to choose the style of access and presentation most appropriate for your application.

Getting Started: Using Bing API from a Browser

Bing API Version 2.0 Beta's HTTP endpoint facilitates the process of using **HTTP GET** to send requests. While this is not the mechanism you're likely to use in a production environment, it is a useful way to get a sense of how the API works. Using the API from a browser can include:

- Getting an AppID
- Determining values for required parameters

- Working with optional parameters
- Choosing a protocol
- Sending a request
- Working with results
- More sample requests
- Parsing JSON results

Getting an AppID

The **AppID** parameter is a value that enables the API to validate that a request is from a registered Bing application developer.

Getting an AppID is a straightforward process. First, go to the [Bing Developer Center](#) and sign in with your Windows Live ID. After signing in, you will be presented with a link to create a new AppID. Click the link, then supply basic information about your application and review the Terms of Use. (For more information, see [Appendix: Terms of Use Overview](#).) After you have supplied the information and reviewed the Terms of Use, you will be presented with an AppID.

Determining values for required parameters

In addition to a valid **AppID**, requests to any of the API interfaces require two additional parameters: **Query** and **Sources**.

The **Query** parameter is the text of the query you want the API to execute. For the sample request, we're going to formulate, we will use the text **sushi**.

The **Sources** parameter is one or more values indicating the **SourceType** or SourceTypes from which you want to request results. The API includes a number of different SourceTypes, which is part of what makes Bing an attractive search engine for your applications. To request results from multiple SourceTypes, separate the SourceType names with a plus (+) sign.

The **Web** SourceType returns a set of results from the **Web** SourceType. In addition to the **Image** and **News** SourceTypes, with which you might be familiar from using Bing online, there are also SourceTypes such as **Spell** and **InstantAnswer**. For the release, only **InstantAnswer** using the Encarta information resource to provide answers to questions is available publicly.

Bing API Version 2.0 SourceTypes are presented in Table 1. For our sample request, we'll use the **Web** SourceType. For more information, see [Working with SourceTypes](#) on MSDN.

Table 1: Bing API Version 2.0 SourceTypes

SourceType	Description	Sample Query parameter
Web	Searches for web content	Sushi
Image	Searches for images on the web	Sushi
News	Searches news stories	Sushi
InstantAnswer	Searches Encarta online	what is sushi convert 5 feet to meters $x*5=7$ 2 plus 2
Spell	Searches Encarta dictionary for spelling suggestions	Coffee
Phonebook	Searches phonebook entries	sushi in los angeles
RelatedSearch	Returns the query strings most similar to yours	{inari sushi; sushi restaurant; California roll}
Ad	Returns advertisements to incorporate with results	Sushi

Working with optional parameters

In addition to the three required parameters documented in the previous section, each SourceType has optional parameters that you can use to refine your request. While implementing these parameters is beyond the scope of this document, you can view information about all API parameters at [API Reference](#) on MSDN.

Choosing a protocol

Bing API Version 2.0 enables you to choose the protocol you want to send a request. In this whitepaper, we will use the XML interface of the HTTP endpoint as our primary example, while also giving you an idea of what is necessary to send requests using the other protocols. For more information, see [Choosing the Appropriate Protocol](#) on MSDN.

Sending a request

A request to the HTTP endpoint consists of an **HTTP GET** request to the appropriate URI. There are two URIs, one for XML results and one for JSON results. These are **<http://api.search.live.net/xml.aspx>** and **<http://api.search.live.net/json.aspx>**, respectively.

For our sample request, we will use the XML endpoint and query the web for results on the term **sushi**: **<http://api.search.live.net/xml.aspx?Appid=<Your App ID HERE!>&query=sushi&sources=web>**.

Working with results

This section discusses basic features of Bing API Version 2.0 results sets that are common to all SourceTypes, initially in the context of our **Web** SourceType example.

Anatomy of a results set

The results returned for a request differ from SourceType to SourceType, but in every case they include a **header** and a **results body**. The document element is always **SearchResponse**, with a **Query** child element that contains the query used to produce the results. These two elements make up the header. You can see this header in Figure 1, which shows results that the request in the previous section might generate.

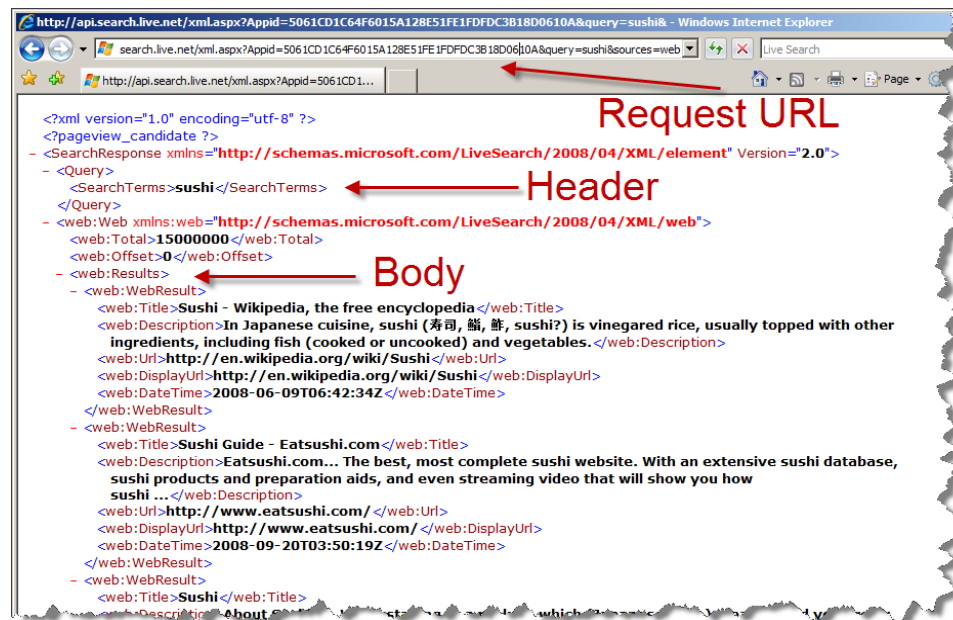


Figure 1: XML Results

The results body follows the header. If only one SourceType is used, there will be one additional child element of **SearchResponse** after the **Query** element, but there can be more than one if multiple SourceTypes are used. Each SourceType has its own XML namespace URI (and, therefore, its own schema), but all SourceTypes share a common structure. In Figure 1, because the request was to the **Web** SourceType, the body is an element named **Web** from the **http://schemas.microsoft.com/LiveSearch/2008/04/XML/web** namespace.

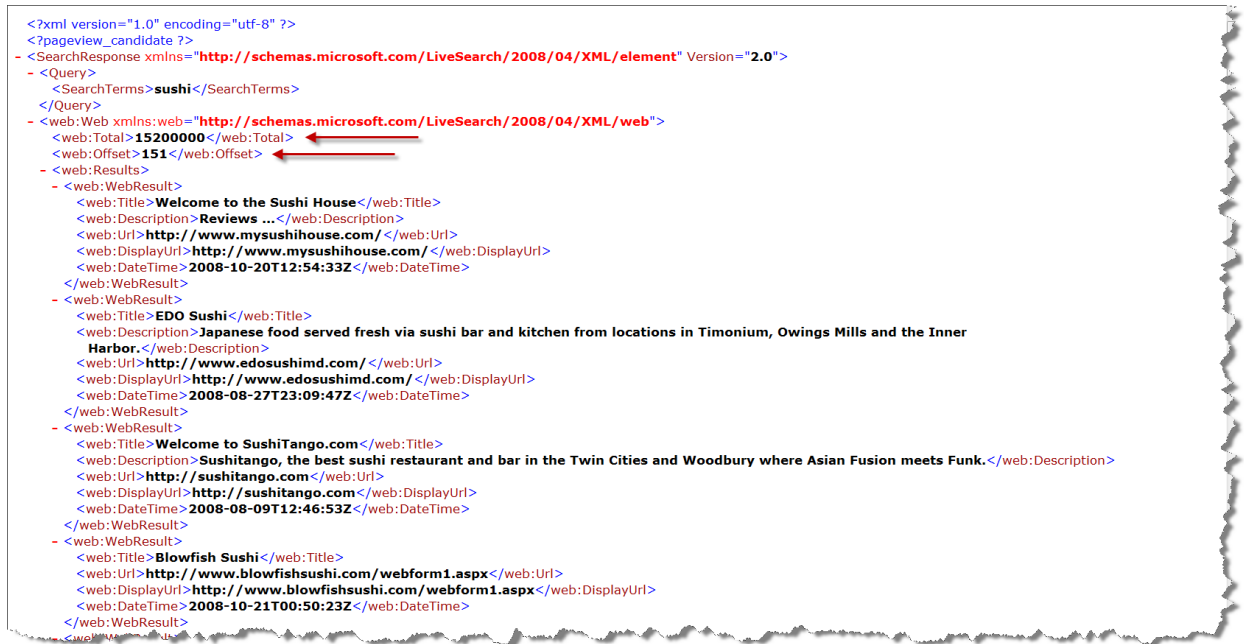
Working with Total, Offset, and Count

The **Web** element subordinates two elements that are qualified by the web namespace, but are common to all SourceTypes: **Total** and **Offset**. The **Total** element contains the estimated¹ number of results for the request in that particular SourceType, while **Offset** indicates how far into the result set you are currently processing. Each SourceType has a default number of results, which can be modified using the optional **Count** parameter. You can change **Offset** using the optional **Offset** parameter. Since

¹ Depending on how popular a query is, the estimated number of results could be very different from the real number. Do not rely on this number for critical computation.

you might be using multiple SourceTypes, both **Count** and **Offset** need to be qualified by the SourceType.

For example, if you wanted to ask for 40 results at a time from the **Web** SourceType, you would pass **web.count=40** as part of the query string. If you wanted to get the next 40 results after getting the first results, you would pass **web.offset=41**. The full URI would be **http://api.search.live.net/xml.aspx?Appid=<AppID>&query=sushi&sources=web&web.count=40&web.offset=41**. Figure 2 display results that this request might return.



```
<?xml version="1.0" encoding="utf-8" ?>
<?pageview_candidate ?>
- <SearchResponse xmlns="http://schemas.microsoft.com/LiveSearch/2008/04/XML/element" Version="2.0">
  - <Query>
    <SearchTerms>sushi</SearchTerms>
  </Query>
  - <web:Web xmlns:web="http://schemas.microsoft.com/LiveSearch/2008/04/XML/web">
    <web:Total>15200000</web:Total>
    <web:Offset>151</web:Offset>
    - <web:Results>
      - <web:WebResult>
        <web:Title>Welcome to the Sushi House</web:Title>
        <web:Description>Reviews ...</web:Description>
        <web:Url>http://www.mysushihouse.com/</web:Url>
        <web:DisplayUrl>http://www.mysushihouse.com/</web:DisplayUrl>
        <web:DateTime>2008-10-20T12:54:33Z</web:DateTime>
      </web:WebResult>
      - <web:WebResult>
        <web:Title>EDO Sushi</web:Title>
        <web:Description>Japanese food served fresh via sushi bar and kitchen from locations in Timonium, Owings Mills and the Inner Harbor.</web:Description>
        <web:Url>http://www.edosushimd.com/</web:Url>
        <web:DisplayUrl>http://www.edosushimd.com/</web:DisplayUrl>
        <web:DateTime>2008-08-27T23:09:47Z</web:DateTime>
      </web:WebResult>
      - <web:WebResult>
        <web:Title>Welcome to SushiTango.com</web:Title>
        <web:Description>Sushitango, the best sushi restaurant and bar in the Twin Cities and Woodbury where Asian Fusion meets Funk.</web:Description>
        <web:Url>http://sushitango.com/</web:Url>
        <web:DisplayUrl>http://sushitango.com/</web:DisplayUrl>
        <web:DateTime>2008-08-09T12:46:53Z</web:DateTime>
      </web:WebResult>
      - <web:WebResult>
        <web:Title>Blowfish Sushi</web:Title>
        <web:Url>http://www.blowfishsushi.com/webform1.aspx</web:Url>
        <web:DisplayUrl>http://www.blowfishsushi.com/webform1.aspx</web:DisplayUrl>
        <web:DateTime>2008-10-21T00:50:23Z</web:DateTime>
      </web:WebResult>
    </web:Results>
  </web:Web>
</SearchResponse>
```

Figure 2: Web Results Using Count and Offset

Working with results from multiple SourceTypes

Remember that the prefixing of the **Count** and **Offset** parameters is necessary because you can specify multiple SourceTypes. The URL to make a request for both the **Web** and **Image** SourceTypes might look like this: **http://api.search.live.net/xml.aspx?Appid=<AppID>&query=sushi&sources=web+image**.

Figure 3 shows results that this request might return.

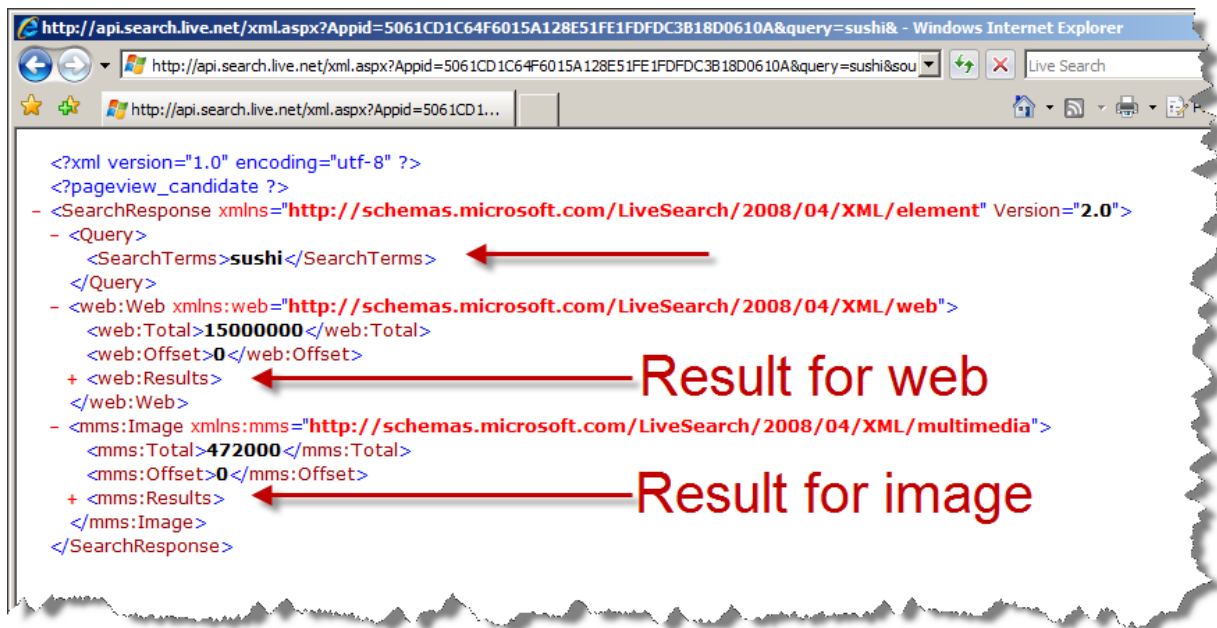


Figure 3: Multiple SourceTypes Results

Note that the **Image** element has its own namespace URI. It also includes **Total** and **Offset** child elements, but qualified with the **mms** (multimedia) namespace prefix. The third child element of any XML results set is an element with a local name of **Results** – again, qualified by its parent namespace URI.

When you are processing the results of a multiple-SourceTypes request, you must determine the order of results dynamically; SourceTypes are not necessarily returned in a particular order.

The **Results** element is the parent element for a number of child elements, each of which represents a particular result from a particular SourceType. These child elements all have SourceType-specific names, such as **WebResult** and **ImageResult**. You can parse these results using the XML processing API of your choice (which will be determined largely by your programming language and runtime choice).

More sample requests

This section contains queries for each Bing SourceType, which you can use to experiment with different Bing result sets. To do so, just substitute your AppID for <AppID>.

http://api.search.live.net/xml.aspx?Appid=<AppID>&query=sushi&sources=web

**http://api.search.live.net/xml.aspx?Appid=<AppID>&query=sushi&sources=ima
ge**

**http://api.search.live.net/xml.aspx?Appid=<AppID>&query=sushi&sources=new
s**

**http://api.search.live.net/xml.aspx?Appid=<AppID>&sources=instantanswer&qu
ery=what is sushi**

http://api.search.live.net/xml.aspx?Appid=<AppID>&sources=instantanswer&query=convert 5 feet to meters

http://api.search.live.net/xml.aspx?Appid=<AppID>&sources=instantanswer&query= x*5=7

http://api.search.live.net/xml.aspx?Appid=<AppID>&sources=instantanswer&query=2 plus 2

http://api.search.live.net/xml.aspx?Appid=<AppID>&sources=spell&query=cofee

http://api.search.live.net/xml.aspx?Appid=<AppID>&sources=phonebook&query=sushi in los angeles

http://api.search.live.net/xml.aspx?Appid=<AppID>&sources=relatedqueries&query=sushi

All the examples we've shown so far use a browser (Internet Explorer) to send a request. Of course, being able to test queries with a browser is a useful feature of the XML API, but in general a programmable environment is the desirable mechanism for invoking the service.

The steps we performed in working with our browser-centric XML example are the same steps you'd perform when using the XML API from the program language of your choice: Building a request based on the SourceTypes and query you want to run, sending the request, and parsing the response.

Parsing JSON results

As aforementioned, Bing API Version 2.0 can return results in multiple media types. Besides XML, the other media type you can request is application/json - JavaScript Object Notation (JSON).

JSON is a serialization format for data that has become popular for two reasons:

- JSON has a smaller serialization size than XML and represents a saving of network bandwidth, as well as of serialization/deserialization costs.
- Second, JSON is a more natural format than XML with which to work when building AJAX applications using JavaScript. (This is because XML processing in most JavaScript implementations isn't very sophisticated).

The principles discussed in the previous sections with respect to sending a request to the XML interface are all applicable to sending a request to the JSON interface. However, processing the results from the JSON interface is different than processing results the XML interface returns.

The same main query we've used in previous sections, when sent to the JSON interface, might produce the results in Example 1.

Example 1: JSON Serialized Response

```
"SearchResponse":{ "Version":"2.0","Query":{  
  "SearchTerms":"sushi"},"Web":{ "Total":15000000,"Offset":0,"Results":[  
  { "Title":"Sushi - Wikipedia, the free encyclopedia","Description":"In  
Japanese cuisine, sushi (寿司, 鮓, 鮓, sushi?) is vinegared rice, usually  
topped with other ingredients, including fish (cooked or uncooked) and  
vegetables.", "Url":"http://en.wikipedia.org/wiki/Sushi","DisplayUrl  
":"http://en.wikipedia.org/wiki/Sushi","DateTime":"2008-06-  
09T06:42:34Z"}}]} /* pageview_candidate */}
```

The full URL for this query would be

<http://api.search.live.net/json.aspx?Appid=<AppID>&query=sushi&sources=web>.

This would be the query we'd run if we wanted to process the JSON-encoded result from a program that was not running inside of a browser (for an example of this scenario, see [Using the API with PHP](#)). In many cases, when we use the JSON interface, we want the query to be executed from a browser and the results displayed for the use of an AJAX application (running in a browser). Note that JSON serialization is supported by other runtimes and languages, such as PHP and Ruby.

In order to get the JSON result into the browser, we are going to have to add a script tag into our HTML page dynamically. This is necessary because browsers will not let you use JSON results from another website due to security restrictions. AJAX applications can only invoke URLs from the site from which the AJAX application was downloaded. Because our AJAX application can't be uploaded to <http://api.search.live.net>, we must use this alternate mechanism to use the JSON API. We will add a **script** tag to the page dynamically using **JSONP**.

JSONP is a technique for injecting JavaScript from another website into a browser. Adding a script element (with its **src** attribute pointing to the **json.aspx** URL) causes the browser to send a request to Bing. Bing will return JavaScript dynamically to the page. The JavaScript injected into the page will call a method with the JSON-encoded results, the name of the method having been passed as the value of the **JSONCallback** query string parameter. The **JsonType** query string parameter is also passed in this case with the value of **callback**.

Assuming the correct HTML page is running in the browser with an **input** element of type **text** that has an id attribute value of **searchText**, and a **ul** element that has an id attribute value of **resultList**, the JavaScript in Example 2 would dynamically add a script tag to the head tag (the proper place for a script tag). This will cause the browser to, on demand, request the Bing API Version 2.0 JSON interface. The **searchDone** function is the name passed as the **JSONCallback** parameter. Example 2 shows how this JavaScript might look.

Example 2: Brower JavaScript code for JSONP

```
function search() {  
    var search = "&query=" +  
document.getElementById("searchText").value;  
    var fullUri = serviceURI + AppId + search;  
    var head = document.getElementsByTagName('head');
```



```

        var script = document.createElement('script');
        script.type = "text/javascript";
        script.src = fullUri;
        head[0].appendChild(script);
    }
    function searchDone(results) {
        var result = null;
        var parent = document.getElementById('resultList');
        parent.innerHTML = '';
        var child = null;
        for (var i = 0; i <
results.SearchResponse.Image.Results.length; i++) {
            result = results.SearchResponse.Image.Results[i];
            child = document.createElement('li');
            child.className = "resultlistitem";
            child.innerHTML = '<a href="' + result.Url + '"></a>';
            parent.appendChild(child);
        }
    }
}

```

Example 3: JSONP Callback Code

```

if(typeof searchDone == 'function') searchDone({ "SearchResponse":{
"Version":"2.0","Query":{"SearchTerms":"sushi"},"Web":{"
"Total":15000000,"Offset":0,"Results":[ { "Title":"Sushi - Wikipedia,
the free encyclopedia","Description":"In Japanese cuisine, sushi (寿司,
鮓, 鮓, sushi?) is vinegared rice, usually topped with other
ingredients, including fish (cooked or uncooked) and
vegetables.", "Url":"http://en.wikipedia.org/wiki/Sushi","DisplayUrl
":"http://en.wikipedia.org/wiki/Sushi","DateTime":"2008-06-
09T06:42:34Z"}]}} /* pageview_candidate */});

```

The full code for this page is in Example 4.

Example 4: Full JSONP HTML Page

```

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>

    <link href="styles.css" rel="stylesheet" type="text/css" />
<title>Using Bing and JSON</title>
<script type="text/javascript">
    function search() {
        var search = "&query=" + document.getElementById("searchText").value;
        var fullUri = serviceURI + AppId + search;
        var head = document.getElementsByTagName('head');
        var script = document.createElement('script');

```

```

        script.type = "text/javascript";
        script.src = fullUri;
        head[0].appendChild(script);
    }
    function searchDone(results) {
        var result = null;
        var parent = document.getElementById('resultList');
        parent.innerHTML = '';
        var child = null;
        for (var i = 0; i < results.SearchResponse.Image.Results.length; i++) {
            result = results.SearchResponse.Image.Results[i];
            child = document.createElement('li');
            child.className = "resultlistitem";
            child.innerHTML = '<a href="' + result.Url + '"></a>';
            parent.appendChild(child);
        }
    }

    var AppId = "&Appid=YOUR APPID HERE!";
    var serviceURI =
"http://api.search.live.net/json.aspx?JsonType=callback&JsonCallback=searchDone&source=
image";

    </script>
</head>
<body>
    Type in a search:<input type="text" id="searchText" value="sushi"/>
        <input type="button" value="Search!" id="searchButton" onclick="search()" />
        <ul ID="resultList">

            </ul>

</body>
</html>

```

Incorporating Advertisements into Results

Bing API Version 2.0 provides you with the ability to incorporate advertisements into your search results. To take advantage of this ability, perform the procedure at [Monetize your search application](#). Creating a request to the **Ad** SourceType is much the same as creating a request to any Bing API SourceType, although there are a number of additional required parameters. These parameters are identified in Table 2.

Table 2: Ad SourceType Additional Required Parameters

Parameter	Description
Ad.AdUnitId	Your AdCenter Unit ID
Ad.PropertyId	Your AdCenter Property ID
Ad.Pagenumber	The page number requested

Figure 4 shows results returned by a query to the **Ad SourceType**.

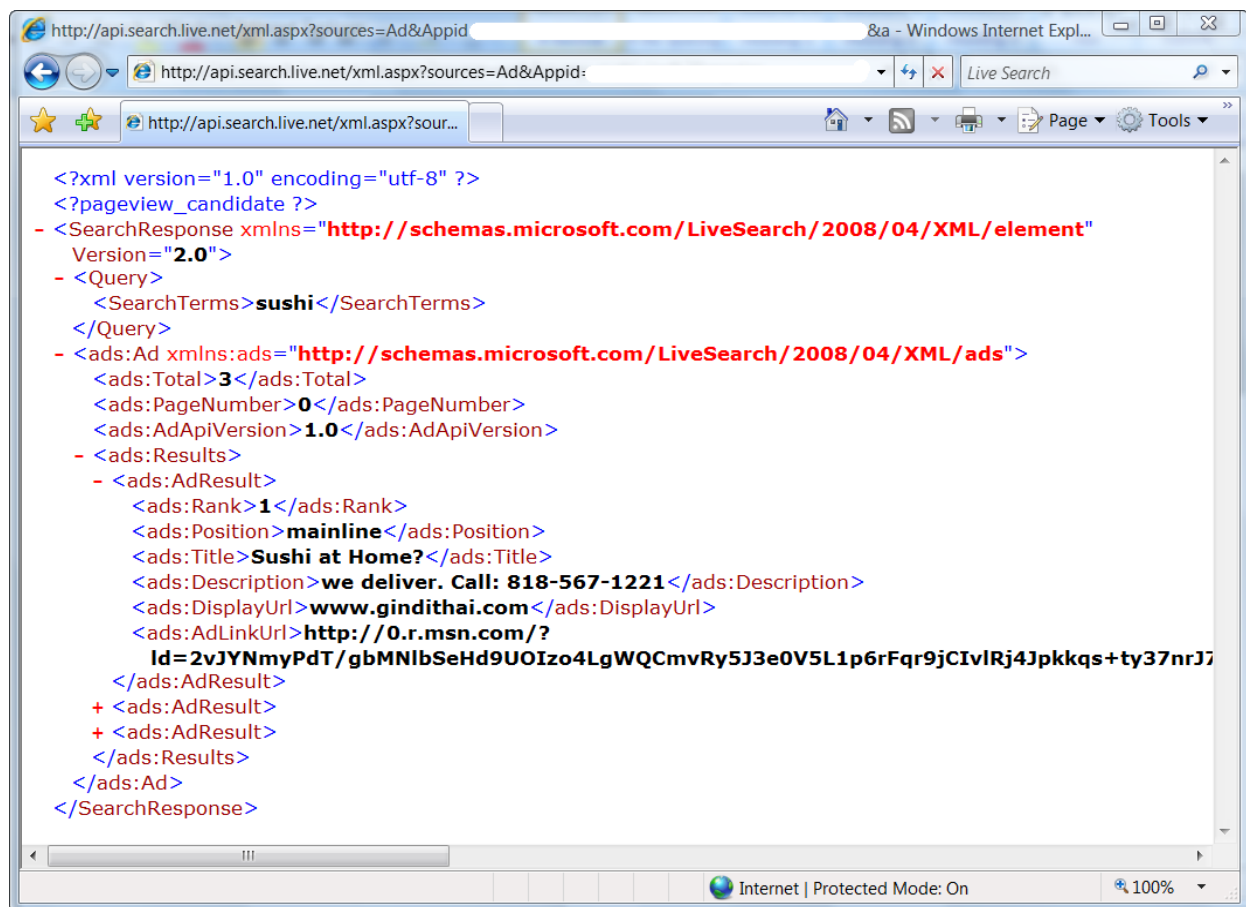


Figure 4: Bing API Version 2.0 Ad SourceType Response

As Figure 4 shows, Ad SourceType result sets are shaped exactly the same as other SourceTypes, with **SearchResponse** followed by the **Ad** element, and the **Ad** element having a number of **AdResult** elements. The data inside of **AdResults** can be used to create advertisements on your page; when a user clicks on that ad, your account at AdCenter is incremented.

- **Note:** If you are going to use the **Ad** SourceType, you must also send the search client IP address in a special custom HTTP header as well as the Client user agent. Specifically, with each search request

you send a special custom HTTP header: "search-client-ip". The value of this header will be the IP address of the machine sending the request. (That is, the end user accessing your web site). You will also need to send the user agent information. (HTTP Header webRequest.Headers.Add("search-client-ip", "1.2.3.4"); User Agent request.UserAgent ="Your user agent";)

Using the API from Programmable Environments

All the examples we've shown so far use a browser (Internet Explorer) to send a request. Of course, being able to test queries with a browser is a useful feature of Bing API Version 2.0 Beta, but in general a programmable environment is the desirable environment for invoking the service.

The steps we performed in working with our browser-centric XML example are the same steps you'd perform when using the XML API from the programming language of your choice: Building a request based on the SourceTypes and query you want to run, sending the request, and parsing the response. In the next section, we'll see how to send a request to the XML interface from .NET.

In subsequent sections, we'll see how to use Bing Version 2.0 with:

- Microsoft.Net Framework (with both the [XML](#) and [SOAP](#) interfaces)
- [PHP](#)
- [Silverlight](#)
- [RSS](#)

Using the API with Microsoft .Net Framework using the XML interface

The most straightforward way to use the XML API from .NET code is to use the **HttpWebRequest** class found in the **System.Net** namespace. This class enables you to make arbitrary **HTTP GET** requests, an ability that you need in order to invoke the XML interface. The code in Example 5 shows a simple usage of the XML API using **HttpWebRequest**.

Example 5: Invoking the XML Interface

```
string url =  
"http://api.search.live.net/xml.aspx?Appid={0}&sources={1}&query={2}";  
string completeUri = String.Format(url, AppId, "image", "sushi");  
HttpWebRequest webRequest = null;  
webRequest = (HttpWebRequest)WebRequest.Create(completeUri);  
HttpWebResponse webResponse = null;  
webResponse = (HttpWebResponse)webRequest.GetResponse();  
XmlReader xmlReader = null;  
xmlReader = XmlReader.Create(webResponse.GetResponseStream());
```

When using **HttpWebRequest**, you generally follow the pattern shown in Example 5. Passing the static **WebRequest.Create** method a valid URL returns an **HttpWebRequest** instance. In this code, we are

creating the URL by using **String.Format**, passing in the **SourceType** and the **Query** as well as the **AppId**. By calling **HttpWebRequest.GetResponse**, the HTTP request is made to the Bing API XML interface, which returns results once the HTTP request has completed. We can then take the stream and use the XML processing engine of your choice to process it. In this case, we've chosen **XmlReader**.

What you do with the results next is dependent on the XML processing code you'd like to write. On the .NET platform the choices are to continue to use the **XmlReader**, turn the **XmlReader** into an **XmlDocument**, or use **XLINQ**. If, for example, you wanted to use the results from your query and data bind the results to an **ASP.NET**, **WebForms**, or **WPF** control, you might choose to use **XLINQ** to query the results and project them into a collection of strongly-typed objects that could easily be used for data binding, as shown in Example 6.

Example 6: Processing XML API Image SourceType Results Using XLINQ

```
XDocument data = XDocument.Load(xmlReader);
IEnumerable<XNode> nodes = null;
nodes = data.Descendants(XName.Get("Results", IMAGE_NS)).Nodes();
if (nodes.Count() > 0)
{
    var results = from uris in nodes
                  select new LiveSearchResultImage
                  {
                      URI =
((XElement)uris).Element(XName.Get("Url", IMAGE_NS)).Value,
                      Title =
((XElement)uris).Element(XName.Get("Title", IMAGE_NS)).Value,
                      ThumbnailURI =
((XElement)uris).Element(XName.Get("Thumbnail", IMAGE_NS)).Value,
                  };
    return results;
}
```

Using **XLINQ** is fairly easy, as we can use the **XDocument.Descendants** method to get at the **Result** element (again, note that the **Result** element is namespace qualified, as we have the Image XML namespace URI as a constant **IMAGE_NS** already defined in our code). In this case, we are projecting each XML **ImageResult** element (the child nodes under the **Result** element) into a .NET type we've defined called **LiveSearchResultImage** (see Example 7).

Example 7: LiveSearchResultImage Type Definition

```
public class LiveSearchResultImage
{
    public string Title { get; set; }
    public string Description { get; set; }
    public string URI { get; set; }
}
```

```

        public string ImageURI { get; set; }
        public string ThumbnailURI { get; set; }
    }

```

Regardless of the XML processing API you choose, processing the child elements under the **Results** element is the primary goal.

Using the API with Microsoft .Net Framework using SOAP

This version of the API offers enhanced support for SOAP through stronger typing. The previous version had a single SearchRequest object that included all request parameters and a single SearchResponse object that included all response fields. The fields in the response were populated (or not) based on the SourceTypes in the request. In this version, every SourceType has its own request and response objects.

Note: The changes that created stronger typing in SOAP were made in the underlying object model. Hence, the benefits of strong typing are present in all interfaces, not just SOAP.

The advantage of using the SOAP interface over the XML interface from statically-typed languages like C# and VB.NET is that there is less XML processing to do, since the web services infrastructure in .NET handles deserializing the SOAP XML responses from the SOAP interface into strongly-typed .NET collections.

To use the SOAP interface from a .NET project, you can use the **Add Web Reference** functionality in Visual Studio to consume the Bing WSDL found at <http://api.search.live.net/search.wsdl?AppID=<YourAppIDHere>>. For more information, see [Using SOAP](#).

Example 8 shows typical .NET code using the SOAP API (after the web service reference has been added).

Example 8: Using the Bing SOAP interface

```

LiveSearchService soapClient = new LiveSearchService();
SearchRequest request = new SearchRequest();
request.AppId = AppId;
request.Sources = new SourceType[] { SourceType.Image };
request.Query = query;
SearchResponse response = soapClient.Search(request);
if (response.Image != null && response.Image.Results.Count() > 0)
{
    var results = from uris in response.Image.Results
                  select new LiveSearchResultImage
                  {
                      URI = uris.Url,
                      Title = uris.Title,
                      ThumbnailURI = uris.Thumbnail.Url,
                  };
}

```

```

        return results;
    }

```

In Example 8, we continue to use the **LiveSearchResultImage** to simplify our data binding code, but notice that we can use LINQ to Objects instead of XLINQ (which saves us the burden of the XML processing).

Instead of passing the sources as a query string parameter, there is a strongly-typed enumeration, **SourceType**, which is passed as an array on the **SearchRequest** object. **SearchRequest** represents the parameters that, in Getting Started: Using Bing API from a Browser, we passed as query string parameters to the HTTP endpoint. **SearchRequest.Query** represents the query we'd like to run. If you want to set parameters like **Offset** or **Count**, there are strongly-typed properties on **SearchRequest** for doing this for different SourceTypes. For example, to change the **Offset** for this search (using the **image** SourceType), we'd create an instance of **ImageRequest**, set its **Offset** property to the appropriate value, and set the **SearchRequest.ImageRequest** on our **SearchRequest** instance to the new instance of **ImageRequest**.

Using the API with PHP

PHP has a powerful JSON parsing mechanism, which, because PHP is a dynamic language, enables PHP developers to program against a JSON object graph in a very straightforward way. Example 9 shows a PHP page that sends a request to the JSON interface using the **file_get_contents** API to call the JSON endpoint, and the **json_decode** function to turn the results into an object graph that can be walked and turned into HTML.

Example 9: Using the API with PHP

```

<html>
<head>
    <link href="styles.css" rel="stylesheet" type="text/css" />
    <title>PHP Bing</title>
</head>
<body><form method="post" action="<?php echo $PHP_SELF;?>">
    Type in a search:<input type="text" id="searchText" name="searchText"
value="<?php
    if (isset($_POST['searchText'])) {
        echo($_POST['searchText']); }
    else { echo('sushi'); }
?>" />
        <input type="submit" value="Search!" name="submit"
id="searchButton" />
<?php
if (isset($_POST['submit'])) {
$request =
'http://api.search.live.net/json.aspx?Appid=<YourAppIDHere>&sources=im
age&query=' . urlencode( $_POST["searchText"]);

```

```

$response = file_get_contents($request);
$jsonobj = json_decode($response);
echo('<ul ID="resultList">');

foreach($jsonobj->SearchResponse->Image->Results as $value)
{
echo('<li class="resultlistitem"><a href="' . $value->Url . '">');
echo('</li>');
}
echo("&</ul>");
} ?>
</form>
</body>
</html>

```

Using the API with Silverlight

Silverlight is a Microsoft framework that enables developers to build Rich Internet Applications (RIAs) using the same .NET framework and programming languages they are used to using in typical .NET applications. Because of its strong integration with browsers as well as with images and other media, it is a very interesting client for building applications using Bing API Version 2.0 Beta.

Like JSON applications that live in the browser, Silverlight applications can't typically invoke services (raw HTTP or SOAP) on domains other than the domain from which the Silverlight application is downloaded. But because Bing API Version 2.0 publishes a crossdomain policy file at the root of its domain (<http://api.search.live.net/crossdomain.xml>) that allows RIAs like Silverlight to access its endpoints, it is possible to use the API from Silverlight.

Because Silverlight is really a .NET programming environment, the choices for API invocation are the same as the earlier .NET examples (although Silverlight also has a JSON-parsing object model). You can either use the raw HTTP XML interface or use the SOAP interface. The only difference is that, because Silverlight runs in the browser, the APIs we use are all asynchronous rather than synchronous. Example 10 shows code running inside of a Silverlight application that calls the HTTP XML interface.

Example 10: Using the HTTP XML Interface from Silverlight

```

WebClient wc = new WebClient();
wc.OpenReadCompleted += new
OpenReadCompletedEventHandler(wc_OpenReadCompleted);
item.CurrentQuery = p;
item.Uri =
    new Uri(String.Format(_baseURI, AppId, "image", "sushi"));
wc.OpenReadAsync(item.Uri, item);

```


The difference between the .Net code show in Example 5 and the .Net code in Example 10 is that we are using the Silverlight **WebClient** type to simplify the HTTP programming model even further. To get the results, we need to set the **WebClient.OpenReadCompleted** delegate. When the request is finished, the **WebClient** will call our **wc_OpenReadCompleted** method. Example 11 shows this method.

Example 11: WebClient callback

```
void wc_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    Stream streamResult = e.Result;
    XmlDocument xd = XmlDocument.Load(XmlReader.Create(streamResult));
    //use the XmlDocument
}
```

In Example 11 we are again using **XmlDocument**, part of the XLINQ programming model, which is available in Silverlight. This allows us to build a collection of strongly-typed objects to data bind against a Silverlight control such as is seen in Figure 5.



Figure 5: Silverlight Application Integrated with Bing

Accessing Search through RSS

In addition to the XML, JSON and SOAP interfaces, Bing API Version 2.0 SourceTypes are exposed via a Web Feed mechanism formatted using Really Simple Syndication (RSS). The RSS endpoint is anonymous; it doesn't require an AppID. Any user can make a feed request of the API and subscribe to that feed using his favorite feed reader. The feed reader can then take care of maintaining state, and the user will be notified whenever there is new data.

This is the only part of the new Bing API that can be used by end users, since it is the only URI that doesn't require an AppID as part of the query string. For example, we can ask for this URI:

<http://api.search.live.com/rss.aspx?source=web&query=sushi+los%20angeles>

Please note that you can only query one SourceType at a time in RSS and that the parameter is consequently named "source" not "sources" as in all other cases.

This URI asks for all **Web** SourceType results for the search term **sushi+los%20angeles**. The resulting RSS feed can be viewed in any feed reader. Internet Explorer has its own support for feeds, so requesting that URI in IE might result in the image in Figure 6.

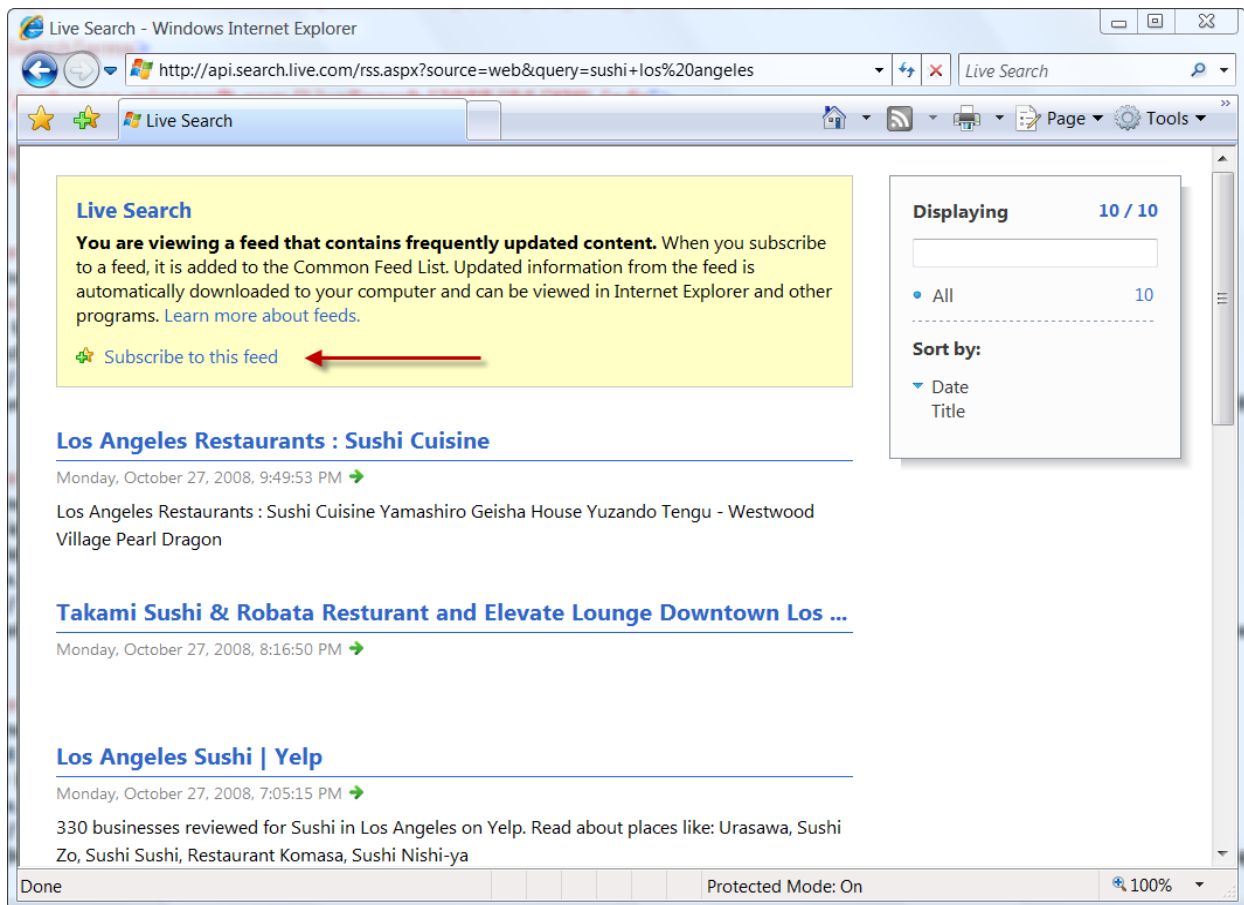


Figure 6: Bing Feed in Internet Explorer

The red arrow in Figure 6 points to the hyperlink a user can click on to add this feed to the Internet Explorer feed reader. Figure 7 shows the feed side bar once this feed has been added to the Internet Explorer feed reader.

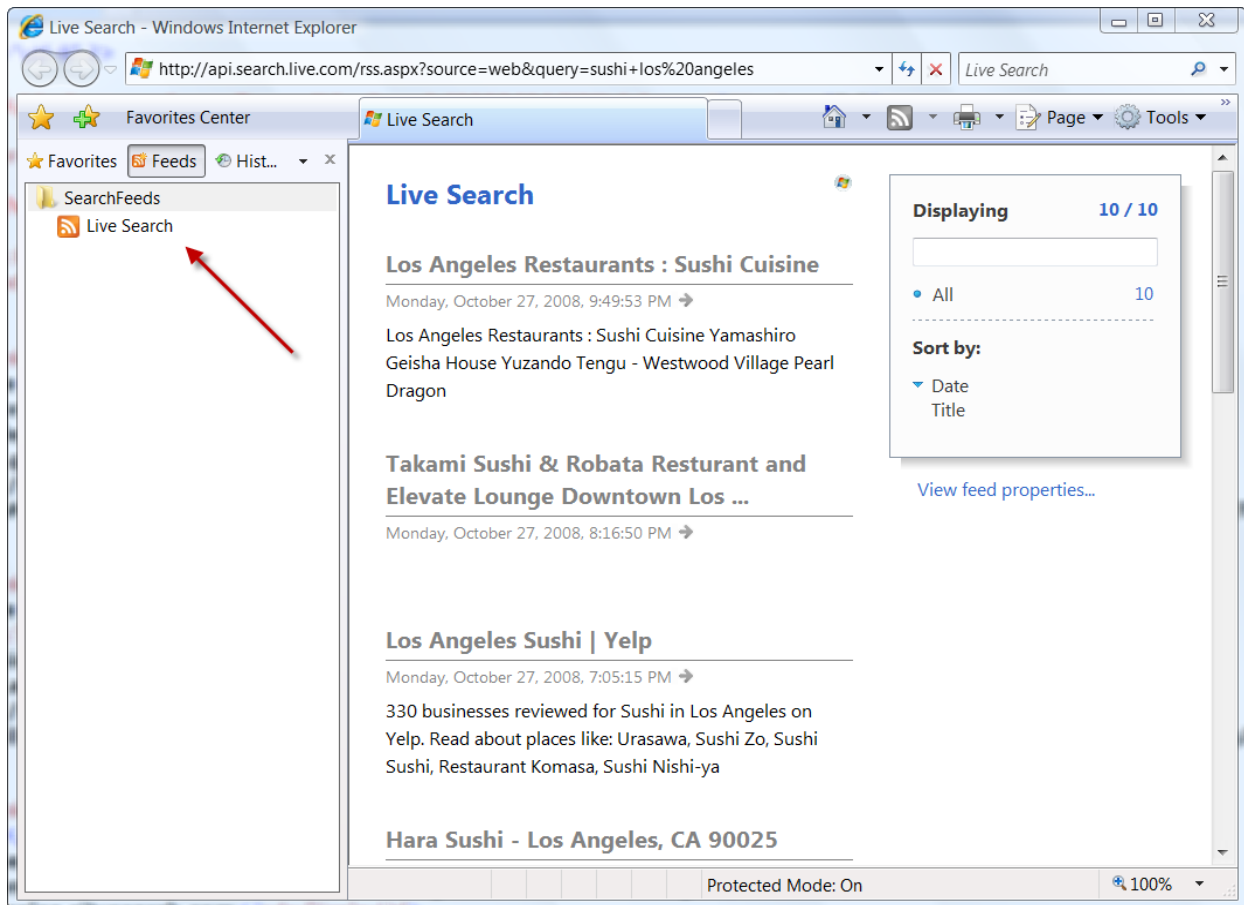


Figure 7: Feed added to Internet Explorer

When a user opens her browser, all the feeds that have new data will be bolded (this is the feed notification feature in Internet Explorer; each feed reader has its own functionality). The feed reader will have already downloaded the new feed data, so the user can just view it. This is the power of feeds as an active polling mechanism for new data.

It should also be noted that the RSS format of Bing is fully Open Search 1.1 compliant; see <http://www.opensearch.org/> for more information.

Appendix: Terms of Use Overview

The [Bing API Version 2.0 Terms of Use](#) is a contract between you, as an application developer using the API, and Microsoft. You must review and agree to the contract prior to obtaining an Application ID and beginning use of the API. This appendix provides an overview of the contract's requirements and prohibitions – that is, in a nutshell, what, as an application developer using the API, you must do and what you cannot do (besides the obvious "you are not going to use the API to plan a terrorist attack or run a drug smuggling ring" that our lawyers love so much).

What you must do

- Display all the results you request.
- Display your results in the context of a user-facing application or website.
- Display attribution to Bing in a manner compliant with our branding rules. Currently, you may determine the specific manner in which you display attribution. A link to <http://www.live.com> with the query echo is a suggested example.
- Restrict your usage to less than 7 queries per second (QPS) per IP address. You may be permitted to exceed this limit under some conditions, but this must be approved through discussion with api_tou@microsoft.com.
- If you interleave data from any source other than the API with data from the API, clearly differentiate the respective sources.

What you cannot do

- Use API results for search engine optimization (SEO). In particular, using the API for rank checks is explicitly prohibited.
- Display advertisements in positions other than the mainline and sidebar.
- Change the order of the results the API returns from a SourceType other than the **Web** SourceType.