

---

# YQL Guide

## Abstract

This guide describes the Yahoo! Query Language (YQL) Web Service, which enables you to access Internet data with SQL-like commands. This guide is for software developers who are familiar with Web applications that call Web services to retrieve data in XML or JSON format. Experience with SQL, MySQL, or Yahoo! Pipes is helpful, but not required.

For an overview of the features of the latest YQL release, refer to the [YQL Release Notes](#).<sup>1</sup>

We welcome your feedback. Have a comment or question about this document? Let us know in the [YDN Forum for Y!OS Documentation](#).<sup>2</sup>

---

<sup>1</sup> <http://developer.yahoo.com/yql/releasenotes/>

<sup>2</sup> <http://developer.yahoo.net/forum/index.php?showforum=64>

---

---

# Table of Contents

Introducing YQL .....	1
1. Overview .....	1
What is YQL? .....	1
Why Use YQL? .....	1
Usage Information and Limits .....	2
Limiting Access to Content Provider Data .....	2
Blocking HTML Data Scraping from YQL .....	2
Blocking Non-HTML from YQL .....	3
Rate Limiting by IP Address .....	3
Internationalization Support .....	4
Character Encoding .....	4
2. YQL Tutorials and Code Examples .....	5
Introduction .....	5
Prerequisites .....	5
YQL Tutorials .....	5
First YQL Application .....	6
Creating YQL Open Data Tables .....	9
Executable JavaScript in Open Data Tables .....	12
YQL Code Examples .....	18
Prerequisites .....	19
Making YQL Queries with JavaScript .....	19
Making YQL Queries with PHP .....	21
YQL Social Application .....	22
YQL INSERT: WordPress Open Application .....	27
Getting Updates with YQL .....	30
Data Scraping with YQL .....	32
Sample Open Data Tables .....	34
YQL Screencasts .....	34
YQL Slideshows .....	35
Additional Tutorials and Code Examples .....	35
Additional References and Resources .....	36
Using YQL and Open Data Tables .....	1
1. Overview for YQL Users .....	1
How to Run YQL Statements .....	1
The Two-Minute Tutorial .....	1
YQL Web Service URLs .....	2
YQL Query Parameters .....	3
YQL Query Aliases .....	3
Summary of YQL Statements .....	5
Authorization .....	5
Best Practices for Forming YQL Statements .....	6
2. SELECT Statement .....	7
Introduction to SELECT .....	7
Syntax of SELECT .....	7
Specifying the Elements Returned (Projection) .....	8
Filtering Query Results (WHERE) .....	8
Remote Filters .....	9
Local Filters .....	10
Combining Filter Expressions (AND, OR) .....	11
Joining Tables With Sub-selects .....	11
Paging and Table Limits .....	12

Remote Limits .....	12
Local Limits .....	12
Sort and Other Functions .....	13
Remote and Local Processing .....	14
Example With Remote and Local Steps .....	14
Summary of Remote and Local Controls .....	15
GUIDs, Social, and Me .....	16
Variable Substitution in the GET Query String .....	16
Extracting HTML .....	16
Parsing HTML 4.01 Versus HTML5 .....	17
Using XPath .....	17
Selecting Content .....	17
3. Using the Query Builder .....	19
YQL Query Builder Overview .....	19
Query Builder Usage .....	20
Loading a Query .....	20
Entering Values .....	21
4. INSERT, UPDATE, and DELETE (I/U/D) Statements .....	22
Introduction to INSERT, UPDATE, and DELETE (I/U/D) Statements .....	22
Bindings Required for I/U/D .....	22
JavaScript Methods Available to I/U/D .....	24
Syntax of I/U/D .....	24
Syntax of INSERT .....	24
Syntax of UPDATE .....	24
Syntax of DELETE .....	25
Limitations of I/U/D .....	25
Open Data Table Examples of I/U/D .....	26
Bit.ly Shorten URL (INSERT) .....	26
WordPress Post (INSERT) .....	27
5. Response Data .....	36
Supported Response Formats .....	36
JSONP-X: JSON envelope with XML content .....	36
Structure of Response .....	38
Information about the YQL Call .....	38
XML-to-JSON Transformation .....	39
JSON-to-JSON Transformation .....	40
Errors and HTTP Response Codes .....	41
6. Using YQL Open Data Tables .....	43
Overview of Open Data Tables .....	43
Invoking an Open Data Table Definition within YQL .....	43
Invoking a Single Open Data Table .....	43
Invoking Multiple Open Data Tables .....	44
Invoking Multiple Open Data Tables as an Environment .....	44
Working with Nested Environment Files .....	45
Setting Key Values for Open Data Tables .....	46
Using SET to Hide Key Values or Data .....	47
Creating YQL Open Data Tables .....	1
1. Creating YQL Open Data Tables .....	1
Before You Begin .....	1
Open Data Tables Reference .....	1
table element .....	2
meta element .....	2
select / insert / update / delete elements .....	3
function Element .....	4

url element .....	5
execute element .....	5
key / value / map elements .....	6
paging element .....	9
YQL Streaming .....	11
Configuring Open Data Tables to Use Streaming .....	11
Using Open Data Tables Configured for Streaming .....	12
Debugging Open Data Tables and YQL Network Calls .....	12
Enabling Logging .....	13
Viewing Logs .....	13
Open Data Table Examples .....	14
Flickr Photo Search .....	14
Digg Events via Gnip .....	16
Open Data Tables Security and Access Control .....	17
Batching Multiple Calls into a Single Request .....	17
Troubleshooting .....	18
2. Executing JavaScript in Open Data Tables .....	19
Introduction .....	19
Features and Benefits .....	19
Ensuring the Security of Private Information .....	20
JavaScript Objects, Methods, and Variables Reference .....	20
y Global Object .....	21
request Global Object .....	31
rest Object .....	31
response Global Object .....	38
result Object .....	38
Global Variables .....	38
JavaScript and E4X Best Practices for YQL .....	39
Paging Results .....	39
Including Useful JavaScript Libraries .....	40
Using E4X within YQL .....	40
JavaScript Logging and Debugging .....	42
Executing JavaScript Globally .....	43
Making Asynchronous Calls with JavaScript Execute .....	44
Examples of Open Data Tables with JavaScript .....	45
Hello World Table .....	45
Yahoo! Messenger Status .....	46
OAuth Signed Request to Netflix .....	47
Request for a Flickr frob .....	48
Celebrity Birthday Search using IMDB .....	49
Shared Yahoo! Applications .....	53
CSS Selector for HTML .....	55
Execution Rate Limits .....	56
3. Using Hosted Storage with YQL .....	58
Introduction .....	58
About YQL Hosted Storage .....	58
Storage Limits and Requirements .....	58
Storing New Records .....	59
Storing a New Record using Text .....	59
Storing a New Record using Data from an URL .....	60
Storing a New Named Record using Data from an URL .....	60
Using YQL to Read, Update, and Delete Records .....	60
Accessing Records using YQL .....	60
Deleting Records using YQL .....	61

Updating Records using YQL .....	61
Using Records within YQL .....	61
Using Hosted Environment Files .....	62
Using Hosted YQL Open Data Tables .....	62
Including Hosted JavaScript .....	62

# Introducing YQL

---

# Introducing YQL

## Abstract

This part of the YQL Guide introduces the Yahoo! Query Language (YQL). It discusses YQL uses, benefits, rate limits, and how content providers can limit access to YQL users.

For an overview of the features of the latest YQL release, refer to the [YQL Release Notes](http://developer.yahoo.com/yql/releasenotes/).<sup>3</sup>

Looking for more docs? See the [YOS Documentation](http://developer.yahoo.com/yos/)<sup>4</sup> landing page.

We welcome your feedback. Have a comment or question about this document? Let us know in the [YDN Forum for YOS Documentation](http://developer.yahoo.com/ydn/forum/)<sup>5</sup>.

---

<sup>3</sup> <http://developer.yahoo.com/yql/releasenotes/>

<sup>4</sup> [/yos](http://developer.yahoo.com/yos/)

<sup>5</sup> <http://developer.yahoo.net/forum/index.php?showforum=64>

---

---

# Table of Contents

1. Overview .....	1
What is YQL? .....	1
Why Use YQL? .....	1
Usage Information and Limits .....	2
Limiting Access to Content Provider Data .....	2
Blocking HTML Data Scraping from YQL .....	2
Blocking Non-HTML from YQL .....	3
Rate Limiting by IP Address .....	3
Internationalization Support .....	4
Character Encoding .....	4
2. YQL Tutorials and Code Examples .....	5
Introduction .....	5
Prerequisites .....	5
YQL Tutorials .....	5
First YQL Application .....	6
Creating YQL Open Data Tables .....	9
Executable JavaScript in Open Data Tables .....	12
YQL Code Examples .....	18
Prerequisites .....	19
Making YQL Queries with JavaScript .....	19
Making YQL Queries with PHP .....	21
YQL Social Application .....	22
YQL INSERT: WordPress Open Application .....	27
Getting Updates with YQL .....	30
Data Scraping with YQL .....	32
Sample Open Data Tables .....	34
YQL Screencasts .....	34
YQL Slideshows .....	35
Additional Tutorials and Code Examples .....	35
Additional References and Resources .....	36



---

## List of Figures

2.1. YQL News Application .....	7
2.2. YQL Sushi Restaurant Finder Application .....	8

---

# Chapter 1. Overview

**In this Chapter:**

- [“What is YQL?” \[1\]](#)
- [“Why Use YQL?” \[1\]](#)
- [“Usage Information and Limits” \[2\]](#)
- [“Limiting Access to Content Provider Data” \[2\]](#)
- [“Internationalization Support” \[4\]](#)

## What is YQL?

The YQL Web Service enables applications to query, filter, and combine data from different sources across the Internet. YQL statements have a SQL-like syntax, familiar to any developer with database experience. The following YQL statement, for example, retrieves a list of cat photos from Flickr:

```
SELECT * FROM flickr.photos.search WHERE text="cat"
```

To access the YQL Web Service, a Web application can call HTTP GET, passing the YQL statement as a URL parameter, for example:

```
http://query.yahooapis.com/v1/public/yql?q=SELECT * FROM flickr.photos.search WHERE text="Cat"
```

When it processes a query, the YQL Web Service accesses a datasource on the Internet, transforms the data, and returns the results in either XML or JSON format. YQL can access several types of datasources, including Yahoo! Web Services, other Web services, and Web content in formats such as HTML, XML, RSS, and Atom.

## Why Use YQL?

The YQL Web Service offers the following benefits:

- Because it resembles SQL, the syntax of YQL is already familiar to many developers. YQL hides the complexity of Web service APIs by presenting data as simple tables, rows, and columns.
- YQL includes pre-defined tables for popular Yahoo! Web services such as Flickr, Social, MyBlogLog, and Search.
- YQL can access services on the Internet that output data in the following formats: HTML, XML, JSON, RSS, Atom, and microformat.
- YQL is extensible, allowing you to define [Open Data Tables \[43\]](#) to access datasources other than Yahoo! Web Services. This feature enables you to mash up (combine) data from multiple Web services and APIs, exposing the data as a single YQL table.
- You can choose either XML or JSON for the [format \[36\]](#) of the results returned by requests to YQL.

- YQL [sub-selects \[11\]](#) enable you to join data from disparate datasources on the Web. YQL returns the data in a structured document, with elements that resemble rows in a table.
- With YQL, you can [filter \[8\]](#) the data returned with an expression that is similar to the WHERE clause of SQL.
- When processing data from large tables, you can [page \[12\]](#) through the query results.
- The [YQL Console<sup>1</sup>](#) enables you to run YQL statements interactively from your browser. The console includes runnable sample queries so that you can quickly learn YQL. For a quick introduction to the console, see the [Two-Minute Tutorial \[1\]](#).

## Usage Information and Limits

The following information describes the use, performance, dependencies, and limits of the YQL Web service. If you have additional questions, please read the [YQL Terms of Service<sup>2</sup>](#) or send an email to <yql-questions@yahoo-inc.com>.

### Usage Information:

- YQL can be used for commercial purposes.
- If we're going to shut down YQL, we will give you at least 6 months notice with an announcement on YDN and in our forum.
- YQL has a performance uptime target of over 99.5%.
- YQL relies on the correct operation of the Web services and content providers it accesses.

### Rate Limits:

- Per application limit (identified by your Access Key): 100,000 calls per day.
- Per IP limits: /v1/public/\*: 2,000 calls per hour; /v1/yql/\*: 20,000 calls per hour.
- All rates are subject to change.
- YQL rate limits are subject to the rate limits of other Yahoo! and 3rd-party Web services.

## Limiting Access to Content Provider Data

Content or API providers can opt out or restrict YQL access to their data by following the instructions in the sections below.

## Blocking HTML Data Scraping from YQL

YQL uses the `robots.txt` file on your server to determine the Web pages accessible from your site. YQL uses the user-agent "Yahoo Pipes 2.0" when accessing the `robots.txt` file and checks it for allows/disallows from this user agent. If the `robots.txt` check does prevent YQL from accessing your content, it will then fetch the target page using a different user agent:

---

<sup>1</sup> <http://developer.yahoo.com/yql/console/>

<sup>2</sup> <http://info.yahoo.com/legal/us/yahoo/yql/yql-4307.html>

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.14)
Gecko/20080404 Firefox/2.0.0.14
```

Therefore, to deny YQL access to your content, simply add "Yahoo Pipes 2.0" to the relevant parts of your `robots.txt`. For example:

```
User-agent: Yahoo Pipes 2.0
Disallow: /
```

Another approach is to block YQL on your Web server. For example, in Apache, add this to your virtual host block in `httpd.conf`:

```
SetEnvIfNoCase User-Agent "Yahoo Pipes" noYQL
<Limit GET POST>
Order Allow,Deny
Allow from all
Deny from env=noYQL
</Limit>
```

## Blocking Non-HTML from YQL

YQL fetches content from URLs when requested by a developer. Because YQL is not a Web crawler, it does not follow the robots exclusion protocol for non-HTML data, such as XML or CSV, from a site. To stop YQL from accessing any content on your site, block the YQL user-agent (Yahoo Pipes 2.0) on your Web server.

For example, on Apache servers, add this rule to your virtual host block in `httpd.conf`:

```
SetEnvIfNoCase User-Agent "Yahoo Pipes" noYQL
<Limit GET POST>
Order Allow,Deny
Allow from all
Deny from env=noYQL
</Limit>
```

## Rate Limiting by IP Address

YQL allows APIs to accurately use IP-based rate limits that will track and count on the YQL developer's IP address, rather than the IP addresses of shared proxy servers that YQL uses to access content on the Web.

For outgoing requests to external content and API providers, YQL determines the last valid client IP address connecting to its Web service and then ensures this is the first IP address in the `X-FORWARDED-FOR` HTTP header.

For example, in the `X-FORWARDED-FOR` HTTP header below, the request arriving at YQL came from the `1.2.3.4` IP address. IP-rate limiters should use this value rather than the IP addresses of YQL proxy servers.

```
X-FORWARDED-FOR: 1.2.3.4, 5.6.7.8, 9.10.11.12
```

We also set the `CLIENT-IP` HTTP header to this IP address.

For example:

CLIENT-IP: 1.2.3.4



### Note

Because these headers are "unsigned," they can be spoofed. Therefore, providers should only use these headers if the proxy setting them is trusted. The IP addresses of the proxy hosts that should be trusted can be found at <http://developer.yahoo.com/yql/proxy.txt>. This file will be updated as our proxy hosts change.

## Internationalization Support

### Character Encoding

YQL supports most of the character sets in the [IANA Character Sets Registry](http://www.iana.org/assignments/character-sets)<sup>3</sup>. YQL uses the HTTP header Content-Type in the request to determine the character encoding for the response body. If no character encoding is specified, YQL uses the default UTF-8. The YQL statement can also determine the character encoding for the body with the key `charset`. If the character encoding is specified in both places, the character set specified by `charset` has precedence.

For example, to request YQL use ISO/IEC 8859-1 to encode the response body, do one of the following:

- In your request, set the HTTP header Content-Type as shown below:

```
Content-Type: text/html; charset=iso-8859-1
```

- In the YQL statement, specify the character set with the key `charset` as shown below:

```
select * from html where url='http://example.com' and charset='iso-8559-1'
```



### Note

The YQL built-in function `sort` only correctly sorts results in English.

---

<sup>3</sup><http://www.iana.org/assignments/character-sets>

---

# Chapter 2. YQL Tutorials and Code Examples

## Introduction

This chapter provides the following:

- [“YQL Tutorials” \[5\]](#): Step-by-step instructions on how to use YQL.
- [“YQL Code Examples” \[18\]](#): PHP and JavaScript code with brief explanations.
- [“Sample Open Data Tables” \[34\]](#): Open Data Table examples that showcase the various capabilities of YQL.
- [“YQL Screencasts” \[34\]](#): Video screencasts that discuss YQL features and usage.
- [“YQL Slideshows” \[35\]](#): Slideshow presentations about YQL and how it can help developers.
- [“Additional Tutorials and Code Examples” \[35\]](#): More advanced tutorials and code examples showing specialized uses for YQL.
- [“Additional References and Resources” \[36\]](#): Additional references and resources for YQL, such as libraries, handouts, blog posts, and demos.

## Prerequisites

- [Two-Minute Tutorial \[1\]](#)
- Have a Web server that can be accessed over the Internet.
- Be familiar with the following technologies:
  - SQL
  - HTML/XHTML
  - JavaScript
  - Web feeds

## YQL Tutorials

- [First YQL Application \[6\]](#): Call YQL with JavaScript to get an RSS news feed.
- [Creating YQL Open Data Tables \[9\]](#): Create a YQL Open Data Table for the Bay Area Regional Transit (BART) ETA feed.
- [Executable JavaScript in Open Data Tables \[12\]](#): Run JavaScript within an Open Data Table to make requests to Web services and perform authorization.

# First YQL Application

Time Required: 10 min.

## Introduction

This tutorial shows how to create a simple Web application that uses YQL to fetch an RSS feed. The call to YQL is made within an HTML `script` tag, and the returned JSON response is parsed with JavaScript.

## Setting Up the Example

1. From your Web server, create a new file called `yql_news_app.html`.
2. Copy the HTML below into `yql_news_app.html`. The `src` attribute in the second script tag is given an empty value intentionally. We will assign a YQL statement to `src` later.

```
<html>
  <head><title>YQL and RSS: Yahoo! Top News Stories</title>
  <style type='text/css'>
    #results{ width: 40%; margin-left: 30%; border: 1px solid gray;
padding: 5px; height: 200px; overflow: auto; }
  </style>
  <script type='text/javascript'>
    // Parses returned response and extracts
    // the title, links, and text of each news story.
    function top_stories(o){
      var items = o.query.results.item;
      var output = '';
      var no_items=items.length;
      for(var i=0;i<no_items;i++){
        var title = items[i].title;
        var link = items[i].link;
        var desc = items[i].description;
        output += "<h3><a href='" + link + "'>"+title+"</a></h3>" +
desc + "<hr/>";
      }
      // Place news stories in div tag
      document.getElementById('results').innerHTML = output;
    }
  </script>
</head>
<body>
  <!-- Div tag for stories results -->
  <div id='results'></div>
  <!-- The YQL statment will be assigned to src. -->
  <script src=''></script>
</body>
</html>
```

3. Run the example query [get rss feed from yahoo top stories](http://developer.yahoo.com/yql/console/#h=select%20title%20from%20rss%20where%20url%3D%22http%3A%2F%2Fnews.yahoo.com%2F%2Frss%2F%2F%3F%3D%22&format=json)<sup>1</sup> in the YQL Console. Select the **JSON** radio button and click **TEST**.

You should see the returned JSON response on the **Formatted View** tab.

4. In the YQL statement, replace the word "title" with an asterisk "\*". Click **TEST**.

The asterisk will return all of the fields (rows) of the response.

5. From the returned JSON response, find the `results` object. Notice that within the results object, there is the array `item` containing objects with the title, link, and description of each news article.
6. Click **Copy URL**. From `yql_news_app.html`, paste the URL into the `src` attribute of the second `script` tag as seen below:

```
<body>
  <div id='results'></div>
  <script
src=http://developer.yahoo.com/yql/console/#h=select%20*%20from%20local.search%20where%20query%3D%22sushi%22%20and%20location%3D%22san%20francisco%2C%20ca%22
  </script>
</body>
```

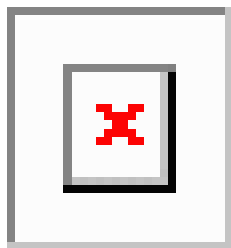
Notice that the YQL statement has been URL-encoded and the `format` parameter specifies that the response be in JSON.

7. At the end of the URL, replace the callback value 'cbfunc' with 'top\_stories'.

The new callback function `top_stories` will be called as soon as YQL returns the response and have access to the returned JSON.

8. Point your browser to `yql_news_app.html`. Your YQL News Application should resemble the figure below.

**Figure 2.1. YQL News Application**



9. (Optional) Use almost the exact same code to create an application for finding sushi restaurants in San Francisco.

Replace the YQL statement for the Yahoo! Top Stories in your YQL News Application with the example query [find sushi restaurants in san francisco](http://developer.yahoo.com/yql/console/#h=select%20*%20from%20local.search%20where%20query%3D%22sushi%22%20and%20location%3D%22san%20francisco%2C%20ca%22)<sup>2</sup>. Be sure that the format is JSON and that the name of the callback function is correct.

---

<sup>1</sup><http://developer.yahoo.com/yql/console/#h=select%20title%20from%20rss%20where%20url%3D%22http%3A%2F%2Fnews.yahoo.com%2F%2Frss%2F%2F%3F%3D%22&format=json>

<sup>2</sup>[http://developer.yahoo.com/yql/console/#h=select%20\\*%20from%20local.search%20where%20query%3D%22sushi%22%20and%20location%3D%22san%20francisco%2C%20ca%22](http://developer.yahoo.com/yql/console/#h=select%20*%20from%20local.search%20where%20query%3D%22sushi%22%20and%20location%3D%22san%20francisco%2C%20ca%22)

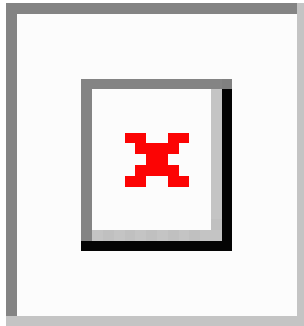


10. Make a few changes below to the function `top_stories` as seen below:

```
function top_stories(o){
  var items = o.query.results.Result;
  var output = '';
  var no_items=items.length;
  for(var i=0;i<no_items;i++){
    var title = items[i].Title;
    var link = items[i].Url;
    var desc = items[i].Rating.LastReviewIntro;
    output += "<h3><a href='" + link + "'>" + title + "</a></h3>" + desc
    + "<hr/>";
  }
  document.getElementById('results').innerHTML = output; }
```

11. The YQL News Application is now the YQL Sushi Restaurant Finder Application.

**Figure 2.2. YQL Sushi Restaurant Finder Application**



## Source Code

### YQL Top News Application

```
<xi:include></xi:include>

<html>
  <head><title>YQL and RSS: Yahoo! Top News Stories</title>      <style
  type='text/css'>
    #results{ width: 40%; margin-left: 30%; border: 1px solid gray;
padding: 5px; height: 200px; overflow: auto; }
  </style>
  <script type='text/javascript'>
    function top_stories(o){
      var items = o.query.results.item;
      var output = '';
      var no_items=items.length;
      for(var i=0;i<no_items;i++){
        var title = items[i].title;
        var link = items[i].link;
        var desc = items[i].description;
        output += "<h3><a href='" + link + "'>" + title + "</a></h3>" +
```

```

desc + "<hr/>";
    }
    document.getElementById('results').innerHTML = output;
    }
</script>
</head>
<body>
    <div id='results'></div>
    <script
src="http://your_domain_name/yql/yql.js" type="text/javascript"></script>
    </script>
</body>
</html>

```

## Creating YQL Open Data Tables

Time Required: 15 min.

### Introduction

This tutorial shows you how create a YQL Open Data Table for the Bay Area Region Transit (BART) Real Time ETA feed. You will then use the YQL console to run a YQL query using the Open Data Table that you created.

### What Are Open Data Tables?

YQL already contains an extensive list of built-in tables for you to use that cover a wide range of Yahoo! Web services and access to off-network data. Developers can now add to this list of built-in tables by creating their own Open Data Tables. YQL allows you to create and use your own table definitions, enabling YQL to bind to any data source through the SQL-like syntax and fetch data. Once an Open Data Table is created, anyone can use these definitions in YQL.

By creating an Open Data Table for the feed, you can then use YQL to make queries on the data from the feed and mash up other data from existing YQL tables with the social data. For more information about Open Data Tables, see [Using YQL Open Data Tables](#)<sup>3</sup>.

### Creating and Using Your Open Data Table

1. Copy [bart.xml](#)<sup>4</sup> to your Web server. The file `bart.xml` defines the Open Data Table, which allows the YQL Web service to access data from the BART ETA feed. For a more detailed explanation, see [A Closer Look at the Code \[10\]](#).
2. Go to the [YQL console](#)<sup>5</sup>.
3. From the YQL console, type the YQL statement below in the "Your YQL Statement" text area and click the button TEST. Be sure to replace "your\_domain\_name" with the domain name of your Web site.

```
USE "http://your_domain_name/bart.xml" AS bart_table; SELECT * FROM
bart_table;
```

<sup>3</sup> [../yql/guide/yql-opentables-chapter.html](#)

<sup>4</sup> [examples/bart.xml.txt](#)

<sup>5</sup> [http://developer.yahoo.com/yql/console/](#)

4. From the "Formatted View" tab, you should see results similar to those below:

```
<results>
  <station>
    <name>12th St. Oakland City Center</name>
    <abbr>12TH</abbr>
    <date>06/01/2009</date>
    <time>11:46:00 AM PDT</time>
    <eta>
      <destination>Fremont</destination>
      <estimate>14 min, 29 min</estimate>
    </eta>
    <eta>
      <destination>Millbrae</destination>
      <estimate>5 min, 20 min, 35 min</estimate>
    </eta>
    <eta>
      <destination>Pittsburg/Bay Point</destination>
      <estimate>7 min, 22 min, 37 min</estimate>
    </eta>
    <eta>
      <destination>Richmond</destination>
      <estimate>2 min, 7 min, 17 min</estimate>
    </eta>
    <eta>
      <destination>SF Airport</destination>
      <estimate>12 min, 27 min, 42 min</estimate>
    </eta>
  </station>
  ...
</results>
```

5. Try a few more YQL statements using the Open Data Tables. Below are a few sample YQL statements.

- This statement will get you the station name and ETAs for any train going to the San Francisco International Airport.

```
USE "http://your_domain_name/bart.xml" AS bart_table; SELECT name,
eta FROM bart_table WHERE eta.destination LIKE "%SF%"
```

- This statement gets the station name, the destination of the train, and the ETA to different destinations from the 24th and Mission Street station.

```
USE "http://your_domain_name/bart.xml" AS bart_table; SELECT name,
eta.destination, eta.estimate FROM bart_table WHERE name LIKE "%24%"
```

## A Closer Look at the Code

This section will examine in greater detail at the Open Data Table and the YQL queries that are used in this tutorial.

## Open Data Table

The Open Data Table must conform to the [XML schema table.xsd](http://query.yahooapis.com/v1/schema/table.xsd)<sup>6</sup>, which is given as the XML namespace attribute for the element `table`. The element `meta` has child elements that provide general information about the Open Data Table such as the author, the location of the documentation, and a sample query, as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <author>Mike Oppentables</author>
    <documentationURL>http://yourbartfeed.com/docs</documentationURL>
    <description>Uses the BART ETA feed.</description>
    <sampleQuery>SELECT * FROM {table} WHERE abbr="12th"</sampleQuery>
  </meta>
```

The element `bindings` allows YQL to map the data source so that information can be queried. The element `select` defines what repeating element of the XML from the data source will act as a row of a table with the attribute `itemPath`, which is `<root><station>...</station>` (given as `root.element`) in the code below. A good example of a repeating element in XML would be the element `entry` in an [Atom 1.0 feed](http://tools.ietf.org/html/rfc4287#section-1.1)<sup>7</sup>.

```
<bindings>
  <select itemPath="root.station" produces="XML">
    <urls>
      <url>http://www.bart.gov/dev/eta/bart_eta.xml</url>
    </urls>
  </select>
</bindings>
```

## YQL Query

You invoke an Open Data Table with the verbs `USE` and `AS`. YQL fetches the Open Data Table definition pointed to by the verb `USE` and then the verb `AS` creates an alias to that definition.

```
USE "http://your_domain_name/bart.xml" AS bart_table;
```

The alias `bart_table` can now be accessed with a YQL statement. The simple YQL query below returns all the rows from the BART feed.

```
SELECT * FROM bart_table;
```

## Source Code

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <author>You</author>
    <documentationURL>None</documentationURL>
    <description>Uses the BART ETA feed.</description>
    <sampleQuery>SELECT * FROM {table} WHERE abbr="12th"</sampleQuery>
  </meta>
```

---

<sup>6</sup> <http://query.yahooapis.com/v1/schema/table.xsd>

<sup>7</sup> <http://tools.ietf.org/html/rfc4287#section-1.1>

```
<bindings>
  <select itemPath="root.station" produces="XML">
    <urls>
      <url>http://www.bart.gov/dev/eta/bart_eta.xml</url>
    </urls>
  </select>
</bindings>
</table>
```

## Executable JavaScript in Open Data Tables

Time Required: 15-20 min.

### Introduction

This tutorial shows how to include JavaScript in an Open Data Table definition that is executed by the YQL Web service. The JavaScript in the Open Data Table for this tutorial includes external libraries and uses OAuth to make an authorized call to the Netflix API. With the Netflix Open Data Table, you can then run YQL statements from the YQL console or create an Open Application that searches and displays movies from the Netflix catalog.

Including JavaScript in an Open Data Table definition allows you to do the following:

- Access external libraries
- Make requests to Web services
- Use E4X to structure the response in XML
- Dynamically control the response with conditional logic
- Perform Web authorization

### Prerequisites

- Creating YQL Open Data Tables
- [Register to be a Netflix developer](http://developer.netflix.com/member/register)<sup>8</sup> and [get Netflix API keys](http://developer.netflix.com/member/register)<sup>9</sup>
- JavaScript
- XML

### Executable JavaScript

JavaScript is placed in the element `execute` of an Open Data Table definition. When a `SELECT` statement calls an Open Data Table definition having the element `execute`, YQL executes the JavaScript on the backend and returns the response generated by the JavaScript. Open Data Table definitions having the element `execute` must return a response formed by the JavaScript.

---

<sup>8</sup> <http://developer.netflix.com/member/register>

<sup>9</sup> <http://developer.netflix.com/member/register>

In contrast, Open Data Tables not having the element `execute` make GET requests to the URI defined in the element `url` and return the response from that URI resource. We'll now look at a couple of examples of Open Data Tables with and without the element `execute` to illustrate the difference.

### Example of Open Data Table Definition without JavaScript

The Open Data Table definition below does not include the element `execute`, so YQL makes a GET request to the URI defined in the element `url`: [http://www.bart.gov/dev/eta/bart\\_eta.xml](http://www.bart.gov/dev/eta/bart_eta.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <author>You</author>
    <documentationURL>None</documentationURL>
    <description>Uses the BART ETA feed.</description>
    <sampleQuery>SELECT * FROM {table} WHERE abbr="12th"</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="root.station" produces="XML">
      <urls>
        <url>http://www.bart.gov/dev/eta/bart_eta.xml</url>
      </urls>
    </select>
  </bindings>
</table>
```

### Example of Open Data Table Definition with JavaScript

Because of the presence of the element `execute`, YQL executes the JavaScript and returns the response assigned to `response.object`, which will be discussed in [Creating the Response \[16\]](#).

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where a='cat' and
b='dog';</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="" produces="XML">
      <urls>
        <url>http://fake.url/{a}</url>
      </urls>
      <inputs>
        <key id='a' type='xs:string' paramType='path' required="true"
/>
        <key id='b' type='xs:string' paramType='variable' required="true"
/>
      </inputs>
      <execute><![CDATA[
        // Your javascript goes here.
        // We will run it on our servers
        response.object = <item>
          <url>{request.url}</url>
          <a>{a}</a>
```

```
                <b>{b}</b>
            </item>;

    ]]></execute>
</select>
</bindings>
</table>
```

## A Closer Look at the Code

JavaScript in Open Data Tables definitions have access to the three global objects `y`, `request`, and `response`. From these global objects, methods can be called to include external libraries, perform 2-legged authorization, make requests to Web services, run YQL queries, convert data structures, such as XML to JSON, and more. We'll be looking at the methods used in this tutorial that allow you to include external libraries, make requests to Web services, and extract the results from a returned response.

To read more about the JavaScript global objects and the available methods, see the [JavaScript Objects and Methods Reference](#)<sup>10</sup>.

## Including External Libraries

The global object `y` includes the method `include` that allows you to reference external JavaScript libraries from an Open Data Table definition. Because calls to the Netflix API must be authorized with OAuth, you include external libraries to handle the authorization and sign the request as shown in the code below:

```
<execute>
y.include("http://oauth.googlecode.com/svn/code/javascript/oauth.js");
y.include("http://oauth.googlecode.com/svn/code/javascript/sha1.js");
...
</execute>
```

## Making Requests

To make a request to a Web service, you use the method `get` from either the global object `request` or from an instance of `request`; the instance is created by calling the method `rest` from the global object `y`. Before looking at the code used in the Open Data Table definition for this tutorial, let's look at a couple of simple examples of using both the global object and an instance of `request`.

This example creates an instance of [request](#)<sup>11</sup> by passing the URI resource to the method `rest`. From the instance, the method `get` can make the request to the Web service.

```
// Create an instance of the request object
var request_instance = y.rest("http://some_web_service");

// Make request to web service with HTTP method 'GET'
var returned_response = request_instance.get();
```

You can also call the method `get` directly from the global object `request`. The element `url` holds the URI to the Web service, which is stored in `request.url`.

```
...
<select itemPath="" produces="XML">
  <urls>
```

---

<sup>10</sup> [../yql/guide/yql-javascript-objects.html](#)

<sup>11</sup> [../yql/guide/yql-javascript-objects.html#yql-execute-requestglobalobject](#)

```
<url>http://some_web_service_or_feed</url>
</urls>
<execute><![CDATA[
  // request.url == 'http://some_web_service_or_feed
  // This is the same as 'y.rest(request.url).get();'
  var returned_response = request.get();
  ...
]]></execute>
</select>
...
```

You can also pass in parameters and define the content returned when making a request to a Web service, which is needed to make authorized requests, such as the request to the Netflix API in this tutorial.

To get the response from the Netflix API, the content type must be defined as 'application/xml', and the OAuth authorization header must be passed. The code snippet below also uses the objects `response` and `response.object` to extract results from the returned response and create the response that YQL returns. We'll examine the object response in [Getting and Creating a Response \[15\]](#).

```
// The content type is defined and the OAuth header
// is passed to get a response from the Netflix API
response.object =
request.contentType('application/xml').header("Authorization",
OAuth.getAuthorizationHeader("netflix.com",
message.parameters)).get().response;
```

## Getting and Creating the Response

### Extracting Data from the Returned Response

To get the data (results) from the returned response, you reference the object [response<sup>12</sup>](#). From this earlier code example, you can see the dot operator being used to reference the results of the returned response.

```
// Create an instance of the request object
var request_obj = y.rest("http://some_web_service");

// Make request to web service with HTTP method 'GET'
// and get the results by referencing 'response'
var response_results = request_obj.get().response;
```

We also saw this example of using the response object to get the data from the Netflix Web service.

```
// The object response lets you extract the
// results from the returned response
response.object =
request.contentType('application/xml').header("Authorization",
```

---

<sup>12</sup> [../yql/guide/yql-javascript-objects.html#yql-execute-responseglobalobject](http://yql/guide/yql-javascript-objects.html#yql-execute-responseglobalobject)



```
OAuth.getAuthorizationHeader("netflix.com",  
message.parameters)).get().response;
```

## Creating the Response

When including JavaScript in an Open Data Table, the response from the YQL Web service is determined by the data that is assigned to the object `response.object`. You can use the returned value from a Web service or create your own with [JSON](#)<sup>13</sup> or [E4X](#)<sup>14</sup>.

In this code snippet, an XML literal is created with E4X and then assigned to `response.object`. To interpolate the variable into the XML literal, enclose the variable in braces.

```
// Use E4X to create XML literals with  
// interpolated variable and assign to  
// response.object  
var error = "Failed to get response";  
response.object = <error><message>{error}</message></error>;
```

The code from the tutorial assigns the returned XML response to `response.object` as seen below. If there is an error, a JSON object holding error message is returned.

```
try {  
    // get the content from service along with the OAuth header, and  
    return the result back out  
    response.object =  
request.contentType('application/xml').header("Authorization",  
OAuth.getAuthorizationHeader("netflix.com",  
message.parameters)).get().response;  
} catch(err) {  
    response.object = {'result':'failure', 'error': err};  
}
```

## Paging Results

The element `paging` lets you have more control over the results returned from the YQL query. We'll look in detail at the paging used in the Netflix Open Data Table that is shown below. For more information about paging, see the [Open Data Tables Reference](#)<sup>15</sup> and [Paging Results](#)<sup>16</sup>.

```
<paging model="offset">  
  <start id="start_index" default="0" />  
  <pagesize id="max_results" max="100" />  
  <total default="20" />  
</paging>
```

The value 'offset' for the attribute `model` states that the YQL query can state an offset from the start of the results. The ability to get an offset from a result set depends on the source of the data, which in the tutorial is the Netflix API. Be sure to verify that your data source allows retrieving data from an offset when you create future Open Data Tables.

```
<paging model="offset">
```

---

<sup>13</sup> <http://json.org>

<sup>14</sup> [http://en.wikipedia.org/wiki/ECMAScript\\_for\\_XML](http://en.wikipedia.org/wiki/ECMAScript_for_XML)

<sup>15</sup> <http://developer.yahoo.com/yql/guide/yql-opentables-reference.html#yql-opentables-paging>

<sup>16</sup> <http://developer.yahoo.com/yql/guide/yql-execute-bestpractices.html#yql-execute-paging>

The default offset is set by the attribute `default` in the element `start`, which is 0 or no offset in this example.

```
<start id="start_index" default="0" />
```

The maximum number of results that can be returned by a YQL query on this table is 100, which is defined by the attribute `max`.

```
<pagesize id="max_results" max="100" />
```

The attribute `default` in the element `total` states the default number of results returned for each page.

```
<total default="20" />
```

## Setting Up This Example

To set up the Netflix Open Data Table and run YQL queries:

1. Copy [netflix\\_table.xml](#)<sup>17</sup> to your Web server. The file `netflix_table.xml` defines the Open Data Table, which allows the YQL Web service to access data from the NetFlix API. For a more detailed explanation, see [A Closer Look at the Code \[10\]](#).
2. If you do not have Netflix API keys, [register for a Netflix developer account](#)<sup>18</sup> and [apply for an API key](#)<sup>19</sup>.
3. Go to the [YQL Console](#)<sup>20</sup>. Make sure that the URL of the YQL Console begins with `https`.
4. Select the first 20 results (see [Paging Results \[16\]](#)) for movies that match the key word "Rocky". Be sure to use your Consumer Key and Consumer Secret from [your Netflix account](#)<sup>21</sup> for the values for `ck` and `cs`.

```
USE "http://your_domain/netflix.xml" AS netflix_table; SELECT * FROM
netflix_table WHERE term="rocky" AND ck="your_consumer_key" AND
cs="your_consumer_secret"
```

5. In this YQL statement, you ask for the first 10 results for movies, and the results will include the film title, the URL of the box art image, and the average rating.

```
USE "http://yourdomain/netflix.xml" AS netflix_table; SELECT cata-
log_title.title, catalog_title.box_art.medium, catalog_title.aver-
age_rating FROM netflix_table(10) WHERE term="rocky" AND
ck="your_consumer_key" AND cs="your_consumer_secret"
```

6. You can also specify an offset for the returned results. The YQL statements below returns the five titles and ratings for Star Trek movies with average ratings over 3.0. The results are sorted by the rating in descending order, and the offset is five.

```
USE "http://yourdomain/netflix.xml" AS netflix_table; SELECT cata-
log_title.title.regular, catalog_title.average_rating FROM net-
flix_table(5,5) WHERE term="star trek" AND ck="your_consumer_key" AND
```

---

<sup>17</sup> [examples/netflix\\_table.xml.txt](#)

<sup>18</sup> <http://developer.netflix.com/member/register>

<sup>19</sup> <http://developer.netflix.com/apps/register>

<sup>20</sup> <https://developer.yahoo.com/yql/console/>

<sup>21</sup> <http://developer.netflix.com/apps/mykeys>

```
cs="your_consumer_secret" AND catalog_title.average_rating > "3.0" |
sort(field="catalog_title.average_rating") | reverse()
```

7. **Optional:** Create a Open Application that uses the Netflix Open Data Table

Copy [yql\\_netflix\\_app.html](#)<sup>22</sup> to your Web server.

8. Edit yql\_netflix\_appl.html and insert [your Netflix keys](#)<sup>23</sup> and the location of your Netflix Open Data Table in the code as shown below:

```
// Place your Netflix keys here to authorize your app
NETFLIX.keys = { ck: "use_your_consumer_key",
cs:"use_your_consumer_secret" }

// Make sure this URI points to your Netflix Open Data Table
NETFLIX.odt = "http://your_domain/netflix.xml";
```

9. From [My Projects](#)<sup>24</sup>, create a new Open Application.
10. In the Application Editor, specify the URL to netflix\_app.html in the "Application URL" field.
11. Click "Preview" to see the canvas view. Your application will have a text field for entering a movie title. Enter "Rocky" and click the button "Find Movie" to see results similar to those in the figure below.



- 12 Try modifying the YQL statement in the code to more precisely return the data that you want from the Netflix API. Take a look at the [XML response from a catalog/titles request](#)<sup>25</sup> to the Netflix API to see the available data and how it is structured.

## Source Code

### Netflix Open Data Table

```
<xi:include></xi:include>
```

### Netflix Application

```
<xi:include></xi:include>
```

## YQL Code Examples

- [Making YQL Queries with JavaScript \[19\]](#): Get data from the GeoPlanet API using the OpenSocial library.
- [Making YQL Queries with PHP \[21\]](#): Get data from the Upcoming API using PHP with cURL.
- [YQL Social Application \[22\]](#): Use YQL with the Yahoo! Social SDK for PHP to make authorized calls to the Flickr API and the Yahoo! Social APIs.

---

<sup>22</sup> [examples/yql\\_netflix\\_app.html.txt](#)

<sup>23</sup> <http://developer.netflix.com/apps/mykeys>

<sup>24</sup> <http://developer.yahoo.com/dashboard>

<sup>25</sup> [http://developer.netflix.com/docs/REST\\_API\\_Reference#0\\_52696](http://developer.netflix.com/docs/REST_API_Reference#0_52696)

- [YQL INSERT: WordPress Open Application \[27\]](#): Post to a WordPress blog from an Open Application using YQL.
- [Getting Updates with YQL \[30\]](#): Get Yahoo! Updates from connections with YQL.
- [Data Scraping with YQL \[32\]](#) Use YQL with XPath to scrape data from Yahoo! Finance.

## Prerequisites

- [YQL First Application \[6\]](#)
- [Getting Started: Build Your First Open Application<sup>26</sup>](#)

## Making YQL Queries with JavaScript

### Summary

This code example shows you how to make YQL queries with JavaScript and the OpenSocial function `makeRequest`. The YQL query in the code example will get data from the GeoPlanet API based on the user's input.

### OpenSocial Method: `makeRequest`

By using the OpenSocial method [gadgets.io.makeRequest<sup>27</sup>](#), you can make calls to a Web service with JavaScript without using a `crossdomain.xml` file. The calls in this example use 2-legged OAuth authorization.

Before calling the function `makeRequest`, define the base URI of the YQL Web service, the YQL query, and a callback function to handle the response, as shown below:

```
var BASE_URI = 'http://query.yahooapis.com/v1/yql';

// Base URI for Web service
var yql_base_uri = "http://query.yahooapis.com/v1/yql";

// Create a variable to make results available
// in the global namespace
var yql_results = "";

// Create a YQL query to get geo data for the
// San Francisco International Airport
var yql_query = "SELECT * from geo.places WHERE text='SFO'";

// Callback function for handling response data
function handler(rsp) {
    if(rsp.data){
        yql_results = rsp.data;
    }
}
```

---

<sup>26</sup> [../yap/guide/creating\\_open\\_app.html](#)

<sup>27</sup> <http://code.google.com/apis/opensocial/docs/0.7/reference/gadgets.io.html#makeRequest>

```
}  
}
```

To make a call to the YQL Web service, you create a URL-encoded query string with the YQL query and the requested format type of the response and append this query string to the base URI of the YQL Web service. The code snippet below shows the base URI with the appended query string for the YQL Web service. Note that the spaces have been URL-encoded.

```
http://query.yahooapis.com/v1/public/yql?q=SELECT%20*%20FROM%20geo.places%20WHERE%20text%3D%22SF%22&format=json
```

This code example has the JavaScript function `toQueryString` that creates the query string for the call to the YQL Web service as seen below:

```
// This utility function creates the query string  
// to be appended to the base URI of the YQL Web  
// service.  
function toQueryString(obj) {  
    var parts = [];  
    for(var each in obj) if (obj.hasOwnProperty(each)) {  
        parts.push(encodeURIComponent(each) + '=' +  
encodeURIComponent(obj[each]));  
    }  
    return parts.join('&');  
};
```

By using a closure in the code below, the variable `query`, which holds the YQL query, is available to `makeRequest`. The function `toQueryString` then builds the query string, which is appended to the base URI by `makeRequest` before making the call to the YQL Web service.

```
// Store the anonymous function that wraps  
// the OpenSocial function makeRequest  
var runQuery = function(ws_base_uri, query, handler) {  
    gadgets.io.makeRequest(ws_base_uri, handler, {  
        METHOD: 'POST',  
        POST_DATA: toQueryString({q: query, format: 'json'}),  
        CONTENT_TYPE: 'JSON',  
        AUTHORIZATION: 'OAuth'  
    });  
};
```

In the code snippet, the function object `runQuery` is passed the parameters for `makeRequest` to call the YQL Web service. The handler returns the response data (JSON object) to the OpenSocial function `stringify` to be converted to a string so it can be displayed in a `div` tag.

```
<div id="results"></div>  
<script>  
// Call YQL Web service and use YQL query  
// to get results from the GEO  
runQuery(yql_base_uri, yql_query, handler);  
  
// Use stringify function from OpenSocial library  
// to convert JSON to string and display the string  
// in the div with the id 'results'  
document.getElementById('results').innerHTML =
```

```
gadgets.json.stringify(yql_results.data);  
</script>
```

## Source Code

```
<xi:include></xi:include>
```

# Making YQL Queries with PHP

## Summary

This example, [yql\\_php.php](#)<sup>28</sup>, is a simple application that uses cURL to make YQL calls to the Upcoming API.

To understand the following material, you should already be familiar with the topics covered in the [Two-Minute Tutorial \[1\]](#).

## Building the YQL URL

To build the YQL URL, append the YQL query to the base URL of the YQL Web service.

1. First, assign the base URL of the YQL Web service to a variable.

```
$yql_base_url = "http://query.yahooapis.com/v1/public/yql";
```

2. Now create the YQL query to the Upcoming API and append it to the YQL base URL. YQL queries are passed as query strings, so they must be URL encoded. The query string must begin with `q=`.

```
$yql_query = "select * from upcoming.events where location='San  
Francisco' and search_text='dance'";  
  
$yql_query_url = $yql_base_url . "?q=" . urlencode($yql_query);
```

3. The YQL Web service returns XML as the default format. This code example requests JSON by appending the name-value pair `format=json` to the query string as shown.

```
$yql_query_url .= "&format=json";
```

4. [Run the query](#)<sup>29</sup> in the YQL console and look at both the URL in **The REST query** window and the JSON response in the **Formatted View** window.

## Calling the YQL Web Service with cURL

Calling the YQL Web service with cURL only requires three lines of code. After initializing the call by passing the curl the YQL URL, you request a response with `curl_setopt` and then execute the call.

---

<sup>28</sup> [./examples/yql\\_php.php](#)

<sup>29</sup> [http://query.yahooapis.com/v1/public/yql?q=select%20\\*%20from%20upcoming.events%20where%20location%3D%22San%20Francisco%22%20and%20search\\_text%3D%22dance%22&format=json](http://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20upcoming.events%20where%20location%3D%22San%20Francisco%22%20and%20search_text%3D%22dance%22&format=json)

```
$session = curl_init($yql_query_url);
curl_setopt($session, CURLOPT_RETURNTRANSFER, true);
$json = curl_exec($session);
```

To make the response easier to handle, convert the JSON response to a PHP object to easily access data.

```
$phpObj = json_decode($json);
```

## Parsing the Response

The YQL Web service will always return results wrapped in the `query` field with meta data. If the YQL query returns data, the data is wrapped in the `results` field, which you can see by [running this query](#)<sup>30</sup>.

If no data is returned, the `results` element in the XML response is empty or the `results` field in JSON is null. Therefore, before parsing the response returned by YQL, your code should always check the `results` field as shown below:

```
if(!is_null($phpObj->query->results)){
    // Safe to parse data
}
```

The structure of the data within the `results` field that is returned by YQL is different for each API. The repeated field (like a row in an SQL table) in the returned response in this code example is the event field. The code below extracts and displays data from each event:

```
if(!is_null($phpObj->query->results)){
    foreach($phpObj->query->results->event as $event){
        $events .= "<div><h2>" . $event->name . "</h2><p>";
        $events .= html_entity_decode(wordwrap($event->description, 80,
"<br/>"));
        $events
        .="</p><br/>$event->venue_name<br/>$event->venue_address<br/>";
        $events .="$event->venue_city, $event->venue_state_name";
        $events .="<p><a href=$event->ticket_url>Buy Tickets</a></p></div>";
    }
}
```

## Source Code

```
<xi:include></xi:include>
```

## YQL Social Application

### Summary

This example, [yql\\_basic.php](#)<sup>31</sup>, is a simple application that uses the Yahoo! Social SDK for PHP to make YQL calls. Using YQL with the SDK is ideal because the SDK will handle your OAuth authorization, which is required to access the Social Directory APIs. YQL, in turn, extends the functionality of the SDK

<sup>30</sup> [http://query.yahooapis.com/v1/public/yql?q=select%20\\*%20from%20upcoming.events%20where%20location%3D%22San%20Francisco%22%20and%20search\\_text%3D%22dance%22&format=json](http://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20upcoming.events%20where%20location%3D%22San%20Francisco%22%20and%20search_text%3D%22dance%22&format=json)

<sup>31</sup> [./examples/yql\\_basic.phps](#)

by enabling access to the public data from Yahoo! Web services and external data such as RSS and Atom feeds.

To understand the following material, you should already be familiar with the topics covered in [My Social PHP Application](#)<sup>32</sup> and the [Two-Minute Tutorial \[1\]](#).

## Initial Code

Before proceeding with the code that makes YQL queries, be sure to include the PHP SDK and define an API Key and a Shared Secret:

```
// Include the PHP SDK.
include_once("yosdk/lib/Yahoo.inc");

// Define constants to store your API Key (Consumer Key) and
// Shared Secret (Consumer Secret).
define("API_KEY", "your_Consumer_Key_goes_here");
define("SHARED_SECRET", "your_Shared_Secret_goes_here");
```

## Querying Public Data

There are great number of sources for public data on the Web that can be accessed with YQL. Although public, many of these sources require authorization. With the API Key and the Shared Secret provided by Yahoo!, you can use the PHP SDK to perform two-legged authorization with the `YahooApplication` class. (If you need to read up on two-legged authorization, see [Private Data v. Public Data](#)<sup>33</sup>.) From a `YahooApplication` object, invoke the method `query`, which calls the YQL Web service (that is, runs the YQL queries). The following code snippets show you how to perform two-legged authorization with `YahooApplication` and how to make YQL queries.

The API Key and Shared Secret are passed to the constructor of the `YahooApplication` class. If the application has been authorized by Yahoo!, the `YahooApplication` object `$two_legged_app` is returned:

```
$two_legged_app = new YahooApplication(API_KEY, SHARED_SECRET);
```

Next, the application defines two YQL queries: one to the Flickr API, another to the Yahoo! News RSS feed. You can also run these queries in the [YQL Console](#)<sup>34</sup>:

```
$flickr_query =
    "select * from flickr.photos.search where text=\"panda\" limit 3";
$news_feed =
    "select * from rss where url='http://rss.news.yahoo.com/rss/topstories'
    and title LIKE \"%China%\"";
```

With the `$two_legged_app` object, the application calls the `query` method. The `var_dump` shows the structure and contents of the query response of the YQL Web service:

---

<sup>32</sup> [../yap/guide/other-code-exs.html#my\\_social](#)

<sup>33</sup> [../oauth/guide/about.html#oauth-private\\_public\\_data](#)

<sup>34</sup> <http://developer.yahoo.com/yql/console/>



```
$flickrResponse = $two_legged_app->query($flickr_query);  
var_dump($flickrResponse);  
  
$newsResponse = $two_legged_app->query($news_feed);  
var_dump($newsResponse);
```

## Querying Private Data

Unlike YQL queries for public data, you need user authorization to obtain data from the Social Directory APIs (Profiles, Updates, Connection, Contacts, and Presence). During the process of user authorization, the user will be redirected to the Yahoo! login page and then asked to authorize your application. You initiate this three-legged authorization process with the `YahooSession` class. (If you are unfamiliar with the term "three-legged authorization," you may want to read [Private Data v. Public Data](#)<sup>35</sup>.) Getting a valid `YahooSession` object means that the user has authorized your application, allowing you to access this user's private data. The following code snippets show how to perform authorization with `YahooSession` and how to use YQL to access private social data from Yahoo!

The method `requireSession` makes sure the user has logged into Yahoo!, redirecting to the Yahoo! login page if necessary. A `YahooSession` object is returned, confirming that the OAuth authorization has been completed:

```
$session=YahooSession::requireSession(API_KEY, SHARED_SECRET);
```

The YQL queries and API names are assigned to the associative array `$api_queries`:

```
// Define YQL queries for the Social Directory APIs  
$profile = "select * from social.profile where guid=me";  
$contacts = "select fields.value from social.contacts where guid=me";  
$connections = "select * from social.connections where owner_guid=me";  
$updates = "select * from social.updates where guid=me";  
$status = "select value.status from social.presence where guid=me";  
  
$api_queries = array("Profiles"=>$profile,  
    "Contacts"=>$contacts,  
    "Connections"=>$connections,  
    "Updates"=>$updates,  
    "Presence"=>$status);
```

The YQL query is made from the `YahooSession` object `$session` with the method `query`. The use of `var_dump` allows you to see the contents of the returned response for each YQL query:

```
// Make the calls to YQL and dump the responses.  
foreach($api_queries as $api=>$query) {  
    echo "<h2>$api Data</h2>";  
    $queryResponse = $session->query($query);  
    if ($queryResponse == NULL) {  
        echo "<p>";  
        echo "Error: No query response for $api.";
```

---

<sup>35</sup> [../oauth/guide/about.html#oauth-private\\_public\\_data](http://oauth.guide/about.html#oauth-private_public_data)

```
    echo " Check your permissions. Also, check the syntax of the YQL
query.";
    echo "</p>";
}
else {
    echo "<pre>";
    var_dump($queryResponse);
    echo "</pre>";
}
```

The following listing shows the `var_dump` from the YQL query made on the Profiles API.

```
Profiles Data

array(1) {
  ["query"]=>
    array(7) {
      ["count"]=>
        string(1) "1"
      ["created"]=>

        string(20) "2008-10-08T06:11:21Z"
      ["lang"]=>
        string(5) "en-US"
      ["updated"]=>
        string(20) "2008-10-08T06:11:21Z"
      ["uri"]=>
        string(80)
"http://query.yahooapis.com/v1/yql?q=select++from+social.profile+where+guid%3Dme"

      ["diagnostics"]=>
        array(4) {
          ["url"]=>
            array(2) {
              ["execution-time"]=>
                string(1) "9"
              ["content"]=>
                string(89)
"http://social.yahooapis.com/v1/users.guid(UQIDZJNWNLQD4GXZ5NGMZUSTQ4)/profile"

            }
          ["user-time"]=>
            string(2) "11"
          ["service-time"]=>
            string(1) "9"
          ["build-version"]=>
            string(16) "2008.10.06.06:05"
        }
      ["results"]=>
        array(1) {
          ["profile"]=>
            array(16) {
```

```

["xmlns"]=>
string(41) "http://social.yahooapis.com/v1/schema.rng"
["yahoo"]=>
string(36) "http://www.yahooapis.com/v1/base.rng"
["uri"]=>
string(70)
"http://social.yahooapis.com/v1/user/UQIDWJWVNQD4GXZ5NGMZUSTQ4/profile"

["guid"]=>
string(26) "UQIDWJWVNQD4GXZ5NGMZUSTQ4"
["created"]=>
string(20) "2008-09-16T05:06:29Z"
["familyName"]=>
string(2) "Me"
["gender"]=>
string(1) "M"
["givenName"]=>
string(2) "Me"
["image"]=>
array(4) {
    ["height"]=>
        string(3) "192"
    ["imageUrl"]=>
        string(102)
        "http://F3.yahofs.com/coreid/401eb0c8b12ezou3spl/t030Lz0rc6luW9s.2tNESY-/1/t192.jpg?ciATxQB_G&KUNdS"

    ["size"]=>
        string(7) "192x192"
    ["width"]=>
        string(3) "192"
}
["interests"]=>
array(2) {
    ["declaredInterests"]=>
        string(13) "Cloud bathing"
    ["interestCategory"]=>
        string(13) "prfFavHobbies"
}
["lang"]=>
string(5) "en-US"
["location"]=>
string(17) "San Francisco, CA"
["nickname"]=>
string(11) "blisterHead"
["profileUrl"]=>
string(54)
"http://profiles.yahoo.com/u/UQIDWJNWIQ7NQD4GXZ5NGMZUSTQ4"
["timeZone"]=>
string(19) "America/Los_Angeles"
["isConnected"]=>
string(4) "true"
}
}
}

```

```
}
```

## Source Code

```
<xi:include></xi:include>
```

# YQL INSERT: WordPress Open Application

## Summary

This code example shows how to use the YQL INSERT statement in an Open Application to post to your WordPress blog. YML is used to create a simple HTML form for the UI, and PHP is used to call the YQL Web service to run the YQL INSERT statement.

## Prerequisites

- [YQL INSERT and DELETE Statements](#)
- [HTML Form in a Canvas View](#)<sup>36</sup>
- [Getting Started: Build Your First Open Application](#)<sup>37</sup>
- [Install Yahoo! Social SDK for PHP](#)<sup>38</sup>
- [Host a WordPress blog](#)<sup>39</sup> or [register for a WordPress.com account](#)<sup>40</sup>

## Small View: yml:form

The WordPress Open Application uses the YML tag `yml:form` for entering information. The attribute `params` is like the HTML form attribute `action`. In the code snippet below, the user triggers the call to the script `yql_insert_wordpress.php` when submitting the form by clicking "Publish".

```
<yml:form name="wordpress" params="yql_insert_wordpress.php"
method="POST" insert="blog_sect">
  <fieldset>
    <b>Username:</b> <input type="text" name="username"/><br/>
    <b>Password:</b> <input type="password" name="password" /><br/>
    <b>Blog URL:</b> <input type="text" name="blogurl" /><br/>
    <input type="hidden" name="blog_form"/>
    <p><b>Title:</b> <input type="text" name="title"/></p>
    <p><b>Add New Post:</b><br/>
    <textarea name="blog" rows='5' cols='60'>Enter your blog post
here.</textarea><br/></p>
    <input type="submit" name="submit" value="Publish" />
```

---

<sup>36</sup> <http://developer.yahoo.com/yap/guide/form.html>

<sup>37</sup> [http://developer.yahoo.com/yap/guide/creating\\_open\\_app.html](http://developer.yahoo.com/yap/guide/creating_open_app.html)

<sup>38</sup> <http://developer.yahoo.com/social/sdk/php>

<sup>39</sup> <http://wordpress.org/download/>

<sup>40</sup> <http://en.wordpress.com/signup/>

```
</fieldset>
</yaml:form>
```

The attribute `insert` places the returned response from `yql_insert_wordpress.php` into the `div` tag `"blog_sect"`. If "Publish" is clicked again, the contents in this `div` will be replaced with the new content returned by `yql_insert_wordpress.php`.

```
<yaml:form name="wordpress" params="yql_insert_wordpress.php"
method="POST" insert="blog_sect">
...
</yaml:form>
<div id="blog_sect"></div>
```

## YQL: INSERT

The syntax for the YQL INSERT statement follows that of the SQL INSERT statement:

```
INSERT INTO (table) (list of comma separated field names) VALUES (list
of comma separated values)
```

Try running the following [YQL INSERT statement](#)<sup>41</sup> that references the WordPress Open Table and then view the results at <http://yqlblog.wordpress.com><sup>42</sup>.

```
USE 'http://www.datatables.org/wordpress/wordpress.post.xml' AS word-
press.post; INSERT INTO wordpress.post(title, description, blogurl,
username, password) VALUES ("YQL meets WordPress", "Posting with YQL",
"http://yqlblog.wordpress.com", "yqlblog", "password")
```

## Calling the YQL Web Service

### Building the URL

The URL for making a request to the YQL Web service has a base URL and a query string. The query string contains the YQL statement, the requested response format, and any environment files. The environment file at <http://datatables.org/alltables.env> includes the USE statement for the WordPress Table, which creates the alias `wordpress.post` as seen below:

```
use 'http://www.datatables.org/wordpress/wordpress.post.xml' as word-
press.post;
```

Below, the URL used to call the YQL Web service is divided and stored in variables to illustrate the different components: `$yql_base_url` holds the base URL to the YQL Web service, `$yql_insert` holds the YQL INSERT statement, and `$yql_format` and `$yql_env_tables` hold the request format and URL to the environment file respectively.

```
// Base URL to the YQL Web service
$yql_base_url = 'https://query.yahooapis.com/v1/public/yql';

// YQL Insert Statement
```

---

<sup>41</sup> [http://developer.yahoo.com/yql/console/?q=insert%20into%20wordpress.post%20\(title%2C%20description%2C%20blogurl%2C%20username%2C%20password\)%20values%20\(%22YQL%20meets%20WordPress%22%2C%20%22Posting%20with%20YQL%22%2C%20%22http%3A%2F%2Fyqlblog.wordpress.com%22%2C%20%22yqlblog%22%2C%20%22password%22\)&env=http://datatables.org/alltables.env](http://developer.yahoo.com/yql/console/?q=insert%20into%20wordpress.post%20(title%2C%20description%2C%20blogurl%2C%20username%2C%20password)%20values%20(%22YQL%20meets%20WordPress%22%2C%20%22Posting%20with%20YQL%22%2C%20%22http%3A%2F%2Fyqlblog.wordpress.com%22%2C%20%22yqlblog%22%2C%20%22password%22)&env=http://datatables.org/alltables.env)

<sup>42</sup> <http://yqlblog.wordpress.com>

```
$yql_insert = "INSERT INTO
wordpress.post(title,description,blogurl,username,password) " .

"VALUES(\"$title\",\"$blog\",\"$blogurl\",\"$username\",\"$password\")";

// Request XML as the response format
$yql_format = "&format=xml&env=";

// Include environment files that references
// WordPress Open Data Table
$yql_env="http://datatables.org/alltables.env";
```

Because the YQL INSERT statement is sent via POST with cURL, we build the POST fields by concatenating the various components. We configure the cURL call to send the POST fields in [Calling the YQL Web Service \[29\]](#). Notice that in this code snippet only \$yql\_format is not URL-encoded because it contains the characters '&' and '='.

```
$post_fields = "q=" . rawurlencode($yql_insert) . $yql_format .
rawurlencode($yql_env);
```

## Sending the Request to YQL

Before making the request with cURL, let's look again at the variables that hold the base URL and the post fields from [Building the URL \[28\]](#):

```
// Base URL to the YQL Web service
$yql_base_url = 'https://query.yahooapis.com/v1/public/yql';

$post_fields = "q=" . rawurlencode($yql_insert) . $yql_format .
rawurlencode($yql_env);
```

The cURL call for this code example is fairly typical. You need to configure cURL to send the request via POST and accept POST fields, ask for a returned response, and turn off SSL verification for both host and peer certificates:

```
// Initialize the curl call with the URI to the YQL Web service
$ch = curl_init($yql_url);

// Set the request to include POST fields
// URL encode the INSERT statement and the
// URL to the environment
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $post_fields);

// Configure the request to get response
// Turn off SSL verification of peer and host certification
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, false);

// Execute the curl call and parse results
$x = curl_exec($ch);
$s = simplexml_load_string($x);
```

## Source Code

### Small View Code

```
<!-- Creates user badge -->
<yml:user-badge uid="viewer" linked="true" width="55" reflexive="false"
  useyou="true" capitalize="true"/>

<!-- Form to enter and post blog -->
<yml:form name="wordpress" params="yql_insert_wordpress.php"
method="POST" insert="blog_sect">
  <fieldset>
    <b>Username:</b> <input type="text" name="username"/><br/>
    <b>Password:</b> <input type="password" name="password" /><br/>
    <b>Blog URL:</b> <input type="text" name="blogurl" /> <br/>
    <input type="hidden" name="blog_form"/>
    <p><b>Title:</b> <input type="text" name="title"/></p>
    <p><b>Add New Post:</b><br/>
    <!-- Text area for entering blog -->
    <textarea name="blog" rows='5' cols='60'>Enter your blog post
here.</textarea><br/></p>
    <input type="submit" name="submit" value="Publish" />   </fieldset>
  </yml:form>
  <!-- Div element that holds returned response from
"yql_insert_wordpress.php" -->
  <div id="blog_sect"></div>
```

### WordPress Open Application

```
<xi:include></xi:include>
```

## Getting Updates with YQL

### Summary

This code example shows you how to use YQL with the Yahoo! Social SDK for PHP to get user updates and the updates for the user's connections. The SDK handles the OAuth authorization, and YQL fetches the user data.

To understand the following material, you should already be familiar with the topics covered in [My Social PHP Application](#)<sup>43</sup> and the [Two-Minute Tutorial \[1\]](#).

### Prerequisites

- [Getting Started: Build Your First Open Application](#)<sup>44</sup>
- [MySocial PHP Application](#)<sup>45</sup>

---

<sup>43</sup> [../yap/guide/other-code-exs.html#my\\_social](#)

<sup>44</sup> [../yap/guide/creating\\_open\\_app.html](#)

<sup>45</sup> [../yap/guide/other-code-exs.html#my\\_social](#)

## YQL Queries

The syntax of YQL is similar to that of SQL, and the YQL Web service, like a MySQL database, returns data based on queries. The YQL Web service, however, returns the data in the form of XML or JSON. For more information about making YQL queries, see [SELECT Statement \[7\]](#).

YQL has a special literal `me` that holds the GUID of the currently logged in Yahoo! user. This YQL query uses this special literal to get the data of five connections of the currently logged in Yahoo! user.

```
SELECT * FROM social.connections WHERE owner_guid=me LIMIT 5
```

From the YQL response for the query above, you can then extract the GUID for each connection. These GUIDs can then be matched against the key `guid` in a new YQL query to get the updates for each connection as shown in the code snippet below:

```
SELECT * FROM social.updates WHERE guid='GUID_of_a_connection' LIMIT 5
```

Now that you have some familiarity with YQL and its syntax, try using the [YQL Console](#)<sup>46</sup> to experiment with different YQL queries and different tables such as Flickr and Search.

## Calling the YQL Web Service with the PHP SDK

The code example uses the method `requireSession` from the class `YahooSession` to get authorization to access user data and the method `query` to call the YQL Web service. The code snippets below will focus on authorization and YQL queries.

This one line of code does a lot of work to complete the OAuth authorization. The method `requireSession` first looks for an existing session, and if it's not found, redirects the user to the Yahoo! login page to sign in. The user can then authorize the app to access private data. The return value of `requireSession` is a `YahooSession` object (a session), which you will use next to make YQL queries.

```
// Authorize the app to get user data
$session = YahooSession::requireSession($consumerKey,
$consumerKeySecret);
```

To call the YQL Web service, pass the YQL query to the method `query` from the `YahooSession` object `$session`. In the code snippet below, the results of the queries are passed to the variables `$user_updates` and `$user_connections` to be parsed for specific data.

```
// Use the SDK method 'query' from the YahooSession class
// Limit the results to 5 updates and 5 connections
$user_updates = $session->query("SELECT * FROM social.updates WHERE
guid=me LIMIT 5");
$user_connections = $session->query("SELECT * FROM social.connections
WHERE owner_guid=me LIMIT 5");
```

## Source Code

```
<xi:include></xi:include>
```

---

<sup>46</sup> <http://developer.yahoo.com/yql/console/>



# Data Scraping with YQL

## Summary

This example, [yql\\_html\\_scraper.html](#)<sup>47</sup>, uses YQL to scrape HTML from Yahoo! Finance and then creates customized output. The [YQL module for YUI](#)<sup>48</sup> is used to call the YQL Web service.

## Using the YQL Module for YUI

You can download the YQL module for YUI from [GitHub](#)<sup>49</sup> or include it in your Web page like any other JavaScript file. The steps below will show you how to use the YQL module to call the YQL Web service.

1. Include the YUI library and the YQL module in your Web page.

```
<script type="text/javascript"
src="http://yui.yahooapis.com/3.0.0/build/yui/yui-min.js"></script>
<script type="text/javascript"
src="http://yui.yahooapis.com/gallery-2010.01.27-20/build/gallery-yql/gallery-yql-min.js"></script>
```

2. [Create a YUI instance](#)<sup>50</sup> to load the Node and YQL modules.

```
YUI().use('gallery-yql', 'node', function(Y) {
    // The code for calling the YQL Web service and parsing the response
    goes here.
})
```

3. Using the html table, create a YQL query to get data from Yahoo! Finance. The variable ticker stores the stock symbol of the company given by the user. See [Using XPath in YQL Statements](#) [33] for details about the YQL query.

```
var yql_query = 'select * from html where
url="http://finance.yahoo.com/q?s=' + ticker + '"' + " and xpath=" +
"'//div[@id=" + '"yfi_headlines"' + "]/div[2]/ul/li/a'";
```

4. Instantiate a YQL object by passing the YQL query to the constructor.

```
var yql_obj = new Y.yql(yql_query);
```

5. To initiate the call to the YQL Web service, call the on method from the YQL object.

```
yql_obj.on('query', function(response) {
    // Handle and parse the response here
})
```

6. Make sure that results were returned and then parse those results.

---

<sup>47</sup> [./examples/yql\\_yui\\_html\\_scraper.html.txt](#)

<sup>48</sup> <http://davglass.github.com/yui-yql/>

<sup>49</sup> <http://github.com/davglass/yui-yql>

<sup>50</sup> <http://developer.yahoo.com/yui/3/examples/yui/yui-core.html>

```

if(response.results){
    var no_stories = response.results.a.length;
    var headlines = "<p>";

    for(var i=0;i<no_stories;i++){
        headlines+= "<a href='" + response.results.a[i].href + "'" + " +
response.results.a[i].content.split('\n').join(' ').replace(/\s+/gi,
" ") + "</a><br/>";
    }
    headlines += "</p>";
document.getElementById("results").innerHTML = headlines; }else{
    headlines += "Sorry, could not find any headlines for the ticker
symbol " + ticker + ". Please try another one.";
}

```

Because each a element is seen as row of data by YQL, you need to iterate through each a element to access the href and content child elements. [Run the query<sup>51</sup>](#) in the YQL Console to examine the data structure or the returned response in greater detail.

## Using XPath in YQL Statements

The YQL statement below is used in this code example to access the links for headline stories for the company represented by *{ticker\_sym}*. This section will examine the data source for the YQL statement and the XPath expression that extracts the links.

```

select * from html where url="http://finance.yahoo.com/q?s={ticker_sym}"
and xpath='//div[@id="yfi_headlines"]/div[2]/ul/li/a'

```

The YQL query in this example allows you to access HTML using the `html` table. YQL also has tables for accessing other data formats, such as XML, CSV, JSON, RSS, XST, etc.

```

select * from html

```

The key `url` determines the data source for the YQL query. The data source in this example is Yahoo! Inc (YHOO) page on Yahoo! Finance.

```

url="http://finance.yahoo.com/q?s=yhoo"

```

The XPath statement below extracts the bulleted links from the third div child of the div with the id of "yfi\_headlines". Each link (a element) is returned as a row (repeated element in XML).

```

xpath='//div[@id="yfi_headlines"]/div[2]/ul/li/a'

```

For more information about XPath, see [WC3 XPath Language<sup>52</sup>](#).

## Source Code

```
<xi:include></xi:include>
```

<sup>51</sup> <http://y.ahoo.it/ogX/hpXo>

<sup>52</sup> <http://www.w3.org/TR/xpath/>

## Sample Open Data Tables

The following table has links to examples of YQL Open Data Tables:

Sample Code	Description
<a href="#">Flickr Photo Search [14]</a>	Allows you to retrieve data from a Flickr photo search.
<a href="#">Access to Digg Events using Gnip [16]</a>	Allows you to retrieve Digg activities using the Gnip API.
<a href="#">Hello World Table [45]</a>	Allows you to search a fictional table in which "a" is the path and "b" is the term.
<a href="#">Yahoo! Messenger Status [46]</a>	Allows you to see the status of a Yahoo! Messenger user.
<a href="#">OAuth Signed Request to Netflix [47]</a>	Allows you to make a two-legged OAuth signed request to Netflix.
<a href="#">Request for a Flickr frob [48]</a>	Returns the Flickr frob, which is analogous to the request token in OAuth.
<a href="#">Celebrity Birthday Search using IM-DB [49]</a>	Retrieves information about celebrities whose birthday is today by default, or optionally on a specific date.
<a href="#">Share Yahoo! Applications [53]</a>	Provides a list of Yahoo! Applications that you and your friends have installed, indicating whether each app is installed exclusively by you, your friends, or both.
<a href="#">CSS Selector for HTML [55]</a>	Allows you to filter HTML using CSS selectors.
<a href="#">Bit.ly Shorten URL (INSERT) [26]</a>	Shortens a long URL into a bit.ly link.

## YQL Screencasts

A screencast is a multimedia demonstration of an application, a tool, or a service. Screencasts guide you through Yahoo! APIs and related tools using audio and video.

The following screencasts are related to YQL:

Screencast	Description	Author/Source
<a href="#">Introducing YQL<sup>53</sup></a>	This is an introductory screencast to YQL using the YQL Console.	Jatin Billimoria, YDN
<a href="#">YDN Screencast: YQL<sup>54</sup></a>	Discusses how to access public and private Web services, filter and join results, extract data from web pages, and extend YQL to query additional data sources.	Dan Theurer and Jonathan Trevor, YDN
<a href="#">YQL Execute<sup>55</sup></a>	This is an intermediate-level screencast that discusses Open Data Tables and the use of JavaScript to manipulate data within Open Data Tables.	Sam Pullara and Nagesh Susarla, YDN
<a href="#">Screencast: Building an online profile of distributed data with YQL<sup>56</sup></a>	This advanced-level screencast shows you how to use YQL, a YUI CSS grid, a few dozen lines of	Christian Heilmann, <a href="http://www.wait-till-i.com">http://www.wait-till-i.com</a>

<sup>53</sup> [http://developer.yahoo.com/yos/screencasts/yql\\_screencast.html](http://developer.yahoo.com/yos/screencasts/yql_screencast.html)

<sup>54</sup> [http://developer.yahoo.net/blogs/theater/archives/2009/03/yn\\_screencast\\_yql.html](http://developer.yahoo.net/blogs/theater/archives/2009/03/yn_screencast_yql.html)

<sup>55</sup> [http://developer.yahoo.net/blogs/theater/archives/2009/04/yql\\_execute\\_screencast.html](http://developer.yahoo.net/blogs/theater/archives/2009/04/yql_execute_screencast.html)

<sup>56</sup> <http://www.wait-till-i.com/2009/04/15/screencast-building-an-online-profile-of-distributed-data-with-yql/>

Screencast	Description	Author/Source
	PHP, and a bit of CSS to create your own personal online profile.	
<a href="#">YQL Console Update</a> <sup>57</sup>	This screencast discusses updates to the YQL Console, including the REST query box and query aliases.	Paul Donnelly and Jatin Billimoria, YDN
<a href="#">YQL Query Builder</a> <sup>58</sup>	This screencast discusses the query builder feature, which allows developers who are new to YQL to explore tables and create queries easily and quickly.	Paul Donnelly and Jatin Billimoria, YDN

## YQL Slideshows

The following slideshows discuss YQL and its usage:

Slideshow	Description	Author/Source
<a href="#">YQL - A Query Language for the Web</a> <sup>59</sup>	Jonathan Trevor, YQL Lead, gives an overview of the YQL Web service and YQL language.	Jonathan Trevor, Slide-share
<a href="#">Open Hack London - Introduction to YQL</a> <sup>60</sup>	This narrated slideshow introduces YQL and uses code example to show its main features.	Christian Heilmann, Slideshare
<a href="#">YQL, Flickr, OAuth, YAP</a> <sup>61</sup>	This slideshow offers a step-by-step explanation of how to use YQL and OAuth together to get Flickr photos.	Erik Eldridge, Slideshare
<a href="#">BayJax July 2009 - Browser MVC with YQL &amp; YUI</a> <sup>62</sup>	In this slideshow, you learn how to use YQL and YUI together to create widgets. YQL retrieves the data (the model layer), and YUI handles the presentation and control layers.	Jonathan LeBlanc, Slide-share

## Additional Tutorials and Code Examples

The following advanced tutorials and code examples show specialized uses for YQL:

Tutorial/Code Example	Description	Author/Source
<a href="#">Building a (re)search interface for Yahoo, Bing and Google with YQL</a> <sup>63</sup>	Uses a single YQL statement with an Open Data Table to make queries to three search engines.	Chris Heilmann, <a href="http://www.wait-till-i.com">http://www.wait-till-i.com</a>
<a href="#">Getting stock information with YQL and open data tables</a> <sup>64</sup>	Turns Yahoo! Finance into an API with executable JavaScript in an Open Data Table.	YQL Team, <a href="http://www.yql-blog.net/blog">http://www.yql-blog.net/blog</a>
<a href="#">Tutorial: scraping and turning a website into a widget with YQL</a> <sup>65</sup>	Scrapes a website with YQL and XPath to make a TV fun facts widget.	Chris Heilmann, <a href="http://www.wait-till-i.com">http://www.wait-till-i.com</a>

<sup>57</sup> [http://developer.yahoo.com/blogs/ydn/posts/2010/03/yql\\_console\\_changes/](http://developer.yahoo.com/blogs/ydn/posts/2010/03/yql_console_changes/)

<sup>58</sup> <http://developer.yahoo.com/blogs/ydn/posts/2010/10/yql-query-builder-and-explorer/>

<sup>59</sup> <http://www.slideshare.net/sh1mmer/yql-a-query-language-for-the-web>

<sup>60</sup> <http://www.slideshare.net/cheilmann/open-hack-london-introduction-to-yql>

<sup>61</sup> <http://www.slideshare.net/erikeldridge/yql-flickr-oauth-yap>

<sup>62</sup> <http://www.slideshare.net/jcleblanc/bayjax-july-2009-browser-mvc-with-yql-yui#notesList>

<sup>63</sup> <http://www.wait-till-i.com/2009/12/09/building-a-research-interface-for-yahoo-bing-and-google-with-yql/>

<sup>64</sup> <http://www.yqlblog.net/blog/2009/06/02/getting-stock-information-with-yql-and-open-data-tables/>

<sup>65</sup> <http://www.wait-till-i.com/2009/08/25/tutorial-scraping-and-turning-a-web-site-into-a-widget-with-yql/>

## Additional References and Resources

The following references and resources for YQL include libraries, handouts, blog posts, and demos:

Reference/Resource	Description	Author/Source
<a href="http://yuilib.com">YQL Module for YUI</a> <sup>66</sup>	JavaScript library for making queries to the YQL Web service.	Dav Glass, <a href="http://yuilib.com">http://yuilib.com</a>
<a href="http://muffinresearch.co.uk/archives/2009/11/24/python-yql-client-library-for-yql/">Python-YQL: client library for YQL</a> <sup>67</sup>	Python library for making queries to the YQL Web service.	Stuart Colville, <a href="http://muffinresearch.co.uk">http://muffinresearch.co.uk</a>
<a href="http://developer.yahoo.com/yql/yql-users-v0.0.2.pdf">Using YQL for Developers (PDF)</a> <sup>68</sup>	Handout that briefly introduces YQL and serves as a cheat sheet.	YDN, <a href="http://developer.yahoo.com">http://developer.yahoo.com</a>
<a href="http://developer.yahoo.com/yql/yql-opendatatables-v0.0.2.pdf">Creating Open Data Tables for YQL (PDF)</a> <sup>69</sup>	Handout that introduces and provides examples of YQL Open Data Tables.	YDN, <a href="http://developer.yahoo.com">http://developer.yahoo.com</a>
<a href="http://developer.yahoo.net/blog/archives/2009/07/yql_insert.html">YQL: INSERT INTO inter-net</a> <sup>70</sup>	YDN blog post that discusses the YQL INSERT INTO statement. YQL examples demo links are also given.	Jonathan Trevor, YDN
<a href="http://icant.co.uk/goohooobi/">GooHooBi Search</a> <sup>71</sup>	YQL-powered application that returns search results from Google, Yahoo!, and Bing.	Chris Heilmann, <a href="http://icant.co.uk">http://icant.co.uk</a>
<a href="http://developer.yahoo.com/yui/theater/">YQL and YUI: Building Blocks for Quick Applications</a> <sup>72</sup>	Video of Chris Heilmann talking about how to use YQL and YUI to simplify Web development.	Chris Heilmann, <a href="http://developer.yahoo.com/yui/theater/">YDN Theater</a> <sup>73</sup>
<a href="http://developer.yahoo.net/blogs/theater/archives/2010/03/yql_console_changes.html">YQL Console Update - Paul Donnelly</a> <sup>74</sup>	Paul Donnelly from the YQL team discusses changes to the YQL console including Query Aliasing.	Paul Donnelly, YDN

<sup>66</sup> <http://yuilib.com/gallery/show/yql>

<sup>67</sup> <http://muffinresearch.co.uk/archives/2009/11/24/python-yql-client-library-for-yql/>

<sup>68</sup> <http://developer.yahoo.com/yql/yql-users-v0.0.2.pdf>

<sup>69</sup> <http://developer.yahoo.com/yql/yql-opendatatables-v0.0.2.pdf>

<sup>70</sup> [http://developer.yahoo.net/blog/archives/2009/07/yql\\_insert.html](http://developer.yahoo.net/blog/archives/2009/07/yql_insert.html)

<sup>71</sup> <http://icant.co.uk/goohooobi/>

<sup>73</sup> <http://developer.yahoo.com/yui/theater/>

<sup>72</sup> <http://developer.yahoo.com/yui/theater/video.php?v=heilmann-yql>

<sup>74</sup> [http://developer.yahoo.net/blogs/theater/archives/2010/03/yql\\_console\\_changes.html](http://developer.yahoo.net/blogs/theater/archives/2010/03/yql_console_changes.html)

# Using YQL and Open Data Tables

---

## Using YQL and Open Data Tables

### Abstract

This part of the YQL Guide focuses on the basics of using YQL and Open Data Tables. It covers `SELECT` statements and YQL response data.

---

# Table of Contents

1. Overview for YQL Users .....	1
How to Run YQL Statements .....	1
The Two-Minute Tutorial .....	1
YQL Web Service URLs .....	2
YQL Query Parameters .....	3
YQL Query Aliases .....	3
Summary of YQL Statements .....	5
Authorization .....	5
Best Practices for Forming YQL Statements .....	6
2. SELECT Statement .....	7
Introduction to SELECT .....	7
Syntax of SELECT .....	7
Specifying the Elements Returned (Projection) .....	8
Filtering Query Results (WHERE) .....	8
Remote Filters .....	9
Local Filters .....	10
Combining Filter Expressions (AND, OR) .....	11
Joining Tables With Sub-selects .....	11
Paging and Table Limits .....	12
Remote Limits .....	12
Local Limits .....	12
Sort and Other Functions .....	13
Remote and Local Processing .....	14
Example With Remote and Local Steps .....	14
Summary of Remote and Local Controls .....	15
GUIDs, Social, and Me .....	16
Variable Substitution in the GET Query String .....	16
Extracting HTML .....	16
Parsing HTML 4.01 Versus HTML5 .....	17
Using XPath .....	17
Selecting Content .....	17
3. Using the Query Builder .....	19
YQL Query Builder Overview .....	19
Query Builder Usage .....	20
Loading a Query .....	20
Entering Values .....	21
4. INSERT, UPDATE, and DELETE (I/U/D) Statements .....	22
Introduction to INSERT, UPDATE, and DELETE (I/U/D) Statements .....	22
Bindings Required for I/U/D .....	22
JavaScript Methods Available to I/U/D .....	24
Syntax of I/U/D .....	24
Syntax of INSERT .....	24
Syntax of UPDATE .....	24
Syntax of DELETE .....	25
Limitations of I/U/D .....	25
Open Data Table Examples of I/U/D .....	26
Bit.ly Shorten URL (INSERT) .....	26
WordPress Post (INSERT) .....	27
5. Response Data .....	36
Supported Response Formats .....	36
JSONP-X: JSON envelope with XML content .....	36



Structure of Response .....	38
Information about the YQL Call .....	38
XML-to-JSON Transformation .....	39
JSON-to-JSON Transformation .....	40
Errors and HTTP Response Codes .....	41
6. Using YQL Open Data Tables .....	43
Overview of Open Data Tables .....	43
Invoking an Open Data Table Definition within YQL .....	43
Invoking a Single Open Data Table .....	43
Invoking Multiple Open Data Tables .....	44
Invoking Multiple Open Data Tables as an Environment .....	44
Working with Nested Environment Files .....	45
Setting Key Values for Open Data Tables .....	46
Using SET to Hide Key Values or Data .....	47

---

## List of Figures

3.1. YQL Query Builder .....	45
6.1. Environment File Transversal .....	45

---

# Chapter 1. Overview for YQL Users

In this Chapter:

- [“How to Run YQL Statements” \[1\]](#)
- [“The Two-Minute Tutorial” \[1\]](#)
- [“YQL Web Service URLs” \[2\]](#)
- [“Summary of YQL Statements” \[5\]](#)
- [“Authorization” \[5\]](#)

## How to Run YQL Statements

You can run [YQL statements \[5\]](#) in the following ways:

- **YQL Console:** In your browser with the [YQL Console<sup>1</sup>](#).
- **SELECT statements using HTTP GET:** A Web application can use an HTTP GET request when running SELECT statements, specifying the YQL statement as a query parameter of the [Web Service URL \[2\]](#).
- **INSERT, UPDATE, and DELETE statements using HTTP POST, PUT, or DELETE:** A Web application can similarly use an HTTP GET, PUT, or DELETE for INSERT, UPDATE, and DELETE statements. The only exception is when you specify a JSONP callback, in which case you can use an HTTP GET request and specify a callback query parameter on the GET URI.
- **PHP SDK:** From a Web application that uses the [PHP SDK<sup>2</sup>](#), by calling the `query` method of the `YahooSession` class.

## The Two-Minute Tutorial

This tutorial shows you how to run YQL statements and examine the resulting data with the [YQL Console<sup>3</sup>](#).

1. In your browser, run the [YQL Console<sup>4</sup>](#).
2. Under "Example Queries", click **get 10 flickr "cat" photos**. The console calls the YQL Web Service with the following query:

```
select * from flickr.photos.search where text="Cat" limit 10
```

This SELECT statement requests Flickr photos of cats. The "\*" indicates that all fields of the `flickr.photos.search` table will be returned. In the filter of the WHERE clause, the string "Cat" is the value of the search query.

---

<sup>1</sup> <http://developer.yahoo.com/yql/console/>

<sup>2</sup> <http://developer.yahoo.com/social/sdk/#php>

<sup>3</sup> <http://developer.yahoo.com/yql/console/>

<sup>4</sup> <http://developer.yahoo.com/yql/console/>

3. Note the XML response in the **FORMATTED VIEW** tab. The information from the `flickr.photos.search` table is in the `results` element. To get the response in JSON format, select the **JSON** radio button and click **TEST**.

4. In **Your YQL Statement**, replace the "\*" with the `owner` and `title` fields:

```
select owner, title from flickr.photos.search where text="Cat" limit 10
```

Make sure that `owner` and `title` are lowercase. Unlike SQL, in YQL the field and table names are case sensitive.

5. To run the command in **Your YQL Statement**, click **TEST**. The returned photo fields should only have the `owner` and `title` attribute fields.

6. In the console, examine the URL below **REST query**:

```
http://query.yahooapis.com/v1/public/yql?q=select%20owner%20%20%20title%20%20%20from%20%20flickr.photos.search%20where%20text%3D%22Cat%22%20limit%2010&format=json&callback=cbfunc
```

To call the YQL Web Service, an application would call an HTTP GET method on this URL. The `q` parameter in the URL matches the SELECT statement displayed under **Your YQL Statement** (except that characters such as spaces are URL encoded). The **COPY URL** button copies this URL to your clipboard, so that you can paste it into the source code of an application.

7. To view YQL's pre-defined tables, expand the **Data Tables** list on the right side of the console. You can run an example query on each of these tables by clicking the table name.
8. Advanced: To view the description of a table, under **Data Tables**, expand the **flickr** menu to see all of the Flickr tables. Move your mouse cursor over the table name `flickr.photos.search`, then click **desc**. On the **TREE VIEW** tab, take a look at these nodes: "query->results->table". The nodes under the "table" node contain information such as meta-data and search fields (input keys). For more information on input keys, see [Remote Filters \[9\]](#).

## YQL Web Service URLs

The YQL Web Service has two URLs. The following URL allows access to public data, which does not require authorization:

```
http://query.yahooapis.com/v1/public/yql?[query_params]
```

The next URL requires authorization by OAuth and allows access to both public and private data:

```
http://query.yahooapis.com/v1/yql?[query_params]
```

The following URLs are for accessing data from Open Data Tables configured to use [YQL streaming<sup>5</sup>](#). The first URL is used to get public data, and the second requires authorization by OAuth and allows access to both public and private data.

```
http://query.yahooapis.com/v1/public/streaming/yql
```

---

<sup>5</sup>[\\_yql-odt-streaming.html](#)

`http://query.yahooapis.com/v1/streaming/yql`



## Note

The public URL has a lower rate limit than the OAuth-protected URL. Therefore, if you plan to use YQL heavily, you should access the OAuth-protected URL.

## YQL Query Parameters

The following table lists the query parameters for the URLs of the YQL Web Service.

Query Parameter	Required?	Default	Description
q	Yes	(none)	The YQL statement to execute, such as SELECT.
format	No	xml	The format of the results of the call to the YQL Web Service. Allowed values: xml or json.
callback	No	(none)	The name of the JavaScript callback function for JSONP format. If callback is set and if format=json, then the response format is JSON. For more information on using XML instead of JSON, see <a href="#">JSONP-X [36]</a> .
diagnostics	No	true	Diagnostic information is returned with the response unless this parameter is set to false.
debug	No	(none)	Enables network-level logging of each network call within a YQL statement or API query.  For more information, see, <a href="#">Logging Network Calls in Open Data Tables [12]</a> .
env	No	(none)	Allows you to use multiple Open Data Tables through a <a href="#">YQL environment file [44]</a> .
jsonCompat	No	(none)	Enables lossless JSON processing. The only allowed value is new. See <a href="#">JSON-to-JSON Transformation [40]</a> .

## YQL Query Aliases

To make YQL Web Service queries easier to remember and share, you can shorten them with query aliases. Take for example a normal YQL REST query:

`http://query.yahooapis.com/v1/yql?q=select+*+from+weather.forecast+where+location%3D%40zip`

The corresponding query alias can look like this:

`http://query.yahooapis.com/v1/public/yql/jonathan/weather?zip=94025`

## Creating Queries Aliases

The easiest way to create query aliases is through the [YQL console](#)<sup>6</sup>, though you can also use some of the more advanced functions through the [yql.queries](#)<sup>7</sup> and [yql.queries.query](#)<sup>8</sup> tables.

<sup>6</sup> <http://developer.yahoo.com/yql/console/>

<sup>7</sup> <http://developer.yahoo.com/yql/console/#h=desc%20yql.queries>

<sup>8</sup> <http://developer.yahoo.com/yql/console/#h=desc%20yql.queries.query>

To create a short query alias based on a statement, follow these steps:

1. Go the [YQL console](http://developer.yahoo.com/yql/console/)<sup>9</sup>.
2. Sign in with your Yahoo! ID if you haven't already. Click on the **Sign In** link, which is located along the top of the page. Signing in is required for saving and accessing query aliases.
3. Using the YQL console, create a YQL statement or Web Service query that you want to save as a query alias.
4. Click on the **Create Query Alias** link, which is located in the upper-right corner of the YQL statement box.
5. When first creating a query alias, type in a prefix that will be associated with all your query aliases. For example, if you choose "prefix," each query alias you create will start like this:

```
http://query.yahooapis.com/v1/public/yql/prefix/
```



### Tip

If you decide later that you want to start over with a new prefix for your query aliases, you can run the following YQL statement to delete your existing prefix:

```
delete from yql.queries
```

Keep in mind that deleting your prefix also deletes all its associated query aliases.

6. Click **Next** and then type the actual name of the query alias. This alias name is appended to the end of your query after the prefix. If you choose "alias", the resulting query will look like this:

```
http://query.yahooapis.com/v1/public/yql/prefix/alias/
```

7. After reviewing the public URL and the Authenticated URL, click **Close** to return to the YQL console.

All of your saved query aliases appear along the right side the YQL console under the **Query Aliases** heading. There, you can click on a alias name to see it in the REST query box. You can also click on the red **X** next to a particular alias name to delete it.

## Variable Substitution with Query Aliases

Query aliases support [variable substitution](#) [16] using the @ symbol. For example, the following YQL statement searches for music artists who contain the name Michael Jackson:

```
select * from music.artist.search where keyword="Michael Jackson"
```

Using variable substitution, you can replace Michael Jackson with the variable `artist`. Here is an example:

```
select * from music.artist.search where keyword=@artist
```

A corresponding query alias with the required parameter can look like this:

```
http://query.yahooapis.com/v1/public/yql/prefix/musicartist?artist="Michael Jackson"
```

---

<sup>9</sup><http://developer.yahoo.com/yql/console/>

# Summary of YQL Statements

The following table lists all YQL statements:

Statement	Example	Description
SELECT	SELECT * FROM social.profile WHERE guid=me	Retrieves data from the specified table. See the <a href="#">SELECT Statement chapter [7]</a> for more information.
INSERT	INSERT INTO table (key1, key2, key3) VALUES ('value1', 'value2', 'value3')	Inserts data into the specified table. See the <a href="#">INSERT, UPDATE, DELETE statements [22]</a> chapter for more information.
UPDATE	UPDATE (table) SET field1=value WHERE filter	Updates data in the specified table. See the <a href="#">INSERT, UPDATE, DELETE statements [22]</a> chapter for more information.
DELETE	DELETE FROM (table) WHERE filter	Deletes data in the specified table. See the <a href="#">INSERT, UPDATE, DELETE statements [22]</a> chapter for more information.
SHOW TABLES	SHOW TABLES	Gets a list of the tables available in YQL.
DESC	DESC social.connections	Gets a description of the table.
USE	USE "http://myserver.com/mytables.xml" AS mytable;  SELECT * FROM mytable WHERE...	Maps a table name to the URL of an <a href="#">Open Data Table [43]</a> .
SET	SET (name)=(value);  SELECT * FROM mytable WHERE...	Allows you to <a href="#">set up key values [46]</a> for use within Open Data Tables

## Authorization

A YQL table contains either public or private data. An example of public data is search information, such as the `local.search` table. An application can access a public table through the `/v1/public/yql` endpoint, which does not require authorization. (For the full endpoint, see [YQL Web Service URLs \[2\]](#).)

A user's personal information, such as the `social.contacts` table, is private. Access to private data requires the user's approval. To access a private table, an application must use OAuth and the `/v1/yql` endpoint. YQL supports [two-legged](#)<sup>10</sup> and [three-legged](#)<sup>11</sup> OAuth.

For YQL code examples with OAuth, see the [YQL Code Examples \[18\]](#). For details, see the [Yahoo! OAuth Quick Start Guide](#)<sup>12</sup> and the [OAuth site](#)<sup>13</sup>.

<sup>10</sup> <http://developer.yahoo.com/yos/glossary/gloss-entries.html#two-legged-authorization>

<sup>11</sup> <http://developer.yahoo.com/yos/glossary/gloss-entries.html#three-legged-authorization>

<sup>12</sup> <http://developer.yahoo.com/oauth/guide/index.html>

<sup>13</sup> <http://oauth.net/>

# Best Practices for Forming YQL Statements

Because YQL statements may contain user input and are passed as a query string parameter to the YQL Web Service URL, it is important to take some measures to make sure that they are properly formed. You can avoid malformed YQL statements that will lead to errors or potentially contain harmful code by following the guidelines below:

- Constrain and sanitize any user input that will be used as a key value.
- Confirm that the values to input keys are enclosed in quotation marks.
- URL-encode the YQL statement.
- Use [variable substitution \[16\]](#) for the values of input keys in the YQL statement.
- For YQL statements that you plan to use often, use [query aliases \[3\]](#), which can also be used with variable substitution.



---

# Chapter 2. SELECT Statement

## In this Chapter

- [“Introduction to SELECT” \[7\]](#)
- [“Syntax of SELECT” \[7\]](#)
- [“Specifying the Elements Returned \(Projection\)” \[8\]](#)
- [“Filtering Query Results \(WHERE\)” \[8\]](#)
- [“Joining Tables With Sub-selects” \[11\]](#)
- [“Paging and Table Limits” \[12\]](#)
- [“Sort and Other Functions” \[13\]](#)
- [“Remote and Local Processing” \[14\]](#)
- [“GUIDs, Social, and Me” \[16\]](#)
- [“Variable Substitution in the GET Query String” \[16\]](#)
- [“Extracting HTML” \[16\]](#)

## Introduction to SELECT

The SELECT statement of YQL retrieves data from YQL tables. The YQL Web Service fetches data from a back-end datasource (often a Web service), transforms the data, and returns the data in either XML or JSON format. Table rows are represented as repeating XML elements or JSON objects. Columns are XML sub-elements or attributes, or JSON name-value pairs. To try out some SELECT examples and to view the results, run the [YQL Console](#)<sup>1</sup> and click the items under "Example Queries" or "Data Tables".

## Syntax of SELECT

The YQL SELECT statement has the following syntax:

```
SELECT what FROM table WHERE filter [| function]
```

The `what` clause contains the fields (columns) to retrieve. The fields correspond to the XML elements or JSON objects in the data returned by the SELECT. An asterisk (the "\*" character) in the `what` clause means all fields. The `table` is either the YQL pre-defined or Open Data Table that represents a datasource. (Unlike in SQL, in YQL only one table can be specified.) The `filter` is a comparison expression that limits the rows returned. The results of the SELECT can be piped to an optional `function`, such as `sort`.

In YQL, statement keywords such as SELECT and WHERE are case-insensitive. Table and field names are case sensitive. In string comparisons, the values are case sensitive. String literals must be enclosed in quotes; either double or single quotes are allowed.

---

<sup>1</sup><http://developer.yahoo.com/yql/console/>

## Specifying the Elements Returned (Projection)

To get a vertical slice (projection) of a table, specify the fields in the clause following the SELECT keyword. In YQL, these fields are analogous to the columns of a SQL table. Multiple fields are delimited by commas, for example:

```
select lastUpdated, itemurl from social.updates where guid=me
```

To get all fields, specify an asterisk:

```
select * from social.updates where guid=me
```

If the fields in the result set contain sub-fields, you can indicate the sub-fields by using periods (dots) as delimiters. (Sometimes this format is called "dot-style syntax.") For example, for the `social.profile` table, to get only the `imageUrl` sub-field of the `image` field, enter the following:

```
select image.imageUrl from social.profile where guid=me
```

The following lines show part of the XML response for this SELECT. Note that only the `imageUrl` sub-field is returned.

```
. . .
<results>
  <profile xmlns="http://social.yahooapis.com/v1/schema.rng">
    <image>

<imageUrl>http://l.yimg.com/us.yimg.com/i/identity/nopic_192.gif</imageUrl>

    </image>
  </profile>
</results>
```

If you specify one or more non-existent fields in the `what` clause, the HTTP response code is 200 OK. If none of the fields in the `what` clause exist, the result set is empty. (That is, zero rows are returned.) Note that field names are case sensitive.

## Filtering Query Results (WHERE)

The filter in the WHERE clause determines which rows are returned by the SELECT statement. The filter in the following statement, for example, returns rows only if the `text` field matches the string `Barcelona`:

```
select * from flickr.photos.search where text='Barcelona'
```

Filters can also be used to return rows if a field is contained in a list of values. The filter in this statement returns rows if the `location` field matches "San Francisco" or "San Jose".

```
select * from upcoming.events where location in ( "San Francisco", "San Jose" )
```

YQL has two types of filters: [remote \[9\]](#) and [local \[10\]](#). These terms refer to where the filtering takes place relative to the YQL Web Service.

## Remote Filters

With a remote filter, the filtering takes place in the back-end datasource (usually a Web service) called by the YQL Web Service.

A remote filter has the following syntaxes:

```
input_key=literal
```

```
input_key IN [list of one or more literals]
```

The input key is a parameter that YQL passes to the back-end datasource. The literal is a value, either a string, integer, or float. The equality (=) operator and the IN operator are allowed in a remote filter. (A [local filter \[10\]](#), in contrast, can contain other types of comparison operators.)

For example, in the following statement, the input key is `photo_id`:

```
select * from flickr.photos.info where photo_id='2186714153'
```

For this SELECT statement, the YQL Web Service calls the Flickr Web Service, passing `photo_id` as follows:

```
http://api.flickr.com/services/rest/?method=flickr.photos.getInfo&photo_id='2186714153'
```

Using the IN operator, the following statement returns the latest percentage change of the stock prices for Yahoo!, Oracle, and Cisco.

```
select * from finance.scrape.trend where name IN ("Yahoo!", "Oracle", "Cisco")
```

Most YQL tables require the SELECT statement to specify a remote filter, which requires an input key. Often, the input key is not one of the fields included in the results returned by a SELECT. To see which input keys are allowed or required, enter the DESC statement for the YQL table and note the key XML element of the results. For example, the results of `DESC flickr.photos.info` show that the input key `photo_id` is required:

```
<results>
. . .
  <select>
    <key name="secret" type="xs:string"/>
    <key name="photo_id" required="true" type="xs:string"/>
  </select>
. . .
</results>
```

Multiple remote filters can be combined with the boolean AND or OR operators, for example:

```
select * from flickr.photos.info where photo_id='2186714153' or
photo_id='3502889956'
```

The SELECT statements for some tables require multiple remote filters, for example:

```
select * from local.search where zip='94085' and query='pizza'
```

## Local Filters

The YQL Web Service performs local filtering on the data it retrieves from the back-end datasource. Before examining the syntax of local filters, let's look at a few examples.

In the following example, YQL gets data from the `flickr.photos.interestingness` table, then applies the local filter `title='moon'`.

```
select * from flickr.photos.interestingness where title='moon'
```

In the next statement, the local filter checks that the value of the `title` field starts with the string `Chinese` or `CHINESE`.

```
select * from flickr.photos.interestingness where title like 'Chinese%'
```

The filter in the following statement contains a regular expression that checks for the substring `blue`:

```
select * from flickr.photos.interestingness where title matches
'.*blue.*'
```

The following statement returns recent photos with the IDs specified in the parentheses:

```
select * from flickr.photos.recent where id in ('3630791520',
'3630791510', '3630791496')
```

A local filter has the following syntax:

```
field comparison_operator literal
```

The `field` (column) specifies the name of the XML element or JSON object in the results. To specify a sub-field, separate the containing fields with periods. For an example sub-field, see `Rating.AverageRating` in the SELECT statement in [Combining Boolean Operations \[11\]](#). The `literal` is either a quoted string, an integer, or a float. The following table lists the allowed comparison operators.

Operator	Description
=	Equal.
!=	Not equal.
>	Greater than.
<	Less than.
>=	Greater than or equal to.
<=	Less than or equal to.
[NOT] IN	Tests whether a value is contained in a set of values. This operator can be followed by either a sub-select or by a comma-delimited set of values within parentheses.
IS [NOT] NULL	Tests for the existence of the field in the results. An IS NULL expression is true if the field is not in the results.
[NOT] LIKE	Tests for a string pattern match. The comparison is case-insensitive. The "%" character in the literal indicates zero or more characters. For example, <code>Sys%</code> matches any string starting with <code>Sys</code> .

Operator	Description
[NOT] MATCHES	Tests for a string pattern match, allowing regular expressions. The comparison is case sensitive.

## Combining Filter Expressions (AND, OR)

Local and remote filter expressions can be combined with the boolean AND and OR operators. The AND operator has precedence over the OR operator. To change precedence, enclose expressions in parentheses.

In the following example, the first two filters are remote expressions because `query` and `location` are input keys. The third filter, containing the field `Rating.AverageRating`, is a local filter.

```
select * from local.search where query="sushi" and location="san fran-
cisco, ca" and Rating.AverageRating="4.5"
```

## Joining Tables With Sub-selects

With sub-selects, you can join data across different YQL tables. (In SQL, a sub-select is usually called a "subquery.") Because YQL tables are often backed by Web services, sub-selects enable you to join data from different Web services. In a join, the sub-select provides input for the [IN operator \[11\]](#) of the outer select. The values in the outer select can be either input keys ([remote filters \[9\]](#)) or fields in the response ([local filters \[10\]](#)).

By using a sub-select, the following statement returns the profiles of all of the connections (friends) of the user currently logged in to Yahoo!. This statement joins the `social.profile` and `social.connection` tables on the values of the [GUIDs \[16\]](#). The inner SELECT, which follows the word `IN`, returns the GUIDs for the user's connections. For each of these GUIDs, the outer SELECT returns the profile information.

```
select * from social.profile where guid in (select guid from social.con-
nections where owner_guid=me)
```

Tables can be joined on multiple keys. In the following example, the `local.search` and `geo.places` tables are joined on two keys. The inner select returns two data fields (`centroid.latitude` and `centroid.longitude`) which are compared with the two input keys (`latitude` and `longitude`) of the outer select.

```
select * from local.search where (latitude,longitude) in (select
centroid.latitude, centroid.longitude from geo.places where text="north
beach, san francisco") and radius=1 and query="pizza" and location=""
```

The next example shows an inner select that returns data from an RSS feed:

```
select * from search.web where query in (select title from rss where
url="http://rss.news.yahoo.com/rss/topstories" | truncate(count=1))
```

One sub-select is allowed in each select. In other words, each select statement can only have one `IN` keyword, but the inner select may also have an `IN` keyword. The following statement is legal:

```
select * from search.siteexplorer.pages where query in (select url from
search.web where query in (select Artist.name from music.release.popular
limit 1) limit 1)
```

However, the next statement is illegal because it has two IN keywords in a select:

```
ILLEGAL: select * from flickr.photos.search where lat in (select
centroid.latitude from geo.places where text="sfo") and lon in (select
centroid.longitude from geo.places where text="sfo")
```

## Paging and Table Limits

Many YQL queries access datasources that contain thousands, or even millions, of items. When querying large datasources, applications need to page through the results to improve performance and usability. YQL enables applications to implement paging or limit table size at two levels: [remote \[12\]](#) and [local \[12\]](#).

To find out how many items (rows) a query (SELECT) returns, in an XML response, check the value of the `yahoo:count` attribute of the query element. In a JSON response, check the value of the `count` object.

The maximum number of items returned by a SELECT is 5000. The maximum processing time for a YQL statement is 30 seconds. For most tables, the default number of items returned is 10. (That is, the default is 10 if you do not specify a limit in the SELECT statement.)

## Remote Limits

A remote limit controls the number of items (rows) that YQL retrieves from the back-end datasource. To specify a remote limit, enter the offset (start position) and number of items in parentheses after the table name.

For example, in the following statement, the offset is 0 and the number of items is 10. When this statement runs, YQL calls Yahoo! Search BOSS (the back-end source for the `search.web` table) and gets the first 10 items that match the `query="pizza"` filter:

```
select title from search.web(0,10) where query="pizza"
```

The following statement gets items 10 through 30. In other words, starting at position 10, it gets 20 items:

```
select title from search.web(10,30) where query="pizza"
```

The default offset is 0. For example, the following statement gets the first 20 items:

```
select title from search.web(20) where query="pizza"
```

The default number of items for a remote limit varies with the table. For most tables, the default number of items is 10.

The maximum number of items also varies with table. To get the maximum number of items, enter 0 in parentheses after the table name. The following statement returns 1000 items from the `search.web` table:

```
select title from search.web(0) where query="pizza"
```

## Local Limits

A local limit controls the number of rows YQL returns to the calling application. YQL applies a local limit to the data set that it has retrieved from the back-end datasource. To specify a local limit, include the LIMIT and OFFSET keywords (each followed by an integer) after the WHERE clause. LIMIT specifies

the number of rows and OFFSET indicates the starting position. The OFFSET keyword is optional. The default offset is 0, which is the first row.

The following statement has a remote limit of 100 and a local limit of 15. When this statement runs, YQL gets up to 100 items from the back-end datasource. On these items, YQL applies the local limit and offset. This statement returns 15 rows to the calling application, starting with the first row (offset 0).

```
select title from search.web(100) where query="pizza" limit 15 offset 0
```

YQL retrieves items from the back-end datasource one page at a time until either the local or remote limit has been reached. The page size varies with the table. The following statement has an unbounded remote limit (0) so YQL retrieves items from the backend datasource until the the local limit of 65 is reached:

```
select title from search.web(0) where query="pizza" limit 65
```

Typically, a SELECT statement includes limits and filters, as shown in [Example With Remote and Local Steps \[14\]](#).

## Sort and Other Functions

YQL includes built-in functions such as `sort`, which are appended to the SELECT statement with the pipe symbol (`|`). These functions are applied to the result set after the SELECT statement performs all other operations, such as applying filters and limits.

In the following SELECT statement, the sub-select returns a list of GUIDs, and the outer select returns a set of profiles, one for each GUID. This set of profiles is piped to the `sort` function, which orders the results according to the value of the `nickname` field.

```
select * from social.profile where guid in (select guid from social.connections where owner_guid=me) | sort(field="nickname")
```

Multiple functions can be chained together with the pipe symbol (`|`). The following statement queries the `local.search` table for restaurants serving pizza. The results are piped to the `sort` function, then to the `reverse` function. The final result contains 20 rows, sorted by rating from high to low.

```
select Title, Rating.AverageRating from local.search(20) where query="pizza" and city="New York" and state="NY" | sort(field="Rating.AverageRating") | reverse()
```

Results can also be sorted with multiple criteria. The `sort` function can take multiple fields with optional values specifying the sort direction. For example, the results returned from the above statement querying for restaurants serving pizza can be sorted first by the average rating (primary sort) and then by the title (secondary sort). In the modified version of the statement below, the results are first sorted by the average rating in descending order and then those restaurants with the same average rating are sorted by the title in ascending order.

```
select Title, Rating.AverageRating from local.search(20) where query="pizza" and city="New York" and state="NY" | sort(field="Rating.AverageRating", descending="true", field="Title", descending="false")
```

The following table lists the YQL functions that can be appended to a SELECT statement. Function arguments are specified as name-value pairs.

Function	Argument	Example	Description
sort	field [descending] [hidden]	sort(field="nickname", descending="true", hidden="true", field="email")	Sorts the result set according to one or more criteria that are specified by field keys. The default value of the optional descending argument is false. If hidden is set to "true", then the preceding field will be omitted from the items in the result set. Syntax: sort(field_1="value" [, descending_1="true" "false"] [, hidden_1="true" "false"], ... , field_n="value" [, descending_n="true" "false"] [, hidden_n="true" "false"])
tail	count	tail(count=4)	Gets the last count items (rows).
truncate	count	truncate(count=4)	Gets the first count items (rows).
reverse	(none)	reverse()	Reverses the order of the items (rows).
unique	field [hideRepeatCount]	unique(field="Rating.AverageRating", hideRepeatCount="true")	Removes items (rows) with duplicate values in the specified field (column). The first item with the value remains in the results. If hideRepeatCount is missing or set to "false", the item that remains in the results will be annotated with the element (XML) or field (JSON) yahoo:repeatcount specifying the number of items (including the returned one) with duplicate values in the specified field.
sanitize	[field]	sanitize(field='foo')	Sanitizes the output for HTML-safe rendering. To sanitize all returned fields, omit the field parameter.

## Remote and Local Processing

When YQL runs a SELECT statement, it accesses a back-end datasource, typically by calling a Web service. [Remote filters \[9\]](#) and [limits \[12\]](#) are implemented by the back-end Web service. Local processing (including [local filters \[10\]](#) and [limits \[12\]](#)) is performed by the YQL Web Service on the data it fetches from the back-end Web service. As shown by the following example, whether an operation is remote or local affects the data returned to the application that calls the SELECT statement.

## Example With Remote and Local Steps

The following SELECT statement gets data about pizza restaurants from the `search.web` table:

```
select title, abstract, url
from search.web(500)
where (query='pizza') and
      ((title like 'Round%') or (abstract matches '.*about.*'))
```



```
limit    5
      | sort(field='title')
```

The steps that follow show the order in which YQL processes the remote and local parts of the SELECT statement:

1. YQL calls Yahoo! Search BOSS, the Web service behind the `search.web` table, at the following URL:

```
h t t p : / / b o s s . y a h o o a p -
is.com/ysearch/web/v1/pizza?format=xml&start=0&count=50
```

By calling this URL, YQL gets the first 50 items that match the SELECT statement's remote filter: `query='pizza'`. The `query` element is an input key for the `search.web` table. Although the remote filter in the SELECT is set to 500, to improve efficiency, YQL only fetches 50 items each time it calls BOSS.

2. On the 50 items it retrieved from BOSS, YQL applies the local filter: `((title like 'Round%' ) or (abstract matches '.*about.*'))`. This filter selects an item if the `title` field (column) starts with `Round` or the `abstract` field contains `about`. In this example, from the set of 50 items, YQL finds 3 items that match the local filter.
3. YQL checks that the items retrieved from BOSS contain the `title`, `abstract`, and `url` fields, which are specified after the SELECT keyword.
4. YQL calls BOSS again, incrementing the `start` parameter to 50, to get the next 50 rows:

```
h t t p : / / b o s s . y a h o o a p -
is.com/ysearch/web/v1/pizza?format=xml&start=50&count=50
```

On this second set of 50 items, YQL applies the local filter, but finds no matches.

5. YQL calls BOSS a third time to get the next 50 items:

```
h t t p : / / b o s s . y a h o o a p -
is.com/ysearch/web/v1/pizza?format=xml&start=100&count=50
```

6. On the third set of items from BOSS, YQL applies the local filter and verifies that the `title`, `abstract`, and `url` fields exist. This time, YQL finds 2 more matches, which brings the total number of matches to 5. Because the local limit of 5 has been reached, YQL does not call BOSS again.
7. YQL pipes the 5 items to the `sort` function, ordering the data by the `title` field.
8. YQL returns 5 rows (containing just the `title`, `abstract`, and `url` fields) to the calling Web application.

## Summary of Remote and Local Controls

The following table identifies whether an element in the SELECT statement is processed locally or remotely by YQL.

Syntax Element in SELECT	Local or Remote	Section With Details
Columns or asterisk after the SELECT keyword.	Local	<a href="#">Specifying the Elements Returned (Projection) [8]</a>

Syntax Element in SELECT	Local or Remote	Section With Details
Remote limit and offset, indicated by integers in parentheses after the table name.	Remote	<a href="#">Remote Limits [12]</a>
Remote filter expression in the WHERE clause. The allowed operators are the equal sign or IN. The value compared is an input key for the back-end datasource.	Remote	<a href="#">Remote Filters [9]</a>
Local filter expression in the WHERE clause. Various operators are allowed, including LIKE and MATCH. The value compared is a field (column) in the data returned by the query.	Local	<a href="#">Local Filters [10]</a>
LIMIT and OFFSET keywords after the WHERE clause.	Local	<a href="#">Local Limits [12]</a>
Sort and other functions after the pipe ( ) symbol.	Local	<a href="#">Sort and Other Functions [13]</a>

## GUIDs, Social, and Me

YQL includes a set of pre-defined tables that call the Yahoo! Social APIs. The `social.profile` table, for example, contains information about a Yahoo! user, and the `social.connections` table is a list of the user's friends. The Global User Identifier (GUID) is a string that uniquely identifies a Yahoo! user. In YQL, the `me` keyword is the GUID value of the user currently logged in to Yahoo!. For example, if you are logged in to Yahoo!, and you run the following statement, YQL returns your profile information:

```
select * from social.profile where guid=me
```

Because `me` is a keyword, it is not enclosed in quotes. To specify a GUID value, enclose the string in quotes, for example:

```
select * from social.updates where guid='7WQ7JILMQKTSTTURDDAF3NT35A'
```

## Variable Substitution in the GET Query String

If the URL contains `@var` literals, YQL replaces the literals with the values of query parameters with the same names. For example, suppose that the URL for the call to the YQL Web Service has the `animal` query parameter:

```
http://query.yahooapis.com/v1/yql?animal=dog&q=select * from sometable where animal=@animal
```

For this URL, YQL will run the following SELECT statement:

```
select * from sometable where animal="dog"
```

## Extracting HTML

A key feature of YQL is the ability to access data from structured data feeds such as RSS and ATOM. However, if no such feed is available, you can use the `html` table to get HTML from a page. The `html` table can parse HTML 4.01 or HTML5 and be used with XPath to extract portions of the HTML page.

## Parsing HTML 4.01 Versus HTML5

The `html` table by default parses any HTML page using HTML 4.01 specification, but it can be configured to parse using the HTML5 specification. To parse using HTML5 specification, you assign the `compat` key the value `"html5"` as seen in the first example below. When using the HTML5 specification to parse, the returned response may be slightly different.

For example, to get HTML from Yahoo! Finance that is parsed according to the HTML5 specification, you would use the following YQL statement containing `compat="html5"`:

```
select * from html where url="http://finance.yahoo.com/q?s=yhoo" and
compat="html5"
```

[Run this example in the YQL Console<sup>2</sup>](#)

The same statement without `compat="html5"` returns HTML parsed according to the HTML 4.01 specification:

```
select * from html where url="http://finance.yahoo.com/q?s=yhoo"
```

## Using XPath

The `html` table without XPath returns all of the page's HTML, which may not be useful in an application. By adding an XPath expression to the statement, you can retrieve specific portions of the HTML page.

The XPath expression in the following statement traverses through the nodes in the HTML page to isolate the latest headlines. In this case, the XPath expression looks first for a `div` tag with the ID `yfi_headlines`. Next, the expression gets the second `div` tag and looks for an anchor tag (`a`) within a list item (`li`) of an unordered list (`ul`).

```
select * from html where url="http://finance.yahoo.com/q?s=yhoo" and
xpath='//div[@id="yfi_headlines"]/div[2]/ul/li/a'
```

[Run this example in the YQL console<sup>3</sup>](#)

The following statement also gets information about Yahoo! Inc. stock, but traverses the nodes to get key statistics:

```
select * from html where url="http://finance.yahoo.com/q?s=yhoo" and
xpath='//div[@id="yfi_key_stats"]/div[2]/table'
```

## Selecting Content

Instead of the wildcard asterisk (\*), you can specify a particular element to process. To get just the content from an HTML page, you can specify content keyword after the word `select`. A statement with the content keyword processes the HTML in the following order:

1. It looks for any element named "content" within the elements found.

---

<sup>2</sup> [http://developer.yahoo.com/yql/console/?q=select%20\\*%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22#h=select%20\\*%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20compat%3D%22html5%22](http://developer.yahoo.com/yql/console/?q=select%20*%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22#h=select%20*%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20compat%3D%22html5%22)

<sup>3</sup> [http://developer.yahoo.com/yql/console/?q=select%20\\*%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20%20xpath%3D%27%2Fdiv%5B40id%3D%22yfi\\_headlines%22%5D%2Fdiv%5B2%5D%2Fli%2Fa%27%20A](http://developer.yahoo.com/yql/console/?q=select%20*%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20%20xpath%3D%27%2Fdiv%5B40id%3D%22yfi_headlines%22%5D%2Fdiv%5B2%5D%2Fli%2Fa%27%20A)

2. If an element named "content" is not found, the statement looks for an attribute named "content".
3. If neither an element nor attribute named "content" is found, the statement returns the element's `textContent`.

For example, the following statement extracts only the HTML links (`href` tags) within the headlines on Yahoo! Finance:

```
select href from html where url="http://finance.yahoo.com/q?s=yhoo" and  
xpath='//div[@id="yfi_headlines"]/div[2]/ul/li/a'
```

[Run this example in the YQL console<sup>4</sup>](#)

The following statement, for example, returns the `textContent` of each anchor (`a`) tag retrieved by the XPath expression:

```
select content from html where url="http://finance.yahoo.com/q?s=yhoo"  
and xpath='//div[@id="yfi_headlines"]/div[2]/ul/li/a'
```

[Run this example in the YQL console<sup>5</sup>](#)

---

<sup>4</sup> [http://developer.yahoo.com/yql/console/?q=select%20href%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20xpath%3D%27%2F%2Fdiv%5B%40id%3D%22yfi\\_headlines%22%5D%2Fdiv%5B2%5D%2Ful%2Fli%2Fa%27](http://developer.yahoo.com/yql/console/?q=select%20href%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20xpath%3D%27%2F%2Fdiv%5B%40id%3D%22yfi_headlines%22%5D%2Fdiv%5B2%5D%2Ful%2Fli%2Fa%27)

<sup>5</sup> [http://developer.yahoo.com/yql/console/?q=select%20content%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20xpath%3D%27%2F%2Fdiv%5B%40id%3D%22yfi\\_headlines%22%5D%2Fdiv%5B2%5D%2Ful%2Fli%2Fa%27](http://developer.yahoo.com/yql/console/?q=select%20content%20from%20html%20where%20url%3D%22http%3A%2F%2Ffinance.yahoo.com%2Fq%3Fs%3Dyhoo%22%20and%20xpath%3D%27%2F%2Fdiv%5B%40id%3D%22yfi_headlines%22%5D%2Fdiv%5B2%5D%2Ful%2Fli%2Fa%27)

---

# Chapter 3. Using the Query Builder

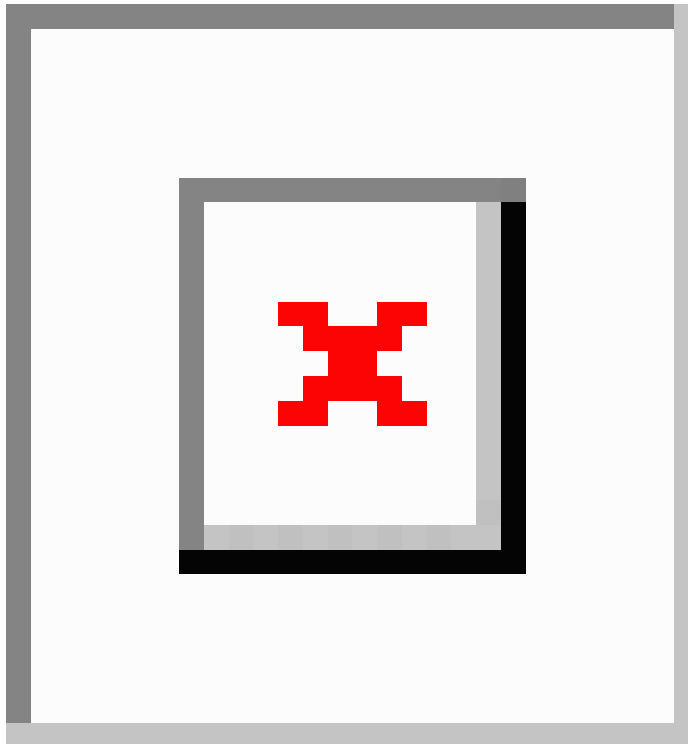
## In this Chapter

- [“YQL Query Builder Overview” \[19\]](#)
- [“Query Builder Usage” \[20\]](#)

## YQL Query Builder Overview

To aid in building YQL statements (queries), the YQL console includes a simple query builder tool. This tool allows you to customize and explore YQL statements.

**Figure 3.1. YQL Query Builder**



### Note

This simple query builder is intended to explore and test simple statements. It does not currently support more advanced statements that include sub-queries, multiple Open Data Tables, or projection.

Features of YQL query builder include:

- **Tabs for each table within a statement:** Each Open Data Table used within a given statement is shown as as a separate tab.
- **Limit and tail functions:** Easily add functions to get the first or last number of results.

- **Table metadata:** Click "**show table info**" to see metadata within each Open Data Table.

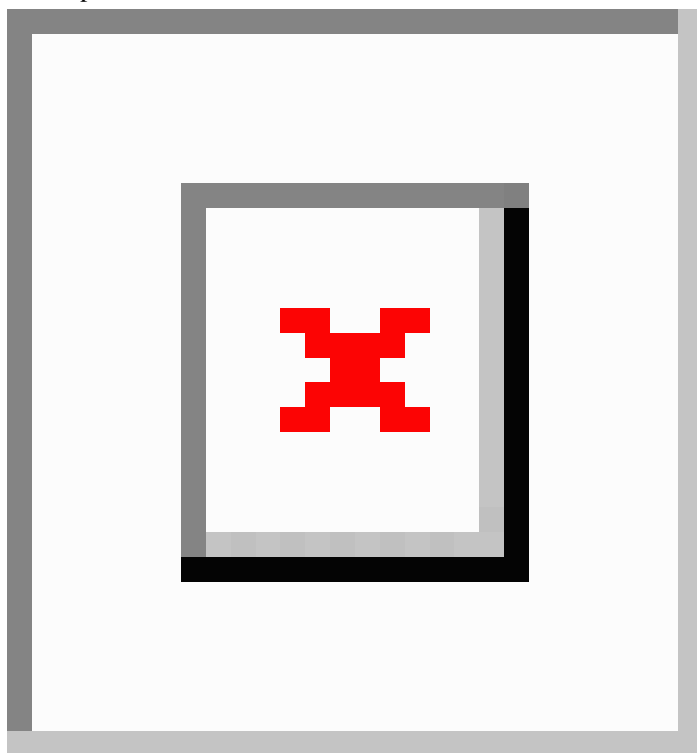
## Query Builder Usage

This section describes how to use the various aspects of the query builder.

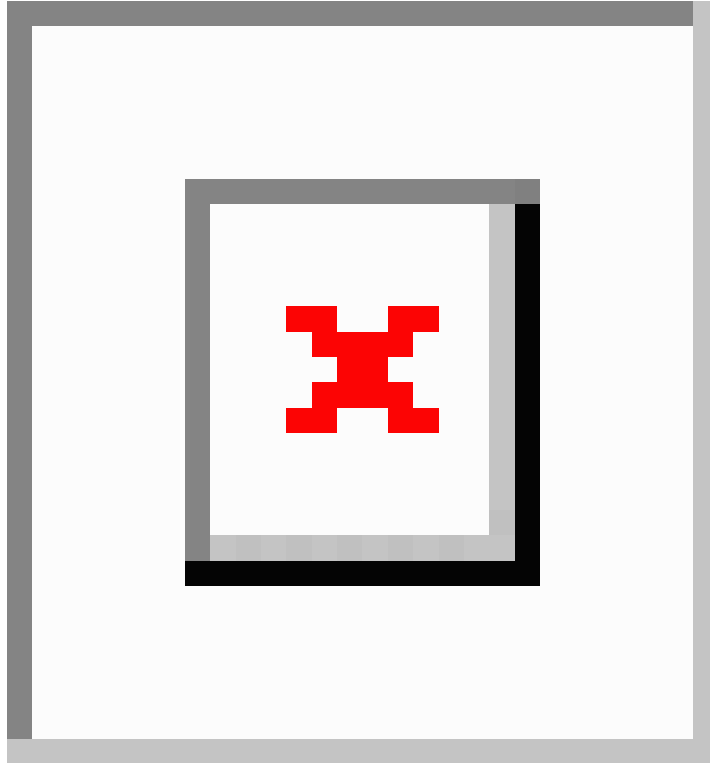
### Loading a Query

1. To run the query builder, first test a YQL statement in the console.

Below the statement box and above the results, a tab appears for each Open Data Table that is required to complete the statement.



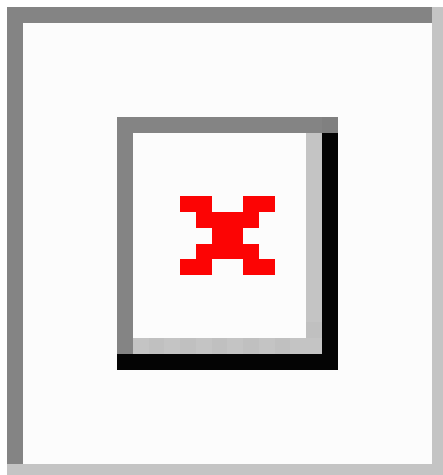
2. Click on a tab associated with an Open Data Table to load the query builder for that table.



## Entering Values

After you load a query, you are presented with a form that contains fields for each required binding.

1. Type values for each required field.
2. Below the required bindings, you can also click **Add Key** to add optional bindings to the statement. Click on the X icon next to a binding to remove it.



3. Click **TEST** within the query builder to run the updated YQL statement.

---

# Chapter 4. INSERT, UPDATE, and DELETE (I/U/D) Statements

In this Chapter:

- [“Introduction to INSERT, UPDATE, and DELETE \(I/U/D\) Statements” \[22\]](#)
- [“Bindings Required for I/U/D” \[22\]](#)
- [“JavaScript Methods Available to I/U/D” \[24\]](#)
- [“Syntax of I/U/D” \[24\]](#)
- [“Limitations of I/U/D” \[25\]](#)
- [“Open Data Table Examples of I/U/D” \[26\]](#)

## Introduction to INSERT, UPDATE, and DELETE (I/U/D) Statements

While YQL SELECT statements allow you to read structured data from almost any source on the Web, their expressed purpose is only to read data.

To perform data manipulation, YQL provides three other SQL-like keywords for writing, updating, and deleting data mapped using a YQL Open Data Table, namely INSERT, UPDATE, and DELETE (I/U/D).

The INSERT statement inserts or adds new data to YQL tables, while the UPDATE statement updates or modifies existing data. DELETE, as the name implies, removes data.

I/U/D statements require the proper [binding inputs \[6\]](#), such as `key`, `value`, or `map`. The actual addition, modification, or deletion of data is performed within the [Open Data Table \[43\]](#).



### Caution

Most sources that provide write capability need authentication. Examples of authentication include username/password combinations or secret API tokens.

If your table requires input that is deemed "private", such as any passwords, authentication keys, or other "secrets", you **MUST** ensure the `https` attribute within the `tables` element is set to `true`.

For more information on about securing private data in Open Data Tables, refer to [Ensuring the Security of Private Information \[20\]](#).

## Bindings Required for I/U/D

I/U/D statements rely entirely on appropriate bindings within an Open Data Table to be usable. Specifically, you must use an [insert, update, or delete bindings element \[3\]](#). These elements help to determine what happens with the information you pass in through a YQL statement.



Consider the following INSERT statement for shortening URLs using bit.ly:

```
INSERT INTO bitly.shorten (login, apiKey, longUrl) VALUES ('USERNAME',  
'API_KEY', 'http://yahoo.com')
```

The corresponding Open Data Table for this statement follows:

```
<?xml version="1.0" encoding="UTF-8"?>  
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"  
https="true">  
  <meta>  
    <author>Nagesh Susarla</author>  
  
    <documentationURL>http://code.google.com/p/bitly-api/wiki/ApiDocumentation</documentationURL>  
  
  </meta>  
  <bindings>  
    <insert itemPath="" produces="XML">  
      <urls>  
  
        <url>http://api.bit.ly/shorten?version=2.0.1&format=xml&{-join|&|longUrl}</url>  
  
      </urls>  
      <inputs>  
        <key id="login" type="xs:string" paramType="query"  
required="true"/>  
        <key id="apiKey" type="xs:string" paramType="query"  
required="true"/>  
        <value id="longUrl" type="xs:string" paramType="path"  
required="true"/>  
      </inputs>  
    </insert>  
  </bindings>  
</table>
```

[Run this example on the YQL console](#)<sup>1</sup>



## Note

To run the example, you must replace USERNAME and API\_KEY in the YQL statement with your actual bit.ly username and API key (available through your [bit.ly](#)<sup>2</sup> account page).

The above Open Data Table shows one of the most basic ways to use an INSERT statement because it does not require JavaScript to massage the data. Consequently, it requires no [execute \[19\]](#) element. It simply uses the login and apiKey as keys to authenticate the user, with the longUrl as the new value passed to the bit.ly API. For more information on how the key, value, and map values are used, refer to [key/value/map elements in Using Open Data Tables \[6\]](#).

---

<sup>1</sup>[https://developer.yahoo.com/yql/console/?q=USE%20%22http%3A%2F%2Fwww.yqlblog.net%2Fsamples%2Fbitly.shorten.xml%22%3B%20INSERT%20INTO%20bitly.shorten%20%28login%2C%20apiKey%2C%20longUrl%29%20VALUES%20%28%27YOUR\\_LOGIN%27%2C%20%27YOUR\\_API\\_KEY%27%2C%20%27http%3A%2F%2Fyahoo.com%27%29](https://developer.yahoo.com/yql/console/?q=USE%20%22http%3A%2F%2Fwww.yqlblog.net%2Fsamples%2Fbitly.shorten.xml%22%3B%20INSERT%20INTO%20bitly.shorten%20%28login%2C%20apiKey%2C%20longUrl%29%20VALUES%20%28%27YOUR_LOGIN%27%2C%20%27YOUR_API_KEY%27%2C%20%27http%3A%2F%2Fyahoo.com%27%29)

<sup>2</sup> <http://bit.ly/>



## Note

In the above example, the url element contains a URI template to aid in the construction of the URI:

```
{-join|&|longUrl}
```

Here the join operator creates a key-value pair using the variable longURL along with its value. This pair is preceded by the ampersand symbol (&).

# JavaScript Methods Available to I/U/D

For Web services that require specific authentication methods or specific types of HTTP requests, YQL provides several JavaScript methods for use within the [execute \[19\]](#) element:

- Methods that allow HTTP PUT, POST, and DELETE requests, in addition to GET.
- The ability to specify the content type on data being sent, using `contentType`.
- The ability to automatically convert the data being returned using `accept`.

For more information on the JavaScript methods available for use within I/U/D statements, refer to the [JavaScript Objects, Methods, and Variables Reference \[20\]](#).

## Syntax of I/U/D

This section discusses the syntax for I/U/D statements.

## Syntax of INSERT

The YQL INSERT statement has the following syntax:

```
INSERT INTO (table) (list of comma separated field names) VALUES (list of comma separated values)
```

The `INSERT INTO` keywords marks the start of an INSERT statement.

The `table` is either the YQL pre-defined or Open Data Table that represents a data source.

Following the table name is a list of field names indicating the table columns where YQL inserts a new row of data.

The `VALUES` clause indicates the data inserted into those columns. String values are enclosed in quotes.

In YQL, statement keywords such as `SELECT` and `WHERE` are case-insensitive. Table and field names are case sensitive. In string comparisons, the values are case sensitive. String literals must be enclosed in quotes; either double or single quotes are allowed.

## Syntax of UPDATE

The YQL UPDATE statement has the following syntax:

```
UPDATE (table) SET field=value WHERE filter
```

The UPDATE keyword marks the start of an UPDATE statement. This is followed by the table name.

The table is either the YQL pre-defined or Open Data Table that represents a data source.

The SET clause is the part of the statement in which we pass new data to the update binding in the Open Data Table.

The WHERE clause indicates which data should be updated. Only remote filters can be present in the WHERE clause of an UPDATE statement.

The following example shows how the UPDATE statement syntax can look for updates to your status on Yahoo! Profiles:

```
UPDATE social.profile.status SET status="Using YQL UPDATE" WHERE guid=me
```

[Try this example in the YQL console<sup>3</sup>](#)

In the above example, status and guid are all bindings within the inputs element, which is nested within an update element. The status is a value element, since this is data that is updating a value using the Open Data Table. The guid binding is simply a key element, as it is a required "key" that determines ownership of this status.

## Syntax of DELETE

The YQL DELETE statement has the following syntax:

```
DELETE FROM [table] WHERE filter
```

The DELETE keyword marks the start of a DELETE statement.

The table is either the YQL pre-defined or Open Data Table that represents a data source.

This is immediately followed by a remote filter that determines what table rows to remove.

The following example deletes a particular Yahoo! update:

```
DELETE FROM social.updates WHERE guid="me" and suid="12345" and  
source="agg.bebo"
```

In the example above, the remote filters are the SUID of the update followed by the guid of the Yahoo! user and the source of the update.

## Limitations of I/U/D

While I/U/D statements support most of the same functionality as the SELECT statement, there are a few caveats to keep in mind:

- **Local filtering:** I/U/D statements do not support [local filtering](#) [10].
- **Sub-selects:** INSERT statements do not support [sub-selects](#) [11].

---

<sup>3</sup> <http://developer.yahoo.com/yql/console/?q=UPDATE%20social.profile.status%20SET%20status%3D%22Using%20YQL%20UPDATE%22%20WHERE%20guid%3Dme>

- **Paging:** I/U/D statements do not support [paging \[12\]](#). However, you can use sub-selects within UPDATE and DELETE statements to narrow down the values you wish to insert or delete, as shown in the following example:

```
DELETE FROM table WHERE guid IN (SELECT guid FROM social.connections  
WHERE owner_guid = me)
```

## Open Data Table Examples of I/U/D

The following Open Data Tables provide examples of INSERT:

- [Bit.ly Shorten URL \[26\]](#)
- [WordPress Post \(Insert\) \[27\]](#)



### Tip

To better understand the examples presented in this section, refer first to [Using YQL Open Data Tables \[43\]](#) and [Executing JavaScript in Open Data Tables \[19\]](#).

## Bit.ly Shorten URL (INSERT)

The following Open Data Table allows you to shorten a long URL into a bit.ly link.

This table showcases the following:

- using INSERT statements
- using value bindings

### Example Statement:

```
INSERT INTO bitly.shorten (login, apiKey, longUrl) VALUES ('USERNAME',  
'API_KEY', 'http://yahoo.com')
```

### Open Data Table Source:

```
<?xml version="1.0" encoding="UTF-8"?>  
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"  
https="true">  
  <meta>  
    <author>Nagesh Susarla</author>  
  
<documentationURL>http://code.google.com/p/bitly-api/wiki/ApiDocumentation</documentationURL>  
  
  </meta>  
  <bindings>  
    <insert itemPath="" produces="XML">  
      <urls>  
  
<url>http://api.bit.ly/shorten?version=2.0.1&format=xml&{-join|&|longUrl}</url>  
  
      </urls>  
    </inputs>
```

```
<key id="login" type="xs:string" paramType="query"
required="true"/>
<key id="apiKey" type="xs:string" paramType="query"
required="true"/>
<key id="longUrl" type="xs:string" paramType="path"
required="true"/>
</inputs>
</insert>
</bindings>
</table>
```

[Run this example on the YQL console](#)<sup>4</sup>



### Note

To run the example, you must replace USERNAME and API\_KEY in the YQL statement with your actual bit.ly username and API key (available through your [bit.ly](#)<sup>5</sup> account page).

## WordPress Post (INSERT)

The following Open Data Table allows you to post to your WordPress blog.

This table showcases the following:

- calling a YQL query within `execute`
- allowing INSERT in addition to SELECT statements
- performing HTTP POST with JavaScript methods
- creating an XML response with [E4X](#)<sup>6</sup>

### Example Statement:

```
USE "http://www.datatables.org/wordpress/wordpress.post.xml"; INSERT
into wordpress.post (title, description, blogurl, username, password)
values ("Test Title", "This is a test body", "http://your_word-
press_blog_site.com", "your_wordpress_username", "your_wordpress_pass-
word")
```

### Open Data Table Source:

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>insert into {table} (title, description, blogurl,
username, password) values ("Test Title", "This is a test body",
"http://yqltest.wordpress.com", "yqltest", "password")</sampleQuery>
  </meta>
  <bindings>
```

---

<sup>4</sup>[https://developer.yahoo.com/yql/console/?q=USE%20%22http%3A%2F%2Fwww.yqlblog.net%2Fsamples%2Fbitly.shorten.xml%22%3B%20INSERT%20INTO%20bitly.shorten%20%28login%2C%20apiKey%2C%20longUrl%29%20VALUES%20%28%27YOUR\\_LO-GIN%27%2C%20%27YOUR\\_API\\_KEY%27%2C%20%27http%3A%2F%2Fyahoo.com%27%29](https://developer.yahoo.com/yql/console/?q=USE%20%22http%3A%2F%2Fwww.yqlblog.net%2Fsamples%2Fbitly.shorten.xml%22%3B%20INSERT%20INTO%20bitly.shorten%20%28login%2C%20apiKey%2C%20longUrl%29%20VALUES%20%28%27YOUR_LO-GIN%27%2C%20%27YOUR_API_KEY%27%2C%20%27http%3A%2F%2Fyahoo.com%27%29)

<sup>5</sup><http://bit.ly/>

<sup>6</sup>[http://en.wikipedia.org/wiki/ECMAScript\\_for\\_XML](http://en.wikipedia.org/wiki/ECMAScript_for_XML)

```
<insert itemPath="" produces="XML">
  <urls>
    <url></url>
  </urls>
  <inputs>
    <key id='username' type='xs:string' paramType='variable'
required="true" />
    <key id='password' type='xs:string' paramType='variable'
required="true" />
    <key id='blogurl' type='xs:string' paramType='variable'
required="true" />
    <value id='allow_comments' type='xs:string' paramType='variable'
required="false" default="1" />
    <value id='tags' type='xs:string' paramType='variable'
required="false"/>
    <value id='keywords' type='xs:string' paramType='variable'
required="false"/>
    <value id='convert_breaks' type='xs:string' paramType='variable'
required="false" default="0" />
    <value id='title' type='xs:string' paramType='variable'
required="true" />
    <value id='excerpt' type='xs:string' paramType='variable'
required="false"/>
    <value id='description' type='xs:string' paramType='variable'
required="true"/>
    <value id='moretext' type='xs:string' paramType='variable'
required="false"/>
    <value id='allow_pings' type='xs:string' paramType='variable'
required="false" default="0" />
    <value id='publish' type='xs:string' paramType='variable'
required="false" default="1" />
  </inputs>
  <execute><![CDATA[
    if (allow_comments == "true") {
      allow_comments = 1;
    } else if (allow_comments == "false") {
      allow_comments = 0;
    }
    if (convert_breaks == "true") {
      conver_tbreaks = 1;
    } else if (convert_breaks == "false") {
      convert_breaks = 0;
    }
    if (allow_pings == "true") {
      allow_pings = 1;
    } else if (allow_pings == "false") {
      allow_pings = 0;
    }
    postData =
    <methodCall><methodName>metaWeblog.newPost</methodName>
      <params>
        <param>
          <value>
            <string>1</string>
```

```
        </value>
    </param>
    <param>
        <value>
            <string>{username}</string>
        </value>
    </param>
    <param>
        <value>
            <string>{password}</string>
        </value>
    </param>
    <param>
        <value>
            <struct>
                <member>
                    <name>mt_allow_comments</name>
                    <value>
                        <int>{allow_comments}</int>
                    </value>
                </member>
                <member>
                    <name>mt_convert_breaks</name>
                    <value>
                        <string>{convert_breaks}</string>
                    </value>
                </member>
                <member>
                    <name>title</name>
                    <value>
                        <string>{title}</string>
                    </value>
                </member>
                <member>
                    <name>description</name>
                    <value>
                        <string>{description}</string>
                    </value>
                </member>
                <member>
                    <name>mt_allow_pings</name>
                    <value>
                        <string>{allow_pings}</string>
                    </value>
                </member>
            </struct>
        </value>
    </param>
    <param>
        <value>
            <boolean>{publish}</boolean>
        </value>
    </param>
</params>
```

```
</methodCall>;
if (tags) {
    postData.params.param..struct.member +=
    <member>
        <name>mt_tags</name>
        <value>
            <string>{tags}</string>
        </value>
    </member>;
} else {
    postData.params.param..struct.member +=
    <member>
        <name>mt_tags</name>
        <value>
            <string></string>
        </value>
    </member>;
}
if (moretext) {
    postData.params.param..struct.member +=
    <member>
        <name>mt_text_more</name>
        <value>
            <string>{moretext}</string>
        </value>
    </member>;
} else {
    postData.params.param..struct.member +=
    <member>
        <name>mt_text_more</name>
        <value>
            <string></string>
        </value>
    </member>;
}
if (excerpt) {
    postData.params.param..struct.member +=
    <member>
        <name>mt_excerpt</name>
        <value>
            <string>{excerpt}</string>
        </value>
    </member>;
} else {
    postData.params.param..struct.member +=
    <member>
        <name>mt_excerpt</name>
        <value>
            <string></string>
        </value>
    </member>;
}
if (keywords) {
    postData.params.param..struct.member +=
```



```

        <member>
            <name>mt_keywords</name>
            <value>
                <string>{keywords}</string>
            </value>
        </member>;
    } else {
        postData.params.param..struct.member +=
        <member>
            <name>mt_keywords</name>
            <value>
                <string></string>
            </value>
        </member>;
    }
    url = blogurl + "/xmlrpc.php";
    myRequest = y.rest(url);
    myRequest.contentType("text/xml");
    //myRequest.accept("text/xml");
    myRequest.header("Connection", "close");
    results = myRequest.post('<?xml version="1.0" encoding="UTF-8"?>'
+ postData.toString()).response;
    postId = results.params.param.value.string.toString();
    //response.object = results;
    response.object = <postid>{postId}</postid>;
    ]]>
</execute>
</insert>
<update itemPath="" produces="XML">
    <urls>
        <url></url>
    </urls>
    <inputs>
        <key id='username' type='xs:string' paramType='variable'
required="true" />
        <key id='password' type='xs:string' paramType='variable'
required="true" />
        <key id='blogurl' type='xs:string' paramType='variable'
required="true" />
        <key id='postId' type='xs:string' paramType='variable'
required="true" />
        <value id='allow_comments' type='xs:string' paramType='variable'
required="true" />
        <value id='tags' type='xs:string' paramType='variable'
required="true"/>
        <value id='keywords' type='xs:string' paramType='variable'
required="true"/>
        <value id='convert_breaks' type='xs:string' paramType='variable'
required="true" />
        <value id='title' type='xs:string' paramType='variable'
required="true" />
        <value id='excerpt' type='xs:string' paramType='variable'
required="true"/>
        <value id='description' type='xs:string' paramType='variable'

```

```
required="true"/>
  <value id='moretext' type='xs:string' paramType='variable'
required="true"/>
  <value id='allow_pings' type='xs:string' paramType='variable'
required="true" />
  <value id='publish' type='xs:string' paramType='variable'
required="true"/>
</inputs>
<execute>
  <![CDATA[
    if (allow_comments == "true") {
      allow_comments = 1;
    } else if (allow_comments == "false") {
      allow_comments = 0;
    }
    if (convert_breaks == "true") {
      conver_tbreaks = 1;
    } else if (convert_breaks == "false") {
      convert_breaks = 0;
    }
    if (allow_pings == "true") {
      allow_pings = 1;
    } else if (allow_pings == "false") {
      allow_pings = 0;
    }
    postData = <methodCall>
    <methodName>metaWeblog.newPost</methodName>
    <params>
      <param>
        <value>
          <string>1</string>
        </value>
      </param>
      <param>
        <value>
          <string>{username}</string>
        </value>
      </param>
      <param>
        <value>
          <string>{password}</string>
        </value>
      </param>
      <param>
        <value>
          <struct>
            <member>
              <name>mt_allow_comments</name>
            </member>
            <member>
              <name>mt_convert_breaks</name>
            </member>
          </struct>
        </value>
      </param>
    </params>
  </methodCall>
  </execute>
</script>
```

```
<value>
    <string>{convert_breaks}</string>
</value>
</member>
<member>
    <name>title</name>
    <value>
        <string>{title}</string>
    </value>
</member>
<member>
    <name>description</name>
    <value>
        <string>{description}</string>
    </value>
</member>
<member>
    <name>mt_allow_pings</name>
    <value>
        <string>{allow_pings}</string>
</value>
</member>
</struct>
</value>
</param>
<param>
    <value>
        <boolean>{publish}</boolean>
    </value>
</param>
</params>
</methodCall>;
if (tags) {
    postData.params.param..struct.member +=
    <member>
        <name>mt_tags</name>
        <value>
            <string>{tags}</string>
        </value>
    </member>;
} else {
    postData.params.param..struct.member +=
    <member>
        <name>mt_tags</name>
        <value>
            <string></string>
        </value></member>;
}
if (moretext) {
    postData.params.param..struct.member +=
    <member>
        <name>mt_text_more</name>
        <value>
            <string>{moretext}</string>
```

```

        </value>
    </member>;
} else {
    postData.params.param..struct.member +=
    <member>
        <name>mt_text_more</name>
        <value>
            <string></string>
        </value>
    </member>;
}
if (excerpt) {
    postData.params.param..struct.member +=
    <member>
        <name>mt_excerpt</name>
        <value>
            <string>{excerpt}</string>
        </value>
    </member>;
} else {
    postData.params.param..struct.member +=
<member>
    <name>mt_excerpt</name>
    <value>
        <string></string>
    </value>
    </member>;
}
if (keywords) {
    postData.params.param..struct.member +=
    <member>
        <name>mt_keywords</name>
        <value>
            <string>{keywords}</string>
        </value>
    </member>;
} else {
    postData.params.param..struct.member +=
    <member>
        <name>mt_keywords</name>
        <value>
            <string></string>
        </value>
    </member>;
}
]]>
</execute>
</update>
<select itemPath="" produces="XML">
    <urls>
        <url></url>
    </urls>
    <inputs>
        <key id='postid' type='xs:string' paramType='variable'

```

```
required="true" />
  <key id='blogurl' type='xs:string' paramType='variable'
required="true" />
  <key id='username' type='xs:string' paramType='variable'
required="true" />
  <key id='password' type='xs:string' paramType='variable'
required="true" />
</inputs>
<execute><![CDATA[
  postData = <methodCall>
    <methodName>metaWeblog.getPost</methodName>
<params>
  <param>
    <value>
      <string>{postId}</string>
    </value>
  </param>
  <param>
    <value>
      <string>{username}</string>
    </value>
  </param>
  <param>
    <value>
      <string>{password}</string>
    </value>
  </param>
</params>
</methodCall>;
url = blogurl + "/xmlrpc.php";
myRequest = y.rest(url);
myRequest.contentType("text/xml");
//myRequest.accept("text/xml");
myRequest.header("Connection", "close");
results = myRequest.post('<?xml version="1.0" encoding="UTF-8"?>'
+ postData.toString()).response;
response.object = results;
]]></execute>
</select>
</bindings>
</table>
```

[Run this example<sup>7</sup>](#) in the YQL Console. Be sure to use your blog's URL, username, and password as values for the blogurl, username, and password keys.

---

<sup>7</sup> [https://developer.yahoo.com/yql/console/?q=select%20\\*%20from%20twitter.account.credentials;&env=store://datatables.org/all-tablesWithkeys#h=USE%20%22http%3A//www.datatables.org/wordpress/wordpress.post.xml%22%3B%20INSERT%20into%20wordpress.post%20%28title%2C%20description%2C%20blogurl%2C%20username%2C%20password%29%20values%20%28%22Test%20Title%22%2C%20%22This%20is%20a%20test%20body%22%2C%20%22http%3A//your\\_wordpress\\_blog\\_site.com%22%2C%20%22your\\_wordpress\\_username%22%2C%20%22your\\_wordpress\\_password%22%29](https://developer.yahoo.com/yql/console/?q=select%20*%20from%20twitter.account.credentials;&env=store://datatables.org/all-tablesWithkeys#h=USE%20%22http%3A//www.datatables.org/wordpress/wordpress.post.xml%22%3B%20INSERT%20into%20wordpress.post%20%28title%2C%20description%2C%20blogurl%2C%20username%2C%20password%29%20values%20%28%22Test%20Title%22%2C%20%22This%20is%20a%20test%20body%22%2C%20%22http%3A//your_wordpress_blog_site.com%22%2C%20%22your_wordpress_username%22%2C%20%22your_wordpress_password%22%29)

---

# Chapter 5. Response Data

## In this Chapter:

- [“Supported Response Formats” \[36\]](#)
- [“JSONP-X: JSON envelope with XML content” \[36\]](#)
- [“Structure of Response” \[38\]](#)
- [“Information about the YQL Call” \[38\]](#)
- [“XML-to-JSON Transformation” \[39\]](#)
- [“Errors and HTTP Response Codes” \[41\]](#)

## Supported Response Formats

The YQL Web Service can return data in either XML, JSON, or JSONP format. The default format is XML. To specify JSON, include the `format=json` parameter in the URL of the YQL Web service, for example:

```
http://query.yahooapis.com/v1/public/yql?q=select * from social.connections where owner_guid=me&format=json
```

To specify JSONP, include both the `format` and `callback` query parameters. The `callback` parameter indicates the name of the JavaScript callback function. Here's an example:

```
http://query.yahooapis.com/v1/public/yql?q=select * from social.connections where owner_guid=me&format=json&callback=cbfunc
```

The format of the response data is not dependent on the format of the original datasource. For example, if a YQL table is backed by a datasource in XML, the YQL Web Service can return data in JSON. For more information, see [XML-to-JSON Transformation \[39\]](#).

## JSONP-X: JSON envelope with XML content

Aside from offering JSON as a response format with callbacks, you can also specify XML as the response format. If in your query you specify a callback (`callback=cbfunction`) and also request the format be in XML (`format=xml`), then YQL returns a string representation of the XML within an array. Compare the following Yahoo! Local search for Indian restaurants in Sunnyvale, California using JSONP and JSONP-X callbacks, respectively:

### JSONP Callback

```
function({"query":{"count":"10","created":"2009-07-10T09:13:28Z","lang":"en-US","updated":"2009-07-10T09:13:28Z",  
"uri":"http://query.yahooapis.com/v1/yql?q=select+title+from+local.search+where+zip%3D%2794085%27+and+query%3D%27indian+restaurants%27",  
"diagnostics":{"publiclyCallable":"true",  
"url":{"execution-time":"332"},
```

```
"content": "http://local.yahooapis.com/LocalSearchService/V3/LocalSearch?zip=94085&query=indian%20restaurants&start=1&results=10",
"user-time": "335", "service-time": "332", "build-version": "2213"}, "results": { "Result": [
  { "Title": "Grand Indian Buffet" },
  { "Title": "Turmeric Restaurant" },
  { "Title": "Taj India" },
  { "Title": "Shalimar" },
  { "Title": "Komala Vilas" },
  { "Title": "Brindavan Fine Indian Cuisine" },
  { "Title": "Panchavati Indian Veggie Foods" },
  { "Title": "Sneha Restaurant" },
  { "Title": "Bhavika's Food to Go" },
  { "Title": "ATHIDHI INDIAN CUISINE" } ] ] } } };
```

### JSONP-X Callback

```
function({ "query": { "content": "10", "created": "2009-07-10T09:10:29Z", "lang": "en-US", "updated": "2009-07-10T09:10:29Z",
"url": "http://query.yahooapis.com/v1/q?select=Title+from+local.search+where+zip%3D%2F94085%2F+and+query%3D%2Findian+restaurants%2F",
"diagnostics": { "publiclyCallable": "true",
"url": { "execution-time": "558",
"content": "http://local.yahooapis.com/LocalSearchService/V3/LocalSearch?zip=94085&query=indian%20restaurants&start=1&results=10",
"user-time": "561", "service-time": "558", "build-version": "2213" } } }, "results": [
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Grand Indian Buffet</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Turmeric Restaurant</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Taj India</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Shalimar</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Komala Vilas</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Brindavan Fine Indian Cuisine</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Panchavati Indian Veggie Foods</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Sneha Restaurant</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>Bhavika's Food to Go</Title></Result>",
  "<Result xmlns=\"urn:yahoo:lcl\"><Title>ATHIDHI INDIAN CUISINE</Title></Result>"] ] } } };
```

## Structure of Response

Every response from YQL includes a query element, which contains the diagnostics and results elements. (For details on the diagnostics element, see [Information About the YQL Call \[38\]](#).) The repeating elements within result are "rows" from a YQL table. For example, `select * from social.connections` returns multiple connection elements within the result element.

The following listing shows the basic structure of the XML data in the response of a call to the YQL Web Service.

```
<query ... (attributes such as count)>
  <diagnostics>
    ... (sub-elements such as publiclyCallable)
  <results>
    ... (data returned by the call to YQL)
  </results>
</query>
```

The next listing shows the basic structure of YQL response in JSON format:

```
{
  "query": {
    "count": ...
    ...
    "diagnostics": {
      "publiclyCallable": ...,
      ...
    },
    "results": {
      // data returned by call to YQL
      ...
    }
  }
}
```

## Information about the YQL Call

To get information about the execution of the YQL call, check the attributes of the query element and the sub-elements of the diagnostics element.

The following table lists the attributes of the query element in an XML response. In a JSON response, these attributes are mapped to the name-value pairs contained in the query object.

Attribute of query Element	Description
count	The number of items (rows) in returned by the YQL statement. In an XML response, count is the number of sub-elements in the results element.
created	The date and time the response was created.



Attribute of query Element	Description
lang	The locale for the response.
updated	The date and time this response was last updated.

The `diagnostics` element contains information about the calls the YQL Web service made to back-end datasources. The following table lists the XML sub-elements of the `diagnostics` element. In a JSON response, these sub-elements are mapped to name-value pairs contained in the `diagnostics` object.

Sub-element of diagnostics Element	Description
publiclyCallable	True if the table is public data, false for private data. Authorization is required for private data.
url	The URL of the Web service called by YQL to get the data. The value of the <code>execution-time</code> attribute is elapsed time, in milliseconds required to call the URL.
user-time	The time YQL would take if each request were sequentially performed. YQL performs most request-related actions in parallel, so this measurement is informational only.
service-time	The time YQL takes to perform the request and return the result, including time taken to execute Javascript. This is the time that you wait for each YQL statement response.

## XML-to-JSON Transformation

If the YQL results are in JSON format, and the table is backed by an XML data source, then YQL transforms the data from XML to JSON. This transformation is "lossy," that is, you cannot transform the JSON back to XML. YQL transforms XML data to JSON according to the following rules:

- Attributes are mapped to name:value pairs.
- Element CDATA or text sections are mapped to "content":value pairs if the element contains attributes or sub-elements. Otherwise they are mapped to the element name's value directly.
- Namespace prefixes are removed from names.
- If the attribute, element, or namespace-less element would result in the same key name in the JSON structure, an array is created instead.

For example, consider the following XML:

```
<doc yahoo:count=10>
  <ns:a>avalue</ns:a>
  <b><subb>bvalue</subb></b>
  <c count=20 yahoo:count=30>
    <count>40</count>
    <count><subcount>10</subcount></count>
  </c>
  <d att="cat">dog</d>
</doc>
```

This XML is transformed to the following JSON structure:

```
{doc: {
  count:10,
  a:"avalue",
  b: { subb: "bvalue"},
  c: { count: [ 20,30,40,{subcount:10} ] },
  d: { att:"cat", content:"dog" }
}}
```

## JSON-to-JSON Transformation

YQL transforms all JSON data sources into XML before returning results. To return JSON results that were obtained from a JSON data source, YQL must first transform the original JSON data to XML and then transform the XML back into a JSON result. During the transformation from XML to JSON, the original JSON may be altered or become "lossy". In other words, the original JSON may not be the same as the returned JSON.

The original JSON may be altered in the following ways:

- JSON numbers are returned as strings.
- JSON arrays containing a single element are returned as a JSON object.

To prevent this "lossy" transformation, you append the query string parameter `jsonCompat=new` [3] to the YQL Web Service URL that you are using. For those creating tables, you use the `jsonCompat("new")` [35] when making REST calls to other Web services. To illustrate how the `jsonCompat` parameter is used, we'll look at the below examples that use the community table that queries the Gowalla API, which only returns JSON.

- **Lossy JSON**

The following REST URI uses the public YQL Web Service URL and the Gowalla table. The `jsonCompat` parameter is not added to the URI, so the original JSON returned from the Gowalla API will be altered in the YQL response.

```
http://query.yahooapis.com/v1/public/yql?q=select * from gowalla.users
where id='sco' and api_key='fa574894bd-dc43aa96c556eb457b4009'&env=store://datatables.org/alltableswithkeys
```

In the returned "lossy" JSON results below, notice that `last_checkins` is an object and that `_comments_count` property has the the string value "0".

```
...
"last_checkins":
{
  "type": "checkin",
  "url": "/checkins/39776909",
  "spot":
  {
    "image_url":
"http://static.gowalla.com/categories/190-2a44f344d6504e3b4510998e7c10f9bd-100.png",

    "url": "/spots/1440379",
    "name": "Authentic Smiles"
  },

```

```

    "message": "<Mumble mumble>",
    "created_at": "2011-06-29T14:10:43Z",
    "_comments_count": "0"
  },
  ...

```

- **Lossless JSON**

Appending the `jsonCompat=new` query parameter to the REST URI as seen below, YQL now returns the same JSON data as the Gowalla API.

```

http://query.yahooapis.com/v1/public/yql?q=select * from gowalla.users
where id='sco' and api_key='fa574894bd-dc43aa96c556eb457b4009'&env=store://datatables.org/allt-
ableswithkeys&jsonCompat=new

```

In the returned "lossless" JSON results below, the `last_checkins` property is now an array with a single element, and `_comments_count` is a number.

```

...
  "last_checkins":
  [
    {
      "type": "checkin",
      "spot": {
        "image_url":
"http://static.gowalla.com/categories/190-2a44f344d6504e3b4510998e7c10f9bd-100.png",

        "url": "/spots/1440379",
        "name": "Authentic Smiles"
      },
      "message": "<Mumble mumble>",
      "_comments_count": 0,
      "url": "/checkins/39776909",
      "created_at": "2011-06-29T14:10:43Z"
    }
  ],
...

```

## Errors and HTTP Response Codes

The YQL Web Service returns the following HTTP response codes:

Error	Description
200 OK	The YQL statement executed successfully. If the YQL statement is syntactically correct and if authorization succeeds, it returns 200 OK even if the calls to back-end data services return 400 or 500 errors. Information about these back-end errors is in the <code>diagnostics</code> element of the response from YQL.
400 Bad Request	Malformed syntax or bad query. This error occurs if the WHERE clause does not include a required input key. The XML <code>error</code> element includes a text

Error	Description
	description of the error. In the YQL Console, the error description appears in a highlighted bar.
401 Authorization Required	The user running the application calling YQL is not authorized to access private data. This error also occurs if the user attempts to access his or her own private data without logging in to Yahoo!.
999 - Unable to process this request at this time	This error normally occurs when too many requests are sent to YQL and the rate limit has been reached.

---

# Chapter 6. Using YQL Open Data Tables

In this Chapter:

- [“Overview of Open Data Tables” \[43\]](#)
- [“Invoking an Open Data Table Definition within YQL” \[43\]](#)
- [“Setting Key Values for Open Data Tables” \[46\]](#)

## Overview of Open Data Tables

YQL contains an extensive list of built-in tables for you to use that cover a wide range of Yahoo! Web services and access to off-network data. Open Data Tables in YQL allow you to create and use your **own** table definitions, enabling YQL to bind to any data source through the SQL-like syntax and fetch data. Once created anyone can use these definitions in YQL.

An Open Data Table definition is an XML file that contains information as you define it, including, but not limited to the following:

- **Authentication and Security Options:** The kind of authentication you require for requests coming into your service. Also, whether you require incoming connections to YQL be made over a secure socket layer (via HTTPS).
- **Sample Query:** A sample query that developers can run via YQL to get information back from this connection.
- **YQL Data Structure:** Instructions on how YQL should create URLs that access the data available from your Web service. Also, an Open Data Table definition provides YQL with the URL location of your Web service along with the individual query parameters (keys) available to YQL.
- **Pagination Options:** How YQL should "page" through results. If your service can provide staggered results, paging will allow YQL to limit the amount of data returned.

## Invoking an Open Data Table Definition within YQL

If you want to access external data that is not provided through the standard YQL set of tables (accessible through the `show tables` query), YQL provides the `use` statement when you want to import external tables defined through your Open Data Table definition.

### Invoking a Single Open Data Table

You can access a single Open Data Table using the **USE** and **AS** verbs:

```
USE "http://myserver.com/mytables.xml" AS mytable;  
SELECT * FROM mytable WHERE...
```

**Tip**

The AS verb in the above example is optional. If you omit the AS verb, YQL uses the filename (without the .xml file ending) to name the table.

In the above query, USE precedes the location of the Open Data Table definition, which is then followed by AS and the table as defined within your Open Data Table definition. After the semicolon, the query is formed as would be any other YQL query. YQL fetches the URL above and makes it available as a table named `mytable` in the current request scope. The statements following use can then select or describe the particular table using the name `mytable`.

## Invoking Multiple Open Data Tables

You can also specify multiple Open Data Tables by using multiple USE statements in the following manner:

```
USE "http://myserver.com/mytables1.xml" as table1;
USE "http://myserver.com/mytables2.xml" as table2;
SELECT * FROM table1 WHERE id IN (select id FROM table2)
```

## Invoking Multiple Open Data Tables as an Environment

An easier way to use multiple Open Data Tables is to write or use a YQL environment file, which allows you to use multiple tables at once without the USE verb in your YQL statements.

An environment file is simply a text file that contains a list of USE and SET statements, typically ending with a ".env" suffix.

Here is how an environment file can look:

```
USE 'http://www.datatables.org/amazon/amazon.ecs.xml' AS amazon.ecs;
USE 'http://www.datatables.org/bitly/bit.ly.shorten.xml' AS
bit.ly.shorten;
USE 'http://www.datatables.org/delicious/delicious.feeds.popular.xml'
AS delicious.feeds.popular;
USE 'http://www.datatables.org/delicious/delicious.feeds.xml' AS
delicious.feeds;
USE 'http://www.datatables.org/dopplr/dopplr.auth.xml' AS dopplr.auth;
USE 'http://www.datatables.org/dopplr/dopplr.city.info.xml' AS
dopplr.city.info;
USE 'http://www.datatables.org/dopplr/dopplr.futuretrips.info.xml' AS
dopplr.futuretrips.info;
USE 'http://www.datatables.org/dopplr/dopplr.traveller.fellows.xml' AS
dopplr.traveller.fellows;
```

**Tip**

The AS verb in the above example is optional. If you omit the AS verb, YQL uses the filename (without the .xml file ending) to name the table.

Once you upload the environment file to your server, you can simply access the YQL console and append the location of the file as follows:

```
http://developer.yahoo.com/yql/console/?env=http://datatables.org/allt-ables.env
```

[Try this example in the YQL console](http://developer.yahoo.com/yql/console/?env=http://datatables.org/alltables.env)<sup>1</sup>



### Tip

You can include multiple environment files at once by using multiple “env” query parameters. These are loaded in the order they appear in the query string.

```
http://developer.yahoo.com/yql/console/?env=http://datatables.org/alltables.env
&env=http://website.com/mytable.env
```

## Working with Nested Environment Files

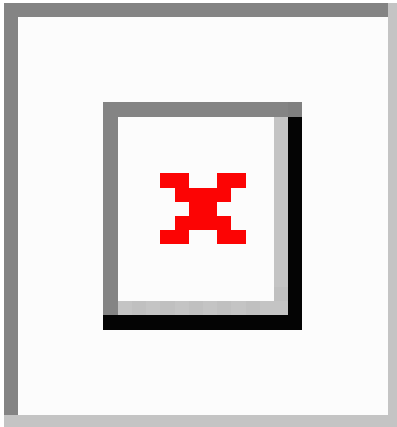
Advanced YQL users can better organize their Open Data Tables by nesting environment files within other environment files. For example, you can group similar tables into different environments and include each environment in a main (root) environment file as necessary. In addition to invoking environments, you can apply any SET statements to environments you invoke.

The following example both invokes multiple environments within a root environment file,

```
// Include the search group of tables
env 'http://foo.com/searchgroup.env';
// Also use this api-key on all of these envs
set apikey='abcsecret' on search;

// Also include the Flickr group since my env requires them
env 'http://foo.com/flickr.env';
```

**Figure 6.1. Environment File Transversal**



## Usage Rules

YQL processes environment files using an in-order traversal. In the example diagram above:

- (A) is the root environment file.
- (B) and (D) are environments that are invoked by (A).

---

<sup>1</sup><http://developer.yahoo.com/yql/console/?env=http://datatables.org/alltables.env>

- (C) and (E) are environments that are invoked by (E).
- Tables are added in the following order (C), (B), (E), (D), (A) assuming that all tables are defined after the environment inclusions.
- The order in which you define tables can be changed by first using `USE` statements and then including other environments.
- `SET` statements can be called only on tables defined in either the defining environment **or** any of its parents. For example, in the example above, sets in (A)/(B)/(C) can apply to tables defined in (C).
- YQL detects and disallows circular references in environments inclusions.
- The last table definition for a given name takes precedence in a given environment.
- `SET` statements do not work across environments or across peers. For example, in the above example, (D) cannot set a key on tables defined in (B) or (C).
- If the same environment is included again as a peer, the last inclusion wins and overrides all previously `SET` statements.
- `SET` statements are scoped and the farthest parent with the most appropriate table prefix match wins.

## Setting Key Values for Open Data Tables

For greater convenience and security, YQL allows you to set up key values for use within Open Data Tables. You can set values, such as passwords, API keys, and other required values, independently of YQL statements and API calls.

The following example sets the `api_key` value within the YQL statement itself:

```
select * from guardian.content.search where api_key="1234567890" and
query="environment"
```

The `SET` keyword allows you to set key values outside of a YQL statement, including [environment files \[44\]](#). The `SET` keyword uses the following syntax within an [environment file \[44\]](#):

```
SET api_key="1234567890" ON guardian;
```

In the example above, `SET` is followed by the key (`api_key`) and its value (1234567890). You must also specify the prefix of the table, which in this case is `guardian`.

Once you set the key value within an environment file, remove these values in the YQL statement:

```
select * from guardian.content.search where query="environment"
```

Because key values set in an environment file using the `SET` keyword are not limited to a specific table binding, those keys must be defined as optional in the [select binding \[3\]](#) to prevent them from being used as local filters in `SELECT` statements.

For example, suppose you want to set the `game_status` value for `INSERT` statements with the following:

```
SET game_status="Available to play" ON online_games;
```



Although you only intended to use `game_status` for INSERT statements, the key would still be used as a local filter in a SELECT statement **unless** it was declared as an optional key (with its `required` attribute set to "false") in the select binding, as shown here:

```
...
<bindings>
  <select itemPath="online_games.player.updates" produces="XML">
    ...
    <inputs>
      <key id="game_status" type="xs:string" paramType="variable"
required="false" />
      ...
    </inputs>
    ...
  </select>
</binding>
...
```

The following precedence rules apply when setting key values with the SET keyword:

- Keys that are set within the YQL statement take precedence over keys that are set using the SET keyword.
- If the set key is defined more than once, the most precise definition, based on the length of the table prefix, takes precedence.
- If the set key is defined more than once with the same preciseness, the last definition is used.

## Using SET to Hide Key Values or Data

To avoid exposing private data when you share YQL Open Data Tables, you can use a combination of YQL features to hide such data:

1. Add your private values to an environment file using the **SET keyword**.
2. Use the `yql.storage.admin` table to import the environment file or an Open Data Table with a memorable name. YQL provides you with a set of shared access keys.
3. [Use the shared execute or select access keys \[61\]](#) as you would an Open Data Table, environment file, or JavaScript.



### Important

Ensure that you place all USE and SET statements together respectively in one environment file to prevent private data being handed over to redefined tables.

# Creating YQL Open Data Tables

---

## Creating YQL Open Data Tables

### Abstract

This part of the YQL Guide provides a reference for YQL Open Data Tables. It also discusses using server-side JavaScript and hosted storage to extend the abilities of Open Data Tables. Before reading this part of the YQL Guide, you should first read [Introducing YQL \[1\]](#) and [Using YQL and Open Data Tables \[1\]](#).

---

---

# Table of Contents

1. Creating YQL Open Data Tables .....	1
Before You Begin .....	1
Open Data Tables Reference .....	1
table element .....	2
meta element .....	2
select / insert / update / delete elements .....	3
function Element .....	4
url element .....	5
execute element .....	5
key / value / map elements .....	6
paging element .....	9
YQL Streaming .....	11
Configuring Open Data Tables to Use Streaming .....	11
Using Open Data Tables Configured for Streaming .....	12
Debugging Open Data Tables and YQL Network Calls .....	12
Enabling Logging .....	13
Viewing Logs .....	13
Open Data Table Examples .....	14
Flickr Photo Search .....	14
Digg Events via Gnip .....	16
Open Data Tables Security and Access Control .....	17
Batching Multiple Calls into a Single Request .....	17
Troubleshooting .....	18
2. Executing JavaScript in Open Data Tables .....	19
Introduction .....	19
Features and Benefits .....	19
Ensuring the Security of Private Information .....	20
JavaScript Objects, Methods, and Variables Reference .....	20
y Global Object .....	21
request Global Object .....	31
rest Object .....	31
response Global Object .....	38
result Object .....	38
Global Variables .....	38
JavaScript and E4X Best Practices for YQL .....	39
Paging Results .....	39
Including Useful JavaScript Libraries .....	40
Using E4X within YQL .....	40
JavaScript Logging and Debugging .....	42
Executing JavaScript Globally .....	43
Making Asynchronous Calls with JavaScript Execute .....	44
Examples of Open Data Tables with JavaScript .....	45
Hello World Table .....	45
Yahoo! Messenger Status .....	46
OAuth Signed Request to Netflix .....	47
Request for a Flickr frob .....	48
Celebrity Birthday Search using IMDB .....	49
Shared Yahoo! Applications .....	53
CSS Selector for HTML .....	55
Execution Rate Limits .....	56
3. Using Hosted Storage with YQL .....	58

Introduction .....	58
About YQL Hosted Storage .....	58
Storage Limits and Requirements .....	58
Storing New Records .....	59
Storing a New Record using Text .....	59
Storing a New Record using Data from an URL .....	60
Storing a New Named Record using Data from an URL .....	60
Using YQL to Read, Update, and Delete Records .....	60
Accessing Records using YQL .....	60
Deleting Records using YQL .....	61
Updating Records using YQL .....	61
Using Records within YQL .....	61
Using Hosted Environment Files .....	62
Using Hosted YQL Open Data Tables .....	62
Including Hosted JavaScript .....	62

---

# Chapter 1. Creating YQL Open Data Tables

**In this Chapter:**

- [“Open Data Tables Reference” \[1\]](#)
- [“Debugging Open Data Tables and YQL Network Calls” \[12\]](#)
- [“Open Data Table Examples” \[14\]](#)
- [“Open Data Tables Security and Access Control” \[17\]](#)
- [“Batching Multiple Calls into a Single Request” \[17\]](#)
- [“Troubleshooting” \[18\]](#)

## Before You Begin

Before reading this chapter, you should first read [Introducing YQL \[1\]](#) and [Using YQL and Open Data Tables \[1\]](#)

## Open Data Tables Reference

The following reference describes the structure of an Open Data Table definition:

The following elements and sub-elements of YQL Open Data Tables are discussed in this reference:

- [tables \[2\]](#)
- [meta \[2\]](#)
- bindings
  - [select / insert/ update / delete \[3\]](#)
  - urls
    - [url \[5\]](#)
  - inputs
    - [key / value / map \[6\]](#)
- [paging \[9\]](#)
  - [pagesize \[10\]](#)
  - [start \[10\]](#)
  - [total \[10\]](#)
  - [nextpage \[11\]](#)

## table element

**Full Path:** *root element*

**Example:**

```
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
```

This is the root element for the document. A table is the level at which an end-user can 'select' information from YQL sources. A table can have many different bindings or ways of retrieving the data. However, we advise that a single table produce a single type of result data.

Attribute	Value(s)	Notes
xmlns	URL	The XML Schema file related to this Open Data Table definition.
security-Level	enumeration, any / app / user	<p>The authorization level required to access.</p> <p><b>any:</b> Anonymous access; any user can access this table.</p> <p><b>app:</b> 2-legged OAuth; involves authorization without access to private user data.</p> <p><b>user:</b> 3-legged OAuth, involves authorization of access to user data.</p> <p>For more information, refer to <a href="#">Open Data Tables Security and Access Control [17]</a>.</p>
https	boolean, true or false	If true, the table is only available if the user is connected via HTTPS. If missing or false, either HTTP or HTTPS connections are acceptable.



### Warning

If your table requires input that is deemed "private", such as any passwords, authentication keys, or other "secrets", you **MUST** ensure the `https` attribute within the table element is set to `true`.

## meta element

**Full Path:** *table/meta*

**Example:**

```
<meta>
  <author>Yahoo! Inc.</author>

  <documentationURL>http://www.flickr.com/services/api/flickr.photos.search.html</documentationURL>

  <sampleQuery>select * from {table} where has_geo="true" and text="san francisco"</sampleQuery>
</meta>
```

Along with the table element, you are required to include the meta sub-element, which provides the following information:

Attribute	Description	Notes
sampleQuery	A sample query that users can run to get output from this table.	{table} should be used in place of the table name, so that the sample can run when used in different namespaces. Multiple sampleQuery elements may occur. Each sampleQuery may have a description attribute that contains a description about the sample.
documentationURL	Additional information about this table or the select called by the table can be found here	More than one documentationURL element may be included for each table.
description	Plain text description about the table	A description of the table.
author	Information regarding the author of this Web service	Examples of author information include an unformatted email, name, or other related information.

## select / insert / update / delete elements

### Full Path:

table/bindings/select

table/bindings/insert

table/bindings/update

table/bindings/delete

### Example:

```
<bindings>
  <select itemPath="rsp.photos.photo" produces="XML">
    ...
</bindings>
```

Situated within each bindings element, there are one of four keywords: select, insert, update, or delete.

The select element describes the information needed for YQL to read data from an API. The insert and update elements describe the information needed to add or modify data from an API, respectively. When removing data, the delete element is used to describe the necessary bindings.

When a keyword such as select or update is repeated within the bindings array, it can be considered to be an alternative way for YQL to call a remote server to get the same type of structured data. Typically, this is needed when the service supports different sets of query parameters (YQL's "keys") or combinations of optional query parameters.

Attribute	Value(s)	Notes
itemPath	URLs	A dot-path that points to where the repeating data elements occur in the response format. These are the "rows" of your table.



Attribute	Value(s)	Notes
pollingFrequencySeconds	Integer	The frequency in seconds that the YQL Engine should update the response from a data source. See <a href="#">YQL Streaming</a> for more information.
produces	enumeration, XML / JSON	The type of data coming back from the Web service.



### Note

Unlike XML, JSON objects have no "root" node. To work with the dot notation, YQL creates a "pseudo" root node for JSON responses called "json". If you need to return a sub-structure from your Open Data Table that fetches or produces JSON, you'll need to add "json" at the root of the path.

## function Element

A stored procedure to bind and run, similar to the built-in [Sort and Other Functions](#)<sup>1</sup>. For more information on the function element, see [Creating YQL Customized Functions](#).

**Full Path:** table/bindings/function

**Example:**

```
<table>
  <bindings>
    <function name="concat" />
      <inputs>
        ...
      </inputs>
      <execute>
        ...
      </execute>
    </function>
    ...
  </bindings>
</table>
```

Attribute	Type	Description
name	string	The name of the function.
type	enumeration	<p>The type of function. Allowed values:</p> <ul style="list-style-type: none"> <li><b>stream</b> - (default) This type of function gets called for each item at which point it can modify and return the modified version of the item. Example: Decorate an item with extra information.</li> <li><b>reduce</b> - This type of function gets called for each item which is collected/stored and then finally, gets one last call to return the reduced or aggregated response. Examples of YQL built-in reduce functions: sort, truncate, count.</li> </ul>

<sup>1</sup><http://developer.yahoo.com/yql/guide/sorting.html>

## url element

**Full Path:** table/bindings/select/urls/url

This is where YQL and the table supporting the service come together. The url element describes the URL that needs to be executed to get data for this table, given the keys in the key elements.



### Note

The CDATA/TEXT for this element contains the URL itself that utilizes substitution of values at runtime based on the [uri template spec \(v3\)](http://tools.ietf.org/html/draft-gregorio-uritemplate-03)<sup>2</sup>. The names of the values will be substituted and formatted according to the uri template spec, but the simplest method is simply to enclose a key name with curly braces ( { } ):

- All {name} keys found in the URL will be replaced by the same id key value in the keys elements.
- YQL currently supports both http and https protocols.

Example:

```
https://prod.gnipcentral.com/publishers/{publisher}/notification/{bucket}.xml
```

YQL will look for key elements with the names publisher and bucket. If the YQL developer does not provide those keys in the WHERE clause (and they are not optional), then YQL detects the problem and will produce an error. If an optional variable is not provided, but is part of the Open Data Table definition, it will be replaced with an empty string. Otherwise, YQL will substitute the values directly into the URL before executing it.

## execute element

**Full Path:** table/bindings/select/execute

The execute sub-element allows you to invoke server-side JavaScript in place of a GET request. For more information on executing JavaScript, refer to [Executing JavaScript within Open Data Tables \[19\]](#).

**Example:**

```
<execute>
  <![CDATA[
    // Include the flickr signing library

y.include("http://blog.pipes.yahoo.net/wp-content/uploads/flickr.js");
    // GET the flickr result using a signed url
    var fs = new flickrSigner(api_key,secret);
    response.object = y.rest(fs.createUrl({method:method,
format:""})).get().response();
```

---

<sup>2</sup> <http://tools.ietf.org/html/draft-gregorio-uritemplate-03>

```
]]>  
</execute>
```

## key / value / map elements

### Full Paths:

```
table/bindings/[select/insert/update/delete]/inputs/key
```

```
table/bindings/[select/insert/update]/inputs/value
```

```
table/bindings/[select/insert/update/delete]/inputs/map
```

### Example:

```
<inputs>  
  <key id='guid' type='xs:string' paramType='path' required="true"  
/>  
  <key id='ck' type='xs:string' paramType='variable'  
required="true" />  
  <key id='cks' type='xs:string' paramType='variable'  
required="true" />  
  <value id='content' type='xs:string' paramType='variable'  
required="true" />  
</inputs>
```

There are three type of elements available within the `inputs` element: `key`, `value`, and `map`.

## key element

Each key element represents a named "key" that you provide in the `WHERE` or `INTO` clause of `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements. YQL inserts these values into the URL request before it is sent to the server. YQL inserts these values into the URL request if the `paramType` is set to `query` or `path` or `header`. For a `variable` type, the key named as the `id` of the element is made available in the `execute` section of the Open Data Table.

## value element

Use the `value` element to assign a new "value" or update an existing one within an Open Data Table. The `value` element defines a field that can only be set as an input and therefore cannot be in YQL statements to satisfy the "where" clause. The `value` element only works with the `INSERT` and `UPDATE` verbs and in different ways.

When used with the `insert` keyword, the `value` element appears in the `VALUE` expression of the YQL statement, indicating that a new value is being passed into the YQL statement, as seen in the following example:

```
INSERT into bitly.shorten (login, apiKey, longUrl) VALUES ('YOUR_LOGIN',  
'YOUR_API_KEY', 'http://yahoo.com')
```

When used with the `update` keyword, the `value` element is called from the `SET` portion of the YQL statement. This indicates that you are "setting" a particular value, as seen in the following example:

```
UPDATE table SET status='Reading the YQL Guide' where guid = me;
```

## map element

Use the map element when you want to use dynamic keys. With this element, YQL uses the value you pass in through the YQL statement as a variable. This variable is used within the execute portion of your Open Data Table to determine what action to take. For example, you may set up a YQL Open Data Table that updates either bit.ly, delicio.us, or tinyurl, depending on the value you specify in the YQL statement.

For a dynamic key called `type`, the actual ID in a YQL query would look like the following:

```
field.type = 'Java'
```



### Note

In the absence of the map element as a binding, all identifiers, not corresponding to a binding element and that appear in a YQL query, are treated as local filters.

The map element can be used for all the four [paramTypes \[9\]](#). Here is an example of the map element being used in a path:

```
<map id="field" paramType="path"/>
```

For a query containing the relational expression `field.type = 'rss'`, only the dynamic parameter name `type` would be substituted in the `urls` element. The URI template would look like the following:

```
http://rss.news.yahoo.com/{type}/topstories
```

## Key, Value, and Map Element Support within YQL Statements

The following table shows the keywords that support the key, value, and map elements:

	<b>select</b>	<b>insert</b>	<b>update</b>	<b>delete</b>
<b>key</b>	yes	yes	yes	yes
<b>value</b>	no	yes	yes	no
<b>map</b>	yes	yes	yes	yes

## Attributes for Key, Value, and Map Elements

The following table provides the attributes available within key, value and map elements:

<b>Attribute</b>	<b>Value(s)</b>	<b>Supported Keywords</b>	<b>Notes</b>
<b>id</b>	string	select, insert, update, delete	The name of the key. This represents what the user needs to provide in the <b>WHERE</b> clause.
<b>as</b>	string	select, insert, update, delete	The alias of the key used in YQL statements.  If the Web source used in the Open Data Table uses a cryptic or poorly named query parameter, you can use <b>as</b> to specify an alias that developers use in the YQL statement. For example, perhaps you have an <b>id</b> called "q" within your Open Data Table, which actually is a search parameter.

Attribute	Value(s)	Supported Keywords	Notes
			<p>Without aliasing, the equivalent YQL statement would look like this:</p> <pre>select * from google.search where q = "pizza"</pre> <p>You can use the <code>as</code> attribute to create an alias in the following way:</p> <pre>&lt;key      id="q"      as="query" type="xs:string" paramType="query"/&gt;</pre> <p>You then can use <code>search</code> in your YQL statement like this:</p> <pre>select * from google.search where query = "pizza"</pre>
<code>type</code>	string	select, insert, update, delete	The type of data coming back from the Web service.
<code>required</code>	boolean	select, insert, update, delete	A boolean that answers the question: Is this key required to be provided in the <code>WHERE</code> clause on the left-hand side of an equality statement? If not set, any key is optional.
<code>paramType</code>	enumeration	select, insert, update, delete	<p>Determines how this key is represented and passed on to the Web service:</p> <ul style="list-style-type: none"> <li>- <code>query</code>: Add the id and its value as a <code>id=value</code> query string parameter to the URL.</li> <li>- <code>matrix</code>: Add the id and its value as a <code>id=value</code> matrix parameter to the URL path.</li> <li>- <code>header</code>: Add the id and its value as a <code>id: value</code> as an HTTP header to the URL request.</li> <li>- <code>path</code>: Substitute all occurrences of <code>{id}</code> in the url string with the value of the id. Not necessarily only in the path.</li> <li>- <code>variable</code>: Use this key or field as a variable to be used within the <a href="#">execute sub-element [19]</a> instead of being used to format or form the URL.</li> </ul>
<code>default</code>	string	select, insert, update, delete	This value is used if one isn't specified by the developer in the <code>SELECT</code> .
<code>private</code>	boolean	select, insert, update, delete	Hide this key's value to the user (in both <code>"desc"</code> and <code>"diagnostics"</code> ). This is useful for parameters like <code>appid</code> and <code>keys</code> .
<code>const</code>	boolean	select, insert, update, delete	A boolean that indicates whether the <code>default</code> attribute must be present and cannot be changed by the end user. Constant keys are not shown in <code>desc [table]</code> .

Attribute	Value(s)	Supported Keywords	Notes
batchable	boolean	select, update, delete	A boolean which answers the question: Does this select and URL support multiple key fetches/requests in a single request (batched fetching)?  For more information about batching requests, refer to <a href="#">Batching Multiple Calls in a Single Request [17]</a> .
maxBatchItems	integer	select, update, delete	How many requests should be combined in a single batch call.  For more information about batching requests, refer to <a href="#">Batching Multiple Calls in a Single Request [17]</a> .

## Aliasing within Key, Value, and Map Elements

If you have an obscurely named `id` in your Open Data Table, you can use an alias to refer to it within YQL statements. For example, perhaps you have an `id` called "q" within your Open Data Table, which actually is a search parameter. You can use "as" to create an alias in the following way:

```
<key id="q" as="type" xs:string paramType="query"/>

select * from google.search where search ="pizza"
```

## paging element

**Full Path:** `table/bindings/select/paging`

**Examples:**

```
<paging model="page">
  <start id="page" default="0" />
  <pagesize id="per_page" max="250" />
  <total default="10" />
</paging>
```

```
<paging model="url">
  <nextpage path="ysearchresponse.nextpage" />
</paging>
```

```
<paging model="offset" matrix="true">
  <start id="page" default="0" />
  <pagesize id="per_page" max="250" />
  <total default="10" />
</paging>
```

This element describes how YQL should "page" through the web service results, if they span multiple pages, or the service supports offset and counts.

Attribute	Value(s)	Supported Keywords	Notes
model	offset, page, url	select	The type of model to use to fetch more than the initial result set from the service.

Attribute	Value(s)	Supported Keywords	Notes
			<p>The <code>offset</code> refers to services that allow arbitrary index offsets into the result set.</p> <p>Use the <code>page</code> value for services that support distinct "pages" of some number of results.</p> <p>Use the <code>url</code> value for services that support a URL to access further data.</p>
<code>matrix</code>	<code>true, false</code>	<code>select</code>	A boolean that answers the question: Is the parameter matrix style (part of the URI path; delimited), or query parameter style? The default value is <code>false</code> .



### Tip

When using the `url` paging model, you can also use the `pagesize` element to, if the Web service allows, adjust the number of results returns at once.

## pagesize element

**Full Path:** `table/bindings/select/paging/pagesize`

This element contains Information about how the number of items per request can be specified.

Attribute	Value(s)	Notes
<code>max</code>	<code>integer</code>	The maximum size of the requested page. If the total requested is below the max <code>pagesize</code> , then the <code>pagesize</code> will be the total requested. Otherwise, the max <code>pagesize</code> will be the size of the page requested.
<code>id</code>	<code>string</code>	The name of the parameter that controls this page size.

## start element

**Full Path:** `table/bindings/select/paging/start`

This element contains Information about how the "starting" item can be specified in the set of results.

Attribute	Value(s)	Notes
<code>default</code>	<code>integer</code>	The starting item number (generally 0 or 1); for paging style this value always defaults to 1.
<code>id</code>	<code>string</code>	The name of the parameter that controls the starting page/offset.

## total element

**Full Path:** `table/bindings/select/paging/total`

This element contains Information about the total number of results available per request by default.

Attribute	Value(s)	Notes
default	integer	The number of items that come back by "default" in YQL if the ( ) syntax is not used when querying the table.

## nextpage element

**Full Path:** table/bindings/select/paging/nextpage

This element contains the location of the next page of results. This is an optional element that is used in conjunction with the parent url element.

Attribute	Value(s)	Notes
path	string	The path to the next page of results

## YQL Streaming

In general, when a YQL statement references an Open Data Table, the YQL Web service will make an HTTP call based on the URLs listed in the bindings element or execute the JavaScript in the execute element. To get updated results from the YQL Web service, you need to execute this YQL statement again. YQL streaming allows you to get updated results in a different way.

With YQL streaming, your Open Data Table can request YQL to poll data from your resource URI at a specified interval, process the responses, and return the results to the client, without the client having to execute the same YQL statements over and over again to get the updated results; this will reduce network latency and the need to store responses.

## Configuring Open Data Tables to Use Streaming

To request streaming in your Open Data Table, you use the `pollingFrequencySeconds` attribute of the [select binding \[7\]](#) to specify the interval that you want YQL to make requests. For example, the code snippet below configures YQL to poll the URL for Foursquare search every two seconds:

```
<bindings>
  <select itemPath="" pollingFrequencySeconds="2"/>
  <urls>
    <url>https://api.foursquare.com/v2/users/search</url>      </urls>
  <inputs>
    <key id="oauth_token" type="xs:string" paramType="query"
required="true" />
    <key id="phone" type="xs:string" paramType="query" />      <key
id="email" type="xs:string" paramType="query" />      <key id="twitter"
type="xs:string" paramType="query" />      <key id="twitterSource"
type="xs:string" paramType="query" />
    <key id="fbid" type="xs:string" paramType="query" />      <key
id="name" type="xs:string" paramType="query" />
  </inputs>
</binding>
```





## Note

When setting a polling frequency with `pollingFrequencySeconds`, try to match the update frequency of your data source. If you configure YQL to poll the source every two seconds, but the source usually updates data every minute, you will be increasing the network latency for YQL without the benefit of getting new data.

## Using Open Data Tables Configured for Streaming

To use Open Data Tables that are configured for streaming, you must call the YQL Web Service URLs for streaming. You use YQL statements and invoke the Open Data Table in the same way as you would for any Open Data Table.

The following YQL Service URL is used to get public data from an Open Data Table that is configured to use YQL streaming:

```
http://query.yahooapis.com/v1/public/streaming/yql
```

The YQL Service URL below requires authorization by OAuth and allows access to both public and private data from an Open Data Table that is configured to use YQL streaming:

```
http://query.yahooapis.com/v1/streaming/yql
```

The following PHP code snippet shows how to get public data from the Open Data Table `books_warehouse.xml` that is configured to use YQL streaming:

```
// The YQL Service URL for streaming public data
$YQL_STREAMING_URL =
"https://query.yahooapis.com/v1/public/streaming/yql";
// Form YQL query and build URI to YQL Web service    $yql_query = "use
'http://yql.lotsofbooks.com/books_warehouse.xml' as bw; select * from
bw where isbn-10='9781849510707'";

$yql_query_url = $YQL_STREAMING_URL . "?q=" . urlencode($yql_query) .
"&format=json";

// Make call with cURL and get returned response
$session = curl_init($yql_query_url);    curl_setopt($session,
CURLOPT_RETURNTRANSFER,true);
$json = curl_exec($session);

// Convert JSON to PHP object
$phpObj = json_decode($json);
```

## Debugging Open Data Tables and YQL Network Calls

To aid in your debugging efforts, YQL provides the option to have network-level logging. When enabled, all network requests are uncached, so you can iteratively develop Open Data Tables more easily, as well as debug network requests between YQL and the remote service. Network logs display both the request headers and the response content for each network call.



When you enable network-level logging, YQL provides a key within the `diagnostics` element for each network call that occurs, seen in the following YQL response snippet as `id` attributes:

The `id` key can be used within 5 minutes of an execution to see log data.

To enable network-level logging, you simply append `debug=true` to the YQL console URL or API query like this:

You can access network-level logs within 5 minutes of running a YQL statement or call. You can view the logs using the **YQL Console** or using the YQL log URL with the query string parameter `id`.

1. From the **YQL Console**, check the **Diagnostics** and **Debug** checkboxes, enter your YQL statement, and click **TEST**.
2. From the returned response in the **FORMATTED** tab, click on the value for the `id` attribute of the `url` element in the diagnostic information.
3. A new tab will open showing log data that includes the HTTP request, the HTTP headers, and the raw response.

## Using the YQL Log URL

1. Append the query string `debug=true&diagnostics=true` to one of the [YQL Web service URLs](#)<sup>3</sup>.  
For example: `http://query.yahooapis.com/v1/public/yql?q=<your_query_statement>&debug=true&diagnostics=true`
2. Make the HTTP request to the YQL Web service URL with the `debug` and `diagnostics` query parameters.
3. From the returned response, copy the value for the `id` key provided in the `url` element of the diagnostics information and append it to the YQL log URL as shown here: `http://query.yahoo.com/v1/logging/dump?id=<id_value>`
4. You will be returned the dumped log data that includes the HTTP request, the HTTP headers, and the raw response.

## Open Data Table Examples

This section includes a few examples of Open Data Tables that showcase the ability of YQL to gather data from external APIs.



### Tip

For a much larger list of publicly available Open Data Tables, refer to [datatables.org](http://datatables.org)<sup>4</sup>.

- [Flickr Photo Search \[14\]](#)
- [Access to Digg Events using Gnip \[16\]](#)

## Flickr Photo Search

This Open Data Table definition ties into the Flickr API and allows YQL to retrieve data from a Flickr photo search:

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta> [2]
    <author>Yahoo! Inc.</author>

  <documentationURL>http://www.flickr.com/services/api/flickr.photos.search.html</documentationURL>

  <sampleQuery>select * from {table} where has_geo="true" and text="san francisco"</sampleQuery>
  </meta> [2]
  <bindings>
    <select itemPath="rsp.photos.photo" produces="XML">
      <urls>
        <url
env="all">http://api.flickr.com/services/rest/?method=flickr.photos.search</url>
```

<sup>3</sup> [yql\\_url.html](#)

<sup>4</sup> <http://datatables.org>

```

</urls>
<paging model="page">
  <start id="page" default="0" />
  <pagesize id="per_page" max="250" />
  <total default="10" />
</paging>
<inputs>
  <key id="woe_id" type="xs:string" paramType="query" />
  <key id="user_id" type="xs:string" paramType="query" />
  <key id="tags" type="xs:string" paramType="query" />
  <key id="tag_mode" type="xs:string" paramType="query" />
  <key id="text" type="xs:string" paramType="query" />
  <key id="min_upload_date" type="xs:string" paramType="query"
/>
  <key id="max_upload_date" type="xs:string" paramType="query"
/>
  <key id="min_taken_date" type="xs:string" paramType="query" />
  <key id="max_taken_date" type="xs:string" paramType="query" />
  <key id="license" type="xs:string" paramType="query" />
  <key id="privacy_filter" type="xs:string" paramType="query" />
  <key id="bbox" type="xs:string" paramType="query" />
  <key id="accuracy" type="xs:string" paramType="query" />
  <key id="safe_search" type="xs:string" paramType="query" />
  <key id="content_type" type="xs:string" paramType="query" />
  <key id="machine_tags" type="xs:string" paramType="query" />
  <key id="machine_tag_mode" type="xs:string" paramType="query"
/>
  <key id="group_id" type="xs:string" paramType="query" />
  <key id="contacts" type="xs:string" paramType="query" />
  <key id="place_id" type="xs:string" paramType="query" />
  <key id="media" type="xs:string" paramType="query" />
  <key id="has_geo" type="xs:string" paramType="query" />
  <key id="lat" type="xs:string" paramType="query" />
  <key id="lon" type="xs:string" paramType="query" />
  <key id="radius" type="xs:string" paramType="query" />
  <key id="radius_units" type="xs:string" paramType="query" />
  <key id="extras" type="xs:string" paramType="query" />
  <key id="api_key" type="xs:string" const="true" private="true"
paramType="query" default="45c53f8...d5f645" />
</inputs>
</select>
</bindings>
</table>

```

[Run this example in the YQL console.](http://developer.yahoo.com/yql/console/?q=select%20*%20from%20flickr.photos.search%20where%20has_geo%3D%22true%22%20and%20text%3D%22san%20francisco%22&env=http%3A%2F%2Fgithub.com%2Fspullara%2Fyql-tables%2Fraw%2Fef685688d649a7514ebd27722366b2918d966573%2Falltables.env)<sup>5</sup>

<sup>5</sup> [http://developer.yahoo.com/yql/console/?q=select%20\\*%20from%20flickr.photos.search%20where%20has\\_geo%3D%22true%22%20and%20text%3D%22san%20francisco%22&env=http%3A%2F%2Fgithub.com%2Fspullara%2Fyql-tables%2Fraw%2Fef685688d649a7514ebd27722366b2918d966573%2Falltables.env](http://developer.yahoo.com/yql/console/?q=select%20*%20from%20flickr.photos.search%20where%20has_geo%3D%22true%22%20and%20text%3D%22san%20francisco%22&env=http%3A%2F%2Fgithub.com%2Fspullara%2Fyql-tables%2Fraw%2Fef685688d649a7514ebd27722366b2918d966573%2Falltables.env)



## Tip

To get a better understanding of how bindings work within YQL Open Data Tables, compare the Open Data Table definition above to [photo.search on the Flickr API](#)<sup>6</sup>.

## Digg Events via Gnip

The following example ties into the [Gnip API](#)<sup>7</sup> to retrieve activities from a Publisher, which in this case is **Digg**.

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where publisher='digg' and
action='dugg'</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="activities.activity" produces="XML" >
      <urls>
        <url
env="all">https://prod.gnipcentral.com/publishers/{publisher}/notification/{bucket}.xml</url>

        </urls>
        <inputs>
          <key id="publisher" type="xs:string" paramType="path"
required="true" />
          <key id="bucket" type="xs:string" paramType="path"
required="true" />
          <key id="Authorization" type="xs:string" paramType="header"
const="true" default="Basic eXFsLXF1ZXN...BpcGVz" />
        </inputs>
      </select>
      <select itemPath="activities.activity" produces="XML" useProxy="true"
auth="callback">
        <urls>
          <url
env="all">https://prod.gnipcentral.com/publishers/{publisher}/notification/current.xml</url>

          </urls>
          <inputs>
            <key id="publisher" type="xs:string" paramType="path"
required="true" />
            <key id="Authorization" type="xs:string" paramType="header"
const="true" default="Basic eXFsLXF1ZXN0a...BpcGVz" />
          </inputs>
        </select>
```

<sup>6</sup> <http://www.flickr.com/services/api/flickr.photos.search.html>

<sup>7</sup> [http://docs.google.com/View?docid=dgkhvp8s\\_5svzn35fw#Examples\\_of\\_Activities](http://docs.google.com/View?docid=dgkhvp8s_5svzn35fw#Examples_of_Activities)

```
</bindings>
</table>
```

[Run this example in the YQL console.](#)<sup>8</sup>

## Open Data Tables Security and Access Control

The `securityLevel` attribute of the table element determines the type of authentication required to establish a connection. In order for a user to connect to your table, the user must be authorized at the level or higher than the level indicated in the `securityLevel` attribute. The following table lists whether access is available depending on the value in the `securityLevel` attribute.

Security of Table ( <code>access attribute</code> )	Anonymous / No Authorization	2-legged OAuth	3-legged OAuth / cookie
any	yes	yes	yes
app	no	yes	yes
user	no	no	yes

For more information about each level, refer to the [securityLevel attribute in the table element \[2\]](#).

## Batching Multiple Calls into a Single Request

YQL Open Data Tables support the ability to send a list of keys in a single request, batching up what would otherwise be a multiple calls.



### Note

In order to do batching in YQL Open Data Tables, the source must support it. An example of a source that supports batching is the [Yahoo! Social Directory call for profiles](#)<sup>9</sup>

Let's take the example of the Social Directory API and see the URI for profile data:

```
http://social.yahooapis.com/v1/user/{guid}/profile
```

In YQL, the table for retrieving this data is `social.profile`. Here is an example that includes a sub-select:

```
select * from social.profile where guid in (select guid from
social.connections where owner_guid = me)
```

When performing sub-selects, the inner sub-select returns a set of values, each of which is a call to the URI above. So if a Yahoo! user has 3 connections in his profile, the sub-select makes three calls to the Social Directory API:

```
http://social.yahooapis.com/v1/user/1/profile
http://social.yahooapis.com/v1/user/2/profile
http://social.yahooapis.com/v1/user/3/profile
```

<sup>8</sup> [http://developer.yahoo.com/yql/console/?q=select%20\\*%20from%20gnip.activity%20where%20publisher%3D%27digg%27%20and%20action%3D%27dugg%27](http://developer.yahoo.com/yql/console/?q=select%20*%20from%20gnip.activity%20where%20publisher%3D%27digg%27%20and%20action%3D%27dugg%27)

<sup>9</sup> [http://developer.yahoo.com/social/rest\\_api\\_guide/extended-profile-resource.html](http://developer.yahoo.com/social/rest_api_guide/extended-profile-resource.html)

Fortunately, the Social Directory URI above also supports batching, so a single call can be made to get all three profiles:

```
http://social.yahooapis.com/v1/users.guid(1,2,3)/profile
```

Since the Social Directory API supports batching, YQL can enable this by [defining the key guid as batchable \[6\]](#) with an extra parameter that denotes the max number of batch items per request:

```
<key id="guid" type="xs:string" paramType="path" batchable="true"
maxBatchItems="3"/>
```

We also need to modify the Open Table definition to support multiple values for the GUID. Combining the modification to the Open Table definition above with the one below results in a batch call to the Social Directory API for not more than 3 profiles:

```
<url
env="int">http://socialstuff.com/v1/users.guid({-listjoin|,|guid})/profile</url>
```

## Troubleshooting

The following section deals with issues you may have while using YQL Open Data Tables:

- **Problem:** The SELECT URL doesn't parse correctly in my Open Data Table definition.

**Solution:** Make sure you've escaped things correctly for XML; for example: & should be encoded as &amp; ; .

- **Problem:** My Open Data Table definition has multiple bindings with different sets of keys. YQL keeps running the "wrong" select. How can I get YQL to choose the right one?

**Solution:** Keep in mind that the order of bindings is important. Once YQL finds a select that satisfies the YQL statement, it uses that one. Try moving the more "specific" select endpoint above the others.

- **Problem:** If my API requires authentication, how do I access it?

**Solution:** If you want to use an API that requires its own authentication mechanism, you use the [execute \[19\]](#) sub-element within an Open Data Table to manage this authentication.

- **Problem:** Open Data Tables seem so complicated? What is the best way to get started?

**Solution:** The best way to avoid being overwhelmed is to first look at [examples \[5\]](#). In general, when creating YQL tables, it is useful to take a bottom-up approach and analyze the result structure of the API(s) that you are encapsulating. First, group together all the services that produce the same result structure. This becomes your "table" or Open Table definition. For each API that produces the response structure, you should create a "select" under the "request" section of the Open Data Table definition. By using this mechanism, you can often consolidate multiple API's into a single versatile YQL table that allows YQL to do the heavy lifting and keep the implementation details hidden.

---

# Chapter 2. Executing JavaScript in Open Data Tables

In this Chapter:

- [“Introduction” \[19\]](#)
- [“Ensuring the Security of Private Information” \[20\]](#)
- [“JavaScript Objects, Methods, and Variables Reference” \[20\]](#)
- [“JavaScript and E4X Best Practices for YQL” \[39\]](#)
- [“Examples of Open Data Tables with JavaScript” \[45\]](#)
- [“Execution Rate Limits” \[56\]](#)

## Introduction

### Features and Benefits

The ability to execute JavaScript extends the functionality of [Open Data Tables \[43\]](#) in many ways, including the following:

- **Flexibility beyond the normal templating within Open Data Tables:** Executing JavaScript allows you to use conditional logic and to format data in a granular manner.
- **Better data shaping and parsing:** Using JavaScript, you can take requests and responses and format or shape them in way that is suitable to be returned.
- **Better support for calling external Web services:** Some Web services use their own security and authentication mechanisms. Some also require authentication headers to be set in the Web service request. The execute element allows you to do both.
- **Better support for adding, modifying, and deleting data using external Web services:** For Web services that support write access, YQL allows you to insert, update, and delete using server-side JavaScript within the insert, update, and delete elements, which are nested within the binding element.
- **Ability to make asynchronous calls:** YQL provides JavaScript methods for making REST calls that take a callback function parameter to handle the returned response. Using these methods with callback functions, you can initiate multiple calls that won't be blocked by a slow response or a timed-out request.

The ability to execute JavaScript is implemented through the `execute` sub-element within an Open Data Table definition.

Within the `execute` sub-element, you can embed JavaScript and E4X (the shortened term for ECMAScript for XML), which adds native XML support to JavaScript. Support for E4X was first introduced in JavaScript 1.6.

When a YQL statement calls an Open Table Definition that contains the `execute` sub-element, YQL no longer performs the request to the templated URI in the endpoint. Instead YQL provides a runtime envir-



onment in which the JavaScript is executed server-side. Your JavaScript in turn must then return data as the output to the original YQL statement.

## Ensuring the Security of Private Information

As mentioned earlier, a important feature of Open Data Tables is the ability to accommodate third-party security and authentication systems. As such, it is critical for developers to ensure an HTTPS connection is required in any case where "secret" or "private" information is being provided.

If your table requires input that is deemed "private", such as any passwords, authentication keys, or other "secrets", you **MUST** ensure the `https` attribute within the `table` element is set to `true`.

When YQL detects the `https` attribute is set to `true`, the table will no longer be usable for connections to the [YQL console](#)<sup>1</sup> or to the Web service API. To test and use tables securely, you should now use the HTTPS endpoints:

- **Console:** <https://developer.yahoo.com/yql/console>
- **Web Service API:** <https://query.yahooapis.com>



### Note

Connections made **from** the Open Data Table to the underlying Web services do **not** need to be made over HTTPS. The same holds true for the actual server hosting the Open Data Table definition.

For more information on the `https` attribute within Open Data Tables, refer to ["tables element" section within "Using Open Data Tables \[2\]"](#).

## JavaScript Objects, Methods, and Variables Reference

As you add JavaScript within your `execute` sub-element, you can take advantage of the following global objects:

Object	Description
<a href="#">y [21]</a>	Global object that provides access to additional language capabilities.
<a href="#">request [29]</a>	The <code>request</code> object is a global reference to the <a href="#">rest object [31]</a> that contains the URL on which YQL would normally perform a GET request. You cannot reference the <code>rest</code> object directly, so you need to use <code>request</code> or <code>y.rest</code> to access the <code>rest</code> object.
<a href="#">response [38]</a>	Response object returned as part of the "results" section of the YQL response.

Let's discuss each of the global objects in detail.

---

<sup>1</sup><http://developer.yahoo.com/yql/console>

## y Global Object

The `y` global object contains methods that provide basic YQL functionality within JavaScript. It also allows you to include YQL Open Data Tables and JavaScript from remote sources.

Method/Property	Description	Returns
<code>context</code>	Provides information about the context or environment that JavaScript code is running in. Currently, one property is supported: <code>context.table</code>	Returns the Open Data Table name where JavaScript within an execute element has run.
<a href="#">crypto [22]</a>	Provides basic cryptographic functions for use within JavaScript. Example:  <pre>var md5string = y.crypto.encodeMd5("encode this string to md5");</pre>	Returns an encrypted string.
<a href="#">date [23]</a>	Provides access to date-related functions.	JavaScript object
<a href="#">decompress(base64 compressed string) [24]</a>	Decodes a base64 string and then decompresses that string with gunzip. This method allows you to decompress the POST request body and query parameters.	Returns a string.
<code>diagnostics</code>	Returns diagnostic information related to the currently executed script.	Returns diagnostic information.
<code>env(environment file)</code>	Includes an environment file for use with your JavaScript. Example:  <pre>y.env("http://data-tables.org/alltables.env");</pre>	-
<a href="#">exit() , exit(status_code,msg) [25]</a>	Stops the execution of the current script.	-
<code>include(url)</code>	Includes JavaScript located at a remote URL.	Returns an evaluation of that include.
<code>jsonToXml(object)</code>	Converts a JavaScript/JSON object into E4X/XML.	E4X object
<code>log(message)</code>	Creates a log entry in diagnostics.	Returns the log entry within the diagnostics output associated with the current select statement
<a href="#">parseJson(json_str) [26]</a>	Returns a JavaScript object when given a well-formed JSON string.	JavaScript object
<a href="#">query(statement) [29]</a>	Runs a YQL statement.	A <a href="#">response object [38]</a> that contains a result instance or an error object

Method/Property	Description	Returns
<a href="#">query(statement, params, timeout, callback) [29]</a>	Prepares and runs a YQL statement. Execute will replace all @name fields in the YQL statement with the values corresponding to the name in the supplied hash table.	Creates a result instance, or returns an error.
<a href="#">rest(url, callback) [29]</a>	Sends a GET request to a remote URL endpoint.	-
<a href="#">sync() [31]</a>	Defers the JavaScript execution until pending asynchronous requests have been completed.	-
tidy (string html)	Tidy and return provided HTML.	Returns HTML that is run through HTML Tidy.
use(url, namespace)	Imports an external Open Data Table definition into the current script at runtime.	-
xpath(object, xpath)	Applies XPath to an E4X object.	Returns a new E4X object
xmlToJson(object)	Converts an E4X/XML object into a JSON object.	JavaScript object

## y.crypto functions

YQL provides several cryptographic functions for use within JavaScript. These functions reduce the need for external libraries and make YQL easier to use.

### Example:

```
var md5string = y.crypto.encodeMd5("encode this string to md5");
```

Function	Description	Returns
encodeHmacSHA256(String secret, String plaintext)	Encrypts a string using HMAC-SHA256 encryption.	Returns an encrypted string.
encodeHmacSHA1(String secret, String plaintext)	Encrypts a string using HMAC-SHA1 encryption.	Returns an encrypted string.
encodeMd5(String plaintext)	Provides the MD5 hash of a string.	Returns an MD5 hash.
encodeSha(String plaintext)	Provides the SHA-1 hash of a string.	Returns an SHA-1 hash.
encodeBase64(String plaintext)	Performs Base64 encoding of a string.	Returns an Base64 encoded string.
decodeBase64(String plaintext)	Performs Base64 decoding of a string.	Returns an Base64 decoded string.
uuid()	Provides a cryptographically secure version 4 Universal Unique Identifier (UUID).	Returns a UUID.

## y.date

Using `y.date` gives you access to date-related functions.

### Methods

#### **y.date.getOffsetFromEpochInMillis(time\_format)**

Parses the given date string in a format specified by [RFC 3339](http://www.ietf.org/rfc/rfc3339.txt)<sup>2</sup> and returns the number of milliseconds since the Unix epoch (0:00:00 UTC on January 1, 1970) as a string.

#### **Parameters:**

- `time_format <String>` A string in the form of the time/date formats or regex patterns below. The regex patterns are case insensitive.
  - [RFC 3339 Timestamp](http://www.ietf.org/rfc/rfc3339.txt)<sup>3</sup>
  - `^( [+ - ] ) ? \s * ( \d + ) \s * ( year | month | week | day | hour | minute ) s ? $`
  - `^( \d + ) \s * ( year | month | week | day | hour | minute ) s ? \s + ( from \s now | after | after \s + today | later | old | ago ) $`
  - `^now \s + ( [ + - ] ) \s * ( \d + ) \s + ( year | month | week | day | hour | minute ) s ? $`
  - `^last \s + ( year | month | week ) $`
  - `now`
  - `yesterday`
  - `tomorrow`

**Returns:** String

#### **Examples:**

In the example table below, `getOffsetFromEpochInMills` is used in conjunction with the JavaScript Date object to log several different times. The table then returns the execution time of the JavaScript in the response.

```
...
<execute><![CDATA[
    var start_time = parseInt(y.date.getOffsetFromEpochInMillis("now"));

    var start_date = new Date(start_time);
    var yesterday_time =
parseInt(y.date.getOffsetFromEpochInMillis("yesterday"));
    var yesterday_date = new Date(yesterday_time);
    var month_from_now = new
Date(parseInt(y.date.getOffsetFromEpochInMillis("now + 1 month")));
    var ali_liston_fight = new
Date(parseInt(y.date.getOffsetFromEpochInMillis("1964-02-25")));
    y.log("Start Time: " + start_date.toString());
]
```

---

<sup>2</sup> <http://www.ietf.org/rfc/rfc3339.txt>

<sup>3</sup> <http://www.ietf.org/rfc/rfc3339.txt>

```
y.log("Yesterday: " + yesterday_date.toDateString());
y.log("Last Month: " + month_from_now.toDateString());
y.log("Ali beats Liston: " + ali_liston_fight.toDateString());
response.object = "Execution Time: " +
String((parseInt(y.date.getOffsetFromEpochInMillis("now")) -
parseInt(start_time))/1000) + " seconds.";
]]>
</execute>
...
```

In this example XML response returned by the above table, the diagnostic information includes the logged times, and the result element contains the execution time.

```
<query xmlns:yahoo="http://www.yahooapis.com/v1/base.rng"
yahoo:count="1" yahoo:created="2011-07-12T02:20:59Z" yahoo:lang="en-US">

  <diagnostics>
    <publiclyCallable>true</publiclyCallable>
    <url execution-time="76">
      <![CDATA[http://zhouyaoji.com/yql/date.xml]]>
    </url>
    <log>Start Time: Tue Jul 12 2011</log>
    <log>Yesterday: Mon Jul 11 2011</log>
    <log>Last Month: Fri Aug 12 2011</log>
    <log>Ali beats Liston: Tue Feb 25 1964</log>
    <javascript execution-time="6" instructions-used="12134"
table-name="sc"/>
    <user-time>89</user-time>
    <service-time>76</service-time>
    <build-version>19521</build-version>
  </diagnostics>
  <results>
    <result>Execution Time: 0.004 seconds.</result>
  </results>
</query>
```

## y.decompress(base64\_compressed\_string)

Allows you to decompress query parameters and the HTTP POST request body. The decompress method decodes a base64 string and then uses gunzip to decompress the string.

### Parameters:

- `base64_compressed_string` <String> A string that has been compressed with gzip and then base64 encoded.

**Returns:** String

### Examples:

**YQL Statement:** use 'http://example.com/decompress.xml' as decompress; select \* from decompress where foo="H4sIAOQTlU0AA8tIzcnJBwCGphA2BQAAAA=="

**Example YQL Open Data Table:** <http://example.com/decompress.xml>

The table `decompress.xml` below uses `y.decompress` to decompress the value assigned to `foo` in the YQL statement above and then returns it as the string.

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
securityLevel="any">
  <bindings>
    <select itemPath="" produces="XML">
      <inputs>
        <key id="foo" type="xs:string"
paramType="variable"required="true"/>
      </inputs>
      <execute><![CDATA[
        try {
          y.log(foo);
          var bar = y.decompress(foo);
        } catch (err) {
          y.log(err.message);
        }
        response.object = bar;
      ]]></execute>
    </select>
  </bindings>
</table>
```

#### Returned XML Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<query xmlns:yahoo="http://www.yahooapis.com/v1/base.rng"
yahoo:count="1"yahoo:created="2011-04-01T00:06:38Z" yahoo:lang="en-US">

  <diagnostics>
    <publiclyCallable>true</publiclyCallable>
    <url
execution-time="2"><![CDATA[http://example.com/decompress.xml]]></url>
    <log>H4sIAOQTlU0AA8tIzcnJBwCGphA2BQAAAA==</log>
  </diagnostics>
  <results>
    <result>hello</result>
  </results>
</query>
```

## y.exit(), y.exit(status\_code,msg)

Causes the script to exit. If no parameters are given to `y.exit`, the YQL engine returns the default HTTP response status code and message to the client. The table author can change the HTTP response code sent to the client by passing a status code and a message as parameters to `y.exit`.

#### Parameters:

- `status_code` <Number> - The HTTP response status code that gets returned to the client.
- `msg` <String> - The message that gets returned to the client.

**Returns:** None

**Example:**

```
...
<execute><![CDATA[
  var myRequest = y.rest('http://example.com');
  var data = myRequest.get().response;
  if(!data){
    y.exit(204, "Sorry, no content.");
  } else {
    response.object = data;
  }
]></execute>
...
```

## **y.getTenantContextsAvailable()**

Gets the names of all the tenants with configurations that can be used to create execution contexts for the query, env, and use methods. The returned tenant names will be the same, or super set, of the names returned by `y.getTenantContextsAllowed`.

**Parameters:** None

**Returns:** Object

**Examples:**

The following example table returns all of the available tenant contexts.

```
...
<execute><![CDATA[
  var tenantContextsAvailable = y.getTenantContextsAvailable();
  var results = "";
  var num_contexts = tenantContextsAvailable.length;
  for (var i=0; i<num_contexts; i++) {
    results += tenantContextsAvailable[i];
  }
  response.object = results;
]></execute>
...
```

## **y.parseJson(json\_str)**

Parses a well-formed JSON string and returns the resulting JavaScript object. If given a malformed JSON string, `y.parseJson` will throw a runtime `JSONException`.

**Parameters:**

- `json_str` <String> A well-formed JSON string.

**Returns:** Object

**Examples:**

**YQL Statement:** use 'http://example.com/parseJson.xml' as jp; select \* from jp where mykey="sample\_key"

**Example YQL Open Data Table:** http://example.com/parseJson.xml

The table parseJson.xml below parses a JSON string and returns a JavaScript object.

```
<table
...
  <bindings>
    <select itemPath="" produces="XML">
      <inputs>
        <key id="mykey" type="xs:string" paramType="variable"
required="true"/>
      </inputs>
      <execute><![CDATA[
        var mycontent = '\{
          "firstName": "John",\
          "lastName": "Smith",\
          "address": {\
            "streetAddress": "21 2nd Street",\
            "city": "New York",\
            "state": "NY",\
            "postalCode": "10021"\
          },\
          "phoneNumbers": [\
            {\
              "type": "home",\
              "number": "212 555-1234"\
            },\
            {\
              "type": "fax",\
              "number": "646 555-4567"\
            }\
          ]\
        }';
        y.log(mycontent);
        j= y.parseJson(mycontent);
        y.log(j);
        response.object=j.get('lastName');
      ]]></execute>
    </select>
  </bindings>
</table>
```

**Example XML Response:**

```
<query xmlns:yahoo="http://www.yahooapis.com/v1/base.rng"
yahoo:count="0" yahoo:created="2011-06-15T23:23:21Z" yahoo:lang="en-US">

  <diagnostics>
```



```
<publiclyCallable>true</publiclyCallable>
<url execution-time="2"
proxy="DEFAULT"><![CDATA[http://example.com/parseJson.xml]]></url>
<log>{
  "firstName": "John",
  "lastName": "Smith",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumbers":
  [
    {
      "type": "home", "number": "212 555-1234"
    },
    {
      "type": "fax", "number": "646 555-4567"
    }
  ]
}</log>
<log>{
  "firstName": "John",
  "lastName": "Smith",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumbers":
  [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ]
}</log>
<javascript execution-time="2" instructions-used="0"
table-name="xxx"/>
<user-time>8</user-time>
<service-time>2</service-time>
<build-version>18749</build-version>
</diagnostics>
<results>
  <result>Smith</result>
</results>
</query>
```

## y.query(statement, params, timeout, callback)

Perhaps you want to use YQL queries while still using JavaScript within YQL. `y.query` allows you to perform additional YQL queries within the execute sub-element.

### Parameters:

- `statement` <String> A YQL statement.
- `params` <Object> The object can contain key-values that can be referenced in the YQL statement for variable substitution. In the example below, the `@url` variable in the passed YQL statement is replaced with the value associated with the `url` key of the `params` object.

```
var q = y.query('select * from html where url=@url and
xpath="//div[@id=\'yfi_headlines\']/div[2]/ul/li/a"',
{url: 'http://finance.yahoo.com/q?s=yhoo'});
var results = q.results;
```

- `timeout` <Number> The number of milliseconds before the call times out.
- `callback` <Function> The callback function to handle the returned response. See [Making Asynchronous Calls with JavaScript Execute \[44\]](#) for more information about using the callback.

**Returns:** Object

The returned object contains the following properties:

Property	Description	Data Type
<code>results</code>	The results.	E4X object
<code>diagnostics</code>	The diagnostics.	E4X object
<code>query</code>	The YQL statement that was passed as a parameter to <code>y.query</code> .	string
<code>timeout</code>	Specifies the request timeout in milliseconds. This is useful when you want to cancel requests that take longer than expected.	None

### Example:

In the following code snippet, `y.query` executes a statement that uses the `html` table to get Yahoo! financial data.

```
var q = y.query('select * from html where
url="http://finance.yahoo.com/q?s=yhoo" and
xpath="//div[@id=\'yfi_headlines\']/div[2]/ul/li/a"');
var results = q.results;
```

## y.rest(url, callback)

The `y.rest` method allows you to make GET requests to remote Web services. It also allows you to pass parameters and headers in your request and use a callback function to make asynchronous requests. See [Making Asynchronous Calls with JavaScript Execute \[44\]](#) for more information about using the callback.

### Parameters:

- `url` <String> The URL endpoint to a query.

- `callback <Function>` (Optional) The callback mechanism waits to receive returned results that it can then process. This callback function allows asynchronous calls because the function can wait for the returned results while `y.rest` is called again.

**Returns:** [rest Object \[31\]](#)

### Examples:

In the following code snippet, an HTTP GET request is made to `example.com` and the response is saved to `data`.

```
var myRequest = y.rest('http://example.com');
var data = myRequest.get().response;
```

The two properties `url` and `timeout` of the response help you determine if a call has timed out or associate the response with the original URL. The code snippet below shows you the `url` and `timeout` properties and how you could potentially use them:

```
// The 'url' property allows you to connect
// the original request url to the response
var uri = y.rest("http://www.yahoo.com").get().url

// The 'timeout' property allows you to make another call
// or take other action if your original call times out
var timed_out = y.rest('http://www.yahoo.com').get().timeout

if(timed_out) {
  // Try one more time
  var resp = y.rest(uri).get().response;
}
```



### Tip

The `y.rest` method supports "chaining", which means that you can construct and run an entire REST request by creating a "chain" of methods. Here is a hypothetical example:

```
var myData = y.rest('http://blah.com')
  .path("one")
  .path("two").query("a","b")
  .header("X-TEST","value")
  .get().response;
```

When chained, the resulting request looks like this:

```
http://blah.com/one/two?a=b
```



### Note

Because JSON does not have a "root" node in most cases, all JSON responses from a remote Web service will be contained within a special `json` root object under `response.results`.

## y.sync()

Causes the JavaScript execution to wait until pending `y.rest` and `y.query` asynchronous requests have been completed. If `y.sync` is not called after making asynchronous `y.rest` and `y.query` requests, the JavaScript execution may complete before the asynchronous requests return. If no asynchronous requests are pending, calling `y.sync` will not have any effect.

**Parameters:** None

**Returns:** N/A

**Examples:**

See [Making Asynchronous Calls with JavaScript Execute \[44\]](#) for examples.

## request Global Object

The `request` global object is essentially a reference to a [rest object \[31\]](#). You must use the `request` object, `y.rest`, or `y.query` to access the `rest` object.

## rest Object

The `rest` object has properties and methods for getting information and making REST calls.

### Properties

Property	Description	Data Type
<code>headers</code>	Gets the hashmap of headers	object
<code>url</code>	Provides a URL endpoint to query	string
<code>queryParams</code>	Gets the hashmap of query parameters	E4X object containing query parameters.
<code>matrixParams</code>	Gets the hashmap of matrix parameters	<a href="#">result object [38]</a> containing matrix parameters.

### Methods

#### accept(content-type)

Specifies the type of content to send in the response using the Accept HTTP header. This tells YQL what kind of data format you want returned, as well as how to parse it.

**Parameters:**

- `content-type <String>` The content type of the data being sent. For example: `application/xml`.

**Returns:** [rest Object \[29\]](#)

**Examples:**

The following is an example of how you would specify JSON as the return format for data you send as XML:

```
var ret =  
request.accept('application/json').contentType('application/xml').post(content).response;
```

Using the above example, YQL will convert XML to JSON prior to returning it in the response.

## contentType(content-type)

Specifies the content-type of the data being sent. An example of a content-type is: `application/json`. This object is useful with INSERT and UPDATE statements.



### Note

This method does not automatically convert one data format to another prior to sending it, so if you indicate one format in `contentType` but actually send another, your Web service may produce an error.

### Parameters:

- `content-type <String>` The content-type of the data being sent. An example of a content-type is: `application/json`.

**Returns:** `rest Object`

The following is an example of how you would specify XML as the data you are sending:

```
var content = <employee>  
  <name>John</name>  
  <id>21</id>  
</employee>;  
  
var ret =  
  request.contentType('application/xml').post(content).response;
```

## del()

Performs an HTTP DELETE. This object is useful with DELETE statements.

**Parameters:** None

**Returns:** [result Object \[38\]](#)

### Examples:

In the below Open Data Table, the `del()` method is used to delete the Mixi voice status identified by the post ID. Using `del()` sends an HTTP DELETE request to the URL defined in the `url` element.

```
<delete itemPath="" produces="JSON">  
  <urls>  
    <url>http://api.mixi-platform.com/2/voice/statuses/{post_id}</url>  
  </urls>  
  <inputs>  
    <key id="oauth_token" type="xs:string" paramType="query"  
required="true" />  
    <key id="post_id" type="xs:string" paramType="path" required="true"  
>  
  </inputs><execute><![CDATA[response.object =
```

```
request.del().response;]]></execute>  
</delete>
```

## fallbackCharset(charset\_list)

Overrides the list of fallback character sets, which is set to "utf-8, iso-8859-1" by default, for decoding the returned response. YQL attempts to decode the response using the character sets listed in `charset_list` when the response either does not specify the character set or specifies an incorrect character set that results in a failed decoding.

### Parameters:

- `charset_list` <String> A list of one or more fallback character sets. For example: "ISO-8859-1, utf-8"

**Response:** [rest Object \[31\]](#)

### Examples:

```
response.object = y.rest(url).fallbackCharset('shift_jis,iso-8859-8,  
UTF-8, iso-8859-1').get().response;
```

## forceCharset(charset\_list)

Forces YQL to decode the response using the first successful character set listed in `charset_list`. Using this method overrides both the character set specified by the response and the fallback character sets.

### Parameters:

- `charset_list` <String> A list of one or more character sets that YQL will use to decode the response. For example: "ISO-8859-1, utf-8"

**Returns:** [rest Object \[29\]](#)

### Examples:

```
y.rest('www.uol.com.br').forceCharset('iso-8859-1').get().response;
```

## get()

Performs a GET request to the URL endpoint. This object is useful with SELECT statements.

**Parameters:** None

**Returns:** [result Object \[38\]](#)

### Examples:

```
r = y.rest("yahoo.com").contentType('application/json').get();  
response.object = r;
```

## head()

Performs an HTTP HEAD request on a URL.

**Parameters:** None

**Returns:** Object

**Examples:**

The head method in this example table is used to make an HTTP HEAD request. The HTTP header fields in the returned object are accessed with the `headers` property.

```
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where url_path='www.example.com'
    and header='true';</sampleQuery>    </meta>
    <bindings>
      <select itemPath="" produces="XML">
        <inputs>
          <key id='url_path' type='xs:string' paramType='variable'
required="true" />
          <key id='header' type='xs:boolean' paramType='variable' />
        </inputs>
        <execute><![CDATA[
          var res = {};
          if(header){
            res.body = y.rest(url_path).head().headers;
          }else{
            res.body= y.rest(url_path).get().response;
          }
          response.object=res.body;
        ]]></execute>
      </select>
    </bindings>
  </table>
```

## header(name, value)

Adds an HTTP header to the request.

**Parameters:**

- `name` <String> The name of the HTTP header.
- `value` <String> The value associated with the HTTP header.

**Returns:** [rest Object \[31\]](#)

**Examples:**

The header method in this code snippet is used to pass authorization credentials to get a response.

```
y.include("http://yqlblog.net/samples/base64.js");
var authheader = "Basic "+Base64.encode(username+": "+password);
response.object =
request.header("Authorization",authheader).get().response;
```

In this code snippet, the header method sets the HTTP header Content-Encoding to gzip, so that the returned response will be compressed.

```
var resp =  
y.rest("http://www.apache.org").header("Accept-Encoding", "gzip").get();  
  
response.object = resp.response;
```

## jsonCompat(mode)

Allows you to get "lossless" JSON when making a REST call to a Web service. For more information about "lossy" JSON, see [JSON-to-JSON Transformation \[40\]](#) and [Preventing Lossy JSON Results](#).

### Parameters:

- mode <String> Currently, the only allowed value is "new". In the future, other modes of lossless JSON compatibility might exist that require passing a different string to jsonCompat.

**Returns:** [rest Object \[31\]](#)

### Examples:

The table below makes two GET calls to the Yahoo! Finance API. The first call returns **lossy** JSON, and the second call returns **lossless** JSON. You can view the difference in the returned JSON in the diagnostic logs.

```
<?xml version="1.0" encoding="UTF-8"?><table  
xmlns="http://query.yahooapis.com/v1/schema/table.xsd"  
securityLevel="any">  
  <bindings>  
    <select produces="JSON">  
      <execute><![CDATA[  
        var url =  
'http://api.finance.yahoo.com:4080/v1/profile/symbol/YHOO?lang=en-US&format=json';  
  
        var a = y.rest(url).get().response;  
        var b = y.rest(url).jsonCompat('new').get().response;  
        y.log(y.jsToString(y.xmlToJson(a)));  
        y.log(y.jsToString(y.xmlToJson(b)));  
        response.object = b;  
      ]]></execute>  
    </select>  
  </bindings>  
</table>
```

Note that to obtain lossless JSON when querying the above table, the client must append the query string parameter jsonCompat=new to the YQL URI as shown here:

```
http://query.yahooapis.com/v1/public/yql?q=use 'http://example.com/json-  
Compat.xml as json.compat; select * from json.compat&format=json&json-  
Compat=new
```

## matrix(name,value)

Adds a matrix parameter to the request.

### Parameters:



- name <String>
- value <String>

**Returns:** [rest Object \[31\]](#)

**Examples:**

The code snippet below posts the matrix parameters to `flower_order_server.com`.

```
y.rest("http://flower_order_service.com").matrix('flower', 'roses').matrix('color', 'red').post();
```

Matrix parameters are attached to the URI resource like query parameters, but are delimited by a semicolon and not an ampersand and the matrix parameters are not separated from the URI resource by a question mark. The `yql.rest` call creates the following URL: `http://flower_order_service.com;flower=roses;color=red`

## path(path\_segment)

Appends a path segment to the URI.

**Parameters:**

- path\_segment <String> The path segment to add to a URI.

**Returns:** [request Object \[31\]](#)

**Examples:**

```
var myData = y.rest('http://blah.com')
    .path("one")
    .path("two") .query("a", "b")
    .header("X-TEST", "value")
    .get().response;
```

## post(content)

Performs an HTTP POST, using the value of the content, to the URL endpoint. This object is useful with INSERT, UPDATE, and DELETE statements.

**Parameters:**

- content <String|Object> The data that is posted.

**Returns:** [result Object \[38\]](#)

**Examples:**

```
var content = { "employee": { "name": "John", "id": 21 } };
var ret = request.contentType('application/json').post(content).response;
```

## put(content)

Performs an HTTP PUT, using the value of the content, to the URL endpoint. This object is useful with INSERT, UPDATE, and DELETE statements.

**Parameters:**

- `content` <String | Object> The data that is used in a PUT request.

**Returns:** [result Object \[38\]](#)

**Examples:**

```
url = blogurl + "/xmlrpc.php";
myRequest = y.rest(url);
myRequest.contentType("text/xml");
results = myRequest.put('<?xml version="1.0" encoding="UTF-8"?>' +
postData.toString()).response;
```

## query(key, value)

Adds a single query parameter.

**Parameters:**

- `key` <String>
- `value` <String>

**Returns:** [request Object \[31\]](#)

**Examples:**

```
var formUrl = "http://example.com/form";
var resp = y.rest(formUrl).query("name", "Tom").post().response;
```

## query(hashmap)

Adds all the query parameters based on key-name hashmap.

**Parameters:**

- `hashmap` <Object>

**Returns:** [request Object \[31\]](#)

**Examples:**

```
var formUrl = "http://example.com/form";
var name = { "name": "John" };
var resp = y.rest(formUrl).query(name).post().response;
```

## timeout(milli\_seconds)

Specifies the request timeout in milliseconds. This is useful when you want to cancel requests that take longer than expected.

**Parameters:**

- `milli_seconds` <Number> The number of seconds before your request times out.

**Returns:** N/A

**Examples:**

```
// timeout after 500 milliseconds
y.rest('http://....').timeout(500).get();
```

## response Global Object

The response global object allows you to determine how responses are handled and is also used to set the value to the data you'd like to return, as an E4X object, a JSON structure, or simply a string.

Object	Description
object	Contains the results of your execute script. Set this value to the data you'd like to return, as an E4X object, a JSON structure, or simply a string.

## result Object


The result object contains the results of your execute script.

The table below lists the properties of the result object.

Property	Description	Data Type
<a href="#">response [38]</a>	Get the response from the remote service. If the response content type is not application/json or text/xml then YQL provides a string. If JSON or XML is specified, the E4X representation of the data is returned.	E4X object or string
headers	The headers returned from the response.	object
status	The HTTP status code.	string
timeout	Indicates if the call timed out.	boolean
url	The URL used to make the request.	string

## Global Variables

The following global variables are available for use within the execute element of YQL Open Data Tables:

Variable	Description
input	<p>This global variable is available for each binding within the inputs element such as key, value, or map. For example, to call the first binding below you would use nameofid.</p> <pre>var mycontent = nameofid;</pre> <div>  <p><b>Important</b></p> <p>If the id name uses an illegal identifier, such as the use of hyphens, you must instead use the inputs global variable.</p> </div> <pre>&lt;inputs&gt;   &lt;key id="nameofid" type="xs:string"     paramType="path" default="my default key" required="true"   /&gt;   &lt;value id="field-name" type="xs:string" paramType="variable"</pre>

Variable	Description
	<pre>default="my default field" /&gt; &lt;/inputs&gt;</pre> <p>When a map binding is present in the Open Data Table, the global variable is present as a named hashtable. Each value provided in the YQL statement is set in the hashmap.</p>
inputs	<p>This global variable is an array that contains each binding within the inputs element, along with its value. For example, to call the second binding above, you would use <code>inputs[field-name]</code>:</p> <pre>var mycontent =inputs['field-name'];</pre>

## JavaScript and E4X Best Practices for YQL

The following is a series of best practices related to using JavaScript and E4X within the `execute` sub-element in YQL:

- [Paging Results \[39\]](#)
- [Including Useful JavaScript Libraries \[40\]](#)
- [Using E4X within YQL \[40\]](#)
- [Logging and Debugging \[42\]](#)

## Paging Results

YQL handles paging of returned data differently depending on how you control paging within an Open Data Table definition. Let us consider the following example, followed by three paging element scenarios:

```
select * from table(10,100) where local.filter>4.0
```

- **No page element:** If no paging element is provided, YQL assumes you want all data available to be returned at once. Any "remote" paging information provided on the `select` (10 being the offset and 100 being the count in our example), will be applied to **all** of the results before being processed by the remainder of the `where` clause. In our example above, the first 10 items will be discarded and only another 100 will be used, and `execute` will only be called once.
- **A paging element that only supports a variable number of results:** If a paging element is provided that only supports a variable number of results (a single page with variable count), then the `execute` sub-element will only be called once, with the total number of elements needed in the variable representing the count. The offset will always be 0. In our example, the count will be 110, and the offset 0.
- **A paging element that supports both offset and count:** If a paging element is provided that supports both offset and count, then the `execute` sub-element will be called for each "page" until it returns fewer results than the paging size. In this case, let's assume the paging size is 10. The `execute` sub-element will be called up to 10 times, and expected to return 10 items each time. If fewer results are returned, paging will stop.



### Note

In most cases, paging within the Open Data Table should match the paging capabilities of the underlying data source that the table is using. However, if the `execute` sub-element

is adjusting the number of results coming back from a fully paging Web service or source, then there is usually no way to unify the "offset" of the page as set up in the Open Data Table with the destinations "offset". You may need to declare your Open Data Table as only supporting a variable number of results in this situation.

## Including Useful JavaScript Libraries

When writing your `execute` code, you may find the following JavaScript libraries useful:

### OAuth:

```
y.include("http://oauth.googlecode.com/svn/code/javascript/oauth.js");  
y.include("http://oauth.googlecode.com/svn/code/javascript/sha1.js");
```

### Flickr:

```
y.include("http://blog.pipes.yahoo.net/wp-content/uploads/flickr.js");
```

### MD5, SHA1, Base64, and other [Utility Functions](#)<sup>4</sup>:

```
y.include("http://v8cgi.googlecode.com/svn/trunk/lib/util.js");
```

## Using E4X within YQL

ECMAScript for XML (simply referred to as E4X) is a standard extension to JavaScript that provides native XML support. Here are some benefits to using E4X versus other formats, such as JSON:

- Preserves all of the information in an XML document, such as namespaces and interleaved text elements. Since most web services return XML this is optimal.
- You can use E4X selectors and filters to find and extract parts of XML structure.
- The engine on which YQL is created natively supports E4X, allowing E4X-based data manipulation to be faster.
- Supports XML literals, namespaces, and qualified names.

To learn more about E4X, refer to these sources online:

- [E4X Quickstart Guide](#)<sup>5</sup> from WS02 Oxygen Tank
- [Processing XML with E4X](#)<sup>6</sup> from Mozilla
- [AJAX and scripting Web service with E4X](#)<sup>7</sup> by IBM
- [Introducing E4X](#)<sup>8</sup> by O'Reilly
- [Popular E4X Bookmarks](#)<sup>9</sup> by delicious

---

<sup>4</sup> [http://code.google.com/p/v8cgi/wiki/API\\_Util](http://code.google.com/p/v8cgi/wiki/API_Util)

<sup>5</sup> <http://wso2.org/project/mashup/0.2/docs/e4xquickstart.html>

<sup>6</sup> [https://developer.mozilla.org/en/Core\\_JavaScript\\_1.5\\_Guide/Processing\\_XML\\_with\\_E4X](https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Processing_XML_with_E4X)

<sup>7</sup> <http://www.ibm.com/developerworks/webservices/library/ws-ajax1/>

<sup>8</sup> <http://www.xml.com/pub/a/2007/11/28/introducing-e4x.html>

<sup>9</sup> <http://delicious.com/popular/e4x>

- [E4X guide<sup>10</sup>](http://rephrase.net) by rephrase.net

## E4X Techniques

In addition to the resources above, the following tables provides a quick list of tips related to using E4X:

E4X Technique	Notes	Code Example
<b>Creating XML literals</b>	-	<code>var xml = &lt;root&gt;hello&lt;/root&gt;;</code>
<b>Substituting variables</b>	Use curly brackets {} to substitute variables. You can use this for E4X XML literals as well.	<code>var x = "text"; var y = &lt;item&gt;{x}&lt;/item&gt;;</code>
<b>Adding sub-elements to an element</b>	When adding sub-elements to an element, include the root node for the element.	<code>item.node+=&lt;subel&gt;&lt;/subel&gt;;</code>
	You can add a sub-element to a node in a manner similar to adding sub-elements to an element.	<code>x.node += &lt;sub&gt;&lt;/sub&gt;;</code>  This above code results in the following structure:  <code>&lt;node&gt;&lt;sub&gt;&lt;/sub&gt;&lt;/node&gt;</code>
	If you try to add a sub-element to a node without including the root node, you will simply append the element and create an XML list.	<code>x += &lt;sub&gt;&lt;/sub&gt;;</code>  The above code results in the following structure:  <code>&lt;node&gt;&lt;node&gt;&lt;sub&gt;&lt;/sub&gt;;</code>
<b>Assigning variably named elements</b>	Use substitution in order to create an element from a variable.	<code>var item = &lt;{name}&gt;;</code>
<b>Assigning a value to an attribute</b>	-	<code>item.@["id"]=path[0];</code>
<b>Getting all the elements within a given element</b>	-	<code>var hs2 = el..*;</code>
<b>Getting specific objects within an object anywhere under a node</b>	-	<code>var hs2 = el..div</code>
<b>Getting the immediate H3 children of an element</b>	-	<code>h2 = el.h3;</code>
<b>Getting an attribute of an element</b>	-	<code>h3 = el.h3.@id;</code>  or <code>h3 = el.h3.@["id"];</code>
<b>Getting elements with a certain attribute</b>	-	<code>var alltn15divs = d..div.(@['id'] == "tn15con- tent");</code>
<b>Getting the "class" attribute</b>	Use brackets to surround the "class" attribute.	<code>className =t.@[ 'class' ];</code>

<sup>10</sup> <http://rephrase.net/days/07/06/e4x>

E4X Technique	Notes	Code Example
<b>Getting a class as a string</b>	To get a class as a string, get its text object and the apply toString.	<code>var classString = className.text().toString()</code>
<b>Getting the name of a node</b>	Use <code>localName()</code> to get the name of a node.	<code>var nodeName = e4xnode.localName();</code>



### Note

When using E4X, note that you can use XML literals to insert XML "in-line," which means, among other things, you do not need to use quotation marks:

```
var myXml = <foo />;
```

## E4X and Namespaces

When working with E4X, you should know that E4X objects are namespace aware. This means that you must specify the namespace before you work with E4X objects within that namespace. The following example sets the default namespace:

```
default xml namespace = 'http://www.inktomi.com/';
```

After you specify a default namespace, all new XML objects will inherit that namespace unless you specify another namespace.



### Caution

If you do not specify the namespace, elements will seem to be unavailable within the object as they reside in a different namespace.



### Tip

To clear a namespace, simply specify a blank namespace:

```
default xml namespace = '';
```

## JavaScript Logging and Debugging

To get a better understanding of how your executions are behaving, you can log diagnostic and debugging information using the `y.log` statement along with the `y.getDiagnostics` element to keep track of things such as syntax errors or uncaught exceptions.

The following example logs "hello" along with a variable:

```
y.log("hello");
```

```
y.log(somevariable);
```

Using `y.log` allows you to get a "dump" of data as it stands so that you can ensure, for example, that the right URLs are being created or responses returned.

The output of `y.log` goes into the YQL diagnostics element when the table is used in a select.

You can also use the follow JavaScript to get the diagnostics that have been created so far:

```
var e4xObject = y.getDiagnostics();
```

## Executing JavaScript Globally

When the `execute` element is placed outside of the binding element, it is available globally within an Open Data Table and is usable across each of the table's bindings. For example, in the following Open Data Table, the function `arrayConcat` is available to be used within the `execute` element for `SELECT` statements:

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
securityLevel="any">
  <meta>
    <author>liang</author>
    <description>execute block outside bindings</description>
    <documentationURL/>
  </meta>

  <execute><![CDATA[
    function arrayConcat(parents, children) {
      var family=parents.concat(children);
      return family;

    }
  ]]></execute>

  <bindings>

    <select itemPath="" produces="XML">
      <urls>
        <url env="all">http://fake.com</url>
      </urls>

      <inputs>
        <key id="mykey" type="xs:string" paramType="variable"
required="true" />
      </inputs>

      <execute><![CDATA[

        myparents=["Sister", "Brother"];
        mychildren=["Dad", "Mom"];
        var myfamily = arrayConcat(myparents,mychildren);
        response.object = <family>{myfamily}</family>;

      ]]></execute>

    </select>
  </bindings>
</table>
```



## Making Asynchronous Calls with JavaScript Execute

The `y.rest` and `y.query` functions take callback functions as a parameter. The callback function is called either when `y.rest` or `y.query` completes a call or errors out. Note that callback functions are currently not called if timeout occurs. This callback mechanism allows for simultaneously handling and processing multiple REST calls.

The callback functions are just JavaScript functions that are passed the results of the REST call. When the callback function below is passed as a parameter to `y.rest`, the response is assigned to `result`.

The code example shows the basic syntax of using `y.rest` with a callback function. You can see how the callback can be implemented in Examples.

```
function callback(result) {  
    // Do something with 'result'.  
}  
y.rest("http://example.com", callback);  
  
// Wait for the callback(s) to complete.  
y.sync();
```

## Examples

### `y.rest`

A single callback function can be used for multiple requests. The following example uses the same callback to log the URL and status of the response as well as the number of responses received.

```
// The callback function logs the order in which  
// URL requests are completed.  
var mycallback = function(result) {  
    count++;  
    y.log("response received for url: " + result.url + ", Status: " +  
result.status + ", total response count: " + count);  
    if (first == null) {  
        first = result.response;  
    }  
}  
// Create four asynchronous 'y.rest' requests.  
y.rest("http://www.yahoo.com", mycallback).get();  
y.rest("http://finance.yahoo.com", mycallback).get();  
y.rest("http://movies.yahoo.com", mycallback).get();  
y.rest("http://news.yahoo.com", mycallback).get();  
  
// Wait for callbacks to complete.  
y.sync();
```

### `y.query`

The syntax and usage for callbacks with `y.query` is nearly identical to that of `y.rest`. The callback function is passed as a parameter to `y.query`.

The code example below uses the callback to keep track of the successful calls, logs the results, and saves the first returned response.

```
// The callback function logs the order in
// which the URL requests are completed.
var mycallback = function(result) {
    count++;
    y.log("Results received for query #" + count.toString() + ": "
+result.query);
    if (first == null) {
        first = result.response;
    }
}
// Create 4 asynchronous y.rest requests.
y.query("select * from flickr.photos.search where has_geo='true' and
text='san francisco'", mycallback);
y.query("select * from feednormalizer where
url='http://rss.news.yahoo.com/rss/topstories' and output='atom_1.0'",
mycallback);
y.query("select * from weather.forecast where location=90210",
mycallback);
y.query("select * from answers.search where query='cars' and
category_id=2115500137 and type='resolved'", mycallback);

// Wait for the callbacks to be completed.
y.sync();
```

## Examples of Open Data Tables with JavaScript

The following Open Data Tables provide a few examples of YQL's abilities:

- [Hello World Table \[45\]](#)
- [Yahoo! Messenger Status \[46\]](#)
- [OAuth Signed Request to Netflix \[47\]](#)
- [Request for a Flickr frob \[48\]](#)
- [Celebrity Birthday Search using IMDB \[49\]](#)
- [Share Yahoo! Applications \[53\]](#)
- [CSS Selector for HTML \[55\]](#)

### Hello World Table

The following Open Data Table allows you to search a fictional table in which "a" is the path and "b" is the term.

This table showcases the following:

- use of E4X to form the response

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
```

```
<sampleQuery>select * from {table} where a='cat' and
b='dog';</sampleQuery>
</meta>
<bindings>
  <select itemPath="" produces="XML">
    <urls>
      <url>http://fake.url/{a}</url>
    </urls>
    <inputs>
      <key id='a' type='xs:string' paramType='path' required="true"
/>
      <key id='b' type='xs:string' paramType='variable' required="true"
/>
    </inputs>
    <execute><![CDATA[
      // Your javascript goes here. We will run it on our servers
      response.object = <item>
        <url>{request.url}</url>
        <a>{a}</a>
        <b>{b}</b>
      </item>;
    ]]></execute>
  </select>
</bindings>
</table>
```

[Run this example in the YQL console.](#)<sup>11</sup>

## Yahoo! Messenger Status

The following Open Data Table allows you to see the status of a Yahoo! Messenger user.

The table showcases the following:

- use of JavaScript to check Yahoo! Messenger status
- use of E4X to form the response

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where u='sample_id';</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="" produces="XML">
      <urls>
        <url>http://opi.yahoo.com/online?m=t</url>
      </urls>
      <inputs>
        <key id='u' type='xs:string' paramType='query' required="true"
/>
      </inputs>
    </select>
  </bindings>
</table>
```

---

<sup>11</sup> <http://bit.ly/eAvgR>

```

</inputs>
<execute><![CDATA[

    //get plain text back from OPI endpoint
    rawStatus = request.get().response;

    //check if users is not offline
    if (!rawStatus.match("NOT ONLINE")) {
        status = "online";
    } else {
        status = "offline";
    }

    //return results as XML using e4x
    response.object =
    <messengerstatus>
        <yahoo_id>{u}</yahoo_id>
        <status>{status}</status>
    </messengerstatus>;
    ]]></execute>
</select>
</bindings>
</table>

```

[Run this example in the YQL console.](#)<sup>12</sup>

## OAuth Signed Request to Netflix

The following Open Data Table allows you to make a two-legged OAuth signed request to Netflix. It performs a [search on the Netflix catalog for specific titles](#)<sup>13</sup>.

This table showcases the following:

- access an authenticating API that requires signatures
- use an external JavaScript library

```

<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
  https="true">
  <meta>
    <author>Paul Donnelly</author>

    <documentationURL>http://developer.netflix.com/docs/REST_API_Reference#0_52696</documentationURL>

  </meta>
  <bindings>
    <select itemPath="" produces="XML" >
      <urls>
        <url env="all">http://api.netflix.com/catalog/titles/</url>

```

<sup>12</sup> [http://developer.yahoo.com/yql/console/?q=use%20%22http%3A%2F%2Fkid666.com%2Fyql%2Fymsg\\_opi.xml%22%20as%20ymsg.status%3B%20select%20\\*%20from%20ymsg.status%20where%20u%20%3D%20%22sample\\_id%22](http://developer.yahoo.com/yql/console/?q=use%20%22http%3A%2F%2Fkid666.com%2Fyql%2Fymsg_opi.xml%22%20as%20ymsg.status%3B%20select%20*%20from%20ymsg.status%20where%20u%20%3D%20%22sample_id%22)

<sup>13</sup> [http://developer.netflix.com/docs/REST\\_API\\_Reference#0\\_52696](http://developer.netflix.com/docs/REST_API_Reference#0_52696)

```

        </urls>
        <paging model="offset">
          <start id="start_index" default="0" />
          <pagesize id="max_results" max="100" />
          <total default="10" />
        </paging>
        <inputs>
          <key id="term" type="xs:string" paramType="query"
required="true" />
          <key id="ck" type="xs:string" paramType="variable"
required="true" />
          <key id="cks" type="xs:string" paramType="variable"
required="true" />
        </inputs>
        <execute><![CDATA[
// Include the OAuth libraries from oauth.net
y.include("http://oauth.googlecode.com/svn/code/javascript/oauth.js");
y.include("http://oauth.googlecode.com/svn/code/javascript/sha1.js");

// Collect all the parameters
var encodedurl = request.url;
var accessor = { consumerSecret: cks, tokenSecret: ""};
var message = { action: encodedurl, method: "GET", parameters:
[["oauth_consumer_key",ck],["oauth_version","1.0"]]};
OAuth.setTimestampAndNonce(message);

// Sign the request
OAuth.SignatureMethod.sign(message, accessor);

try {
  // get the content from service along with the OAuth header, and
  return the result back out
  response.object =
request.contentType('application/xml').header("Authorization",
OAuth.getAuthorizationHeader("netflix.com",
message.parameters)).get().response;
} catch(err) {
  response.object = {'result':'failure', 'error': err};
}

]]></execute>
</select>
</bindings>
</table>

```

[Run this example in the YQL console.](#)<sup>14</sup>

## Request for a Flickr frob

The following Open Data Table example returns the frob, which is analogous to the request token in OAuth.

<sup>14</sup> <http://bit.ly/7yNup>

This table showcases the following:

- access an authenticating API that requires signatures
- use an external JavaScript library
- sign a request, then send the request using y.rest
- require the HTTPS protocol (since private keys are being transmitted)

```
<?xml version="1.0" encoding="UTF-8" ?>
// https="true" ensures that only HTTPS connections are allowed
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
https="true">
  <meta>
    <sampleQuery> select * from {table}</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="rsp" produces="XML">
      <urls>
        <url>http://api.flickr.com/services/rest/</url>
      </urls>
      <inputs>
        <key id='method' type='xs:string' paramType='variable'
const="true" default="flickr.auth.getFrob" />
        <key id='api_key' type='xs:string' paramType='variable'
required="true" />
        <key id='secret' type='xs:string' paramType='variable'
required="true" />
      </inputs>
      <execute><![CDATA[
// Include the flickr signing library
y.include("http://www.yqlblog.net/samples/flickr.js");
// GET the flickr result using a signed url
var fs = new flickrSigner(api_key,secret);
response.object = y.rest(fs.createUrl({method:method,
format:""})).get().response();
]]></execute>
      </select>
    </bindings>
  </table>
```

[Run this example in the YQL console.](#)<sup>15</sup>

## Celebrity Birthday Search using IMDB

The following Open Data Table retrieves information about celebrities whose birthday is today by default, or optionally, on a specific date.

This table showcases the following:

- Creating an API/table from HTML data

---

<sup>15</sup> <http://bit.ly/18jOoM>

- Mixing and matching Web service requests with HTML scraping
- Using E4X for creating new objects, filtering, and searching
- Parallel dispatching of query/REST calls
- Handling page parameters

```
<?xml version="1.0" encoding="UTF-8" ?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery> select * from {table}</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="birthdays.person" produces="XML">
      <urls>
        <url></url>
      </urls>
      <paging model="offset">
        <pagesize id="count" max="300" />
        <total default="10" />
      </paging>
      <inputs>
        <key id='date' type='xs:string' paramType='variable' />
      </inputs>
      <execute><![CDATA[

//object to query imdb to extract bio info for a person
var celebInfo = function(name,url) {
  this.url = url;
  this.name = name;
  var querystring = "select * from html where url = '"+url+"' and
xpath="//div[@id='tn15']\\"";
  this.query = y.query(querystring);
}

//actually extract the info and return an xml object
celebInfo.prototype.getData=function() {
  default xml namespace = '';
  var d = this.query.results;
  var img = d..div.(@["id"]=="tn15lhs").div.a.img;
  var content = d..div.(@['id']=="tn15content");
  var bio = "";
  //this is pretty hacky
  for each (var node in content.p) {
    if (node.text().toString().trim().length>100) {
      bio = node.*;
      break;
    }
  }
  var anchors = content.a;
  var bornInYear = null;
  var bornWhere = null;
  var diedInYear = null;
```

```

var onThisDay = [];
//TODO see if there is a wildcard way of pulling these out using
e4x/xpath
for each (var a in anchors) {
    var href = a.@[ 'href' ].toString();
    if (href.indexOf("/BornInYear")==0) {
        bornInYear = a.toString().trim();
        continue;
    }
    if (href.indexOf("/DiedInYear")==0) {
        diedInYear = a.toString().trim();
        continue;
    }
    if (href.indexOf("/BornWhere")==0) {
        bornWhere = a.toString().trim();
        continue;
    }
    if (href.indexOf("/OnThisDay")==0) {
        onThisDay.push(a.text().toString().trim());
        continue;
    }
}
var bornDayMonth=null;
var diedDayMonth=null;
if (onThisDay.length>0) {
    bornDayMonth =
onThisDay[0].replace(/^\s*(\d{1,2})[\s]+(\w+)\s*/,'$1 $2'); //tidy up
whitespace around text
    if (diedInYear && onThisDay.length>1) {
        diedDayMonth=
onThisDay[1].replace(/^\s*(\d{1,2})[\s]+(\w+)\s*/,'$1 $2'); //tidy up
whitespace around text
    }
}
var url = this.url;
var name = this.name;
var bornTime = null;
if (bornDayMonth) {
    var daymonth = bornDayMonth.split(" ");
    bornTime=new
Date(bornInYear,Date.getMonthFromString(daymonth[1]),parseInt(daymonth[0])).getTime()/1000;

}
var diedTime = null;
if (diedDayMonth) {
    var daymonth = diedDayMonth.split(" ");
    diedTime=new
Date(diedInYear,Date.getMonthFromString(daymonth[1]),parseInt(daymonth[0])).getTime()/1000;

}
var person = <person url={url}><name>{name}</name>{img}<born
utime={bornTime}>{bornDayMonth} {bornInYear}</born></person>;
    if (diedTime) person.person+=<died utime={diedTime}>{diedDayMonth}
{diedInYear}</died>;

```



```

    if (bio) person.person+=<bio>{bio}</bio>;
    return person;
}

//general useful routines
String.prototype.trim =function() {
    return this.replace(/^\s*/, '').replace(/\s*$/, '');
}
Date.getMonthFromString = function(month) {
    return {'January':0, 'February':1, 'March':2, 'April':3, 'May':4,
'June':5, 'July':6, 'August':7, 'September':8, 'October':9,
'November':10, 'December':11}[month];
}
Date.prototype.getMonthName = function() {
    return ['January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November',
'December'][this.getMonth()];
}

//the main object that uses boss to get the list (also gets peoples
"death" days too)
celebSearch = function(when,start,count) {
    //search yahoo/boss using the current day and month only on bio pages
    on imdb
    var bornDayMonth = when.getDate()+" "+when.getMonthName();
    var ud = Math.round(when.getTime()/1000);
    var search = 'site:www.imdb.com "Date of birth" "' +bornDayMonth+'"'
    title:biography'
    var query = "select * from search.web("+start+", "+count+") where
query='"+search+"'";
    var celebs = y.query(query).results;

    //go through each result and start to get the persons name and their
    imdb info page out
    var results = [];
    default xml namespace = 'http://www.inktomi.com/'; //make sure our
e4x is in the right namespace. IMPORTANT
    for each (var celeb in celebs.result) {
        //discard any hits on the date of death that also match in our
        yahoo search
        //(this is going to hurt our paging)
        if (celeb["abstract"].toString().indexOf("<b>Date of Birth</b>."
        "<b>" +bornDayMonth)<0) continue;
        var j = celeb.title.toString().indexOf("-"); //use text up to
        "dash" from title for name
        var name = celeb.title.toString().substring(0,j).trim();
        //start parsing these entries by pulling from imdb directly
        results.push(new celebInfo(name,celeb.url));
    }

    //loop through each imdb fetch result, and create the result object
    default xml namespace = '';
    var data = <birthdays utime={ud} date={when} />;
    for each (var celeb in results) {

```

```
        data.birthdays+=celeb.getData();
    }
    return data;
}

//run it for today if no date was provided
var when = new Date();
if (date && date.length>0) {
    when = new Date(date); //TODO needs a well formed date including
    year
}
response.object = new celebSearch(when,0,count);

    ]]></execute>
</select>
</bindings>
</table>
```

[Run this example in the YQL console.](#)<sup>16</sup>

## Shared Yahoo! Applications

The following Open Data Table provides a list of Yahoo! Applications that you and your friends have installed, indicating whether each app is installed exclusively by you, your friends, or both.

This table showcases the following:

- complex E4X usage, including namespaces, filtering, searching, and creation
- authenticated calls to Yahoo! Social APIs using y.query
- setting a security level to user to force authenticated calls only
- optional variable that changes the function (searches on a specific friend)
- handling page parameters

```
<?xml version="1.0" encoding="UTF-8" ?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
securityLevel="user">
  <meta>
    <sampleQuery> select * from {table}</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="root.install.app" produces="XML">
      <urls>
        <url></url>
      </urls>
      <inputs>
        <key id='friendguid' type='xs:string' paramType='variable' />
      </inputs>
      <execute><![CDATA[
function createInstallElement(update,type) {
```

---

<sup>16</sup> <http://bit.ly/fV16L>

```

        var bits = update.itemurl.toString().split("/");
        var appid = bits[bits.length-2].substring(1);//get the appid
from the install url
        var title = update.title.toString();
        default xml namespace = '';
        var el = <app who={type} id={appid}>{title}</app>;
        default xml namespace =
'http://social.yahooapis.com/v1/updates/schema.rng';
        return el;
    }

    default xml namespace = '';
    var root = <install/>;

    //get my friends installs from updates
    var friendapp_installs = null;
    if (friendguid) {
        //only do deltas to this friend
        friendapp_installs = y.query('select title, itemtxt, itemurl
from social.updates(1000) where guid=@guid and type="appInstall" |
unique(field="itemtxt")',{guid:friendguid});
    } else {
        //all friends
        friendapp_installs = y.query('select title, itemtxt, itemurl
from social.updates(1000) where guid in (select guid from
social.connections(0) where owner_guid=me) and type="appInstall" |
unique(field="itemtxt")');
    }
    //get my installs from updates
    var myapp_installs = y.query('select title, itemtxt, itemurl from
social.updates(1000) where guid=me and type="appInstall" |
unique(field="itemtxt")');
    //we're going to keep a collection for each variant of the diff
between my installs and my friend(s)
    var myapp_installs = myapp_installs.results;
    var friendapp_installs = friendapp_installs.results;
    default xml namespace =
'http://social.yahooapis.com/v1/updates/schema.rng';
    for each (var myupdate in myapp_installs.update) {
        y.log("myupdate "+myupdate.localName());
        //use e4x to search for matching node in friendapp with the
same itemtxt (appid)
        var matching =
friendapp_installs.update.(itemtxt==myupdate.itemtxt.toString());
        if (matching.length()>0) {
            //found, we both have it
            root.install+=createInstallElement(myupdate,"shared");
            //y.log("Found "+myupdate.title+" in both");
            myupdate.@matched = true;
            matching.@matched = true;
        } else {
            // not in my friends apps, so add it to
            // me only list
            // y.log("Found "+myupdate.title+" in mine only");

```

```

        root.install+=createInstallElement(myupdate,"me");
        myupdate.@matched = true;
    }
}
//anything left in the friends app list that doesnt have a "match"
attribute is not installed by me
for each (var friendupdate in
friendapp_installs.update.(@matched!=true)) {
    root.install+=createInstallElement(friendupdate,"friend");
}
//return the three sets of results
default xml namespace = '';
response.object = <root>{root}</root>;
]]></execute>
</select>
</bindings>
</table>

```

[Run this example in the YQL console.](#)<sup>17</sup>

## CSS Selector for HTML

The following Open Data Table allows you to filter HTML using CSS selectors.

This table showcases the following:

- importing external JavaScript utility functions
- calling a YQL query within execute

```

<?xml version="1.0" encoding="UTF-8" ?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where url="www.yahoo.com" and
css="#news a"</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="" produces="XML">
      <urls>
        <url></url>
      </urls>
      <inputs>
        <key id="url" type="xs:string" paramType="variable" required="true"
/>
        <key id="css" type="xs:string" paramType="variable" />
      </inputs>
      <execute><![CDATA[
//include css to xpath convert function

y.include("http://james.padolsey.com/scripts/javascript/css2xpath.js");

var query = null;

```

<sup>17</sup> <http://bit.ly/UeIuq>

```

if (css) {
  var xpath = CSS2XPATh(css);
  y.log("xpath "+xpath);
  query = y.query("select * from html where url=@url and
xpath=\""+xpath+"\"",{url:url});
} else {
  query = y.query("select * from html where url=@url",{url:url});
}
response.object = query.results;
]]</execute>
</select>
</bindings>
</table>

```

[Run this example in the YQL console.](#)<sup>18</sup>

## Execution Rate Limits

The following rate limits apply to executions within Open Data Tables:

Item	Limit
Total Units of Execution	50 million
Total Time for Execution	30 seconds
Total Stack Depth	100 levels
Total Number of Concurrent YQL Queries	5 concurrent queries
Total Number of Objects Created via new	1 million objects
Total Number of Elements per E4X Object	1 million elements per E4X object

### What is a unit of execution?

A unit can be any usage of memory or instruction. For example, if a specific operation is only used twice within an execute script, that would sum up to 2 units:

$f(\text{units}) = f(\text{operation1}) + f(\text{operation2})$



### Note

The total number of units allowed per operation can be lower than the maximum allowed if the script contains other operations which count towards the total units.

The following unit costs apply toward execution rate limits:

Unit	Cost
<code>y.query()</code>	2000 units
Methods of the <code>y</code> global object (such as <code>y.log()</code> and <code>y.rest()</code> )	1000 units
String concatenation	Length of the string being concatenated (1 unit per character)

<sup>18</sup> <http://bit.ly/ThF1b>

Unit	Cost
Operation of an object created via new	500 units per operation
Addition of an element	50 units

The following example calculates the number of units needed when adding two XML trees that each contain 10 elements:

`(10 elements + 10 elements) * 50 unit cost per element = 1000 units.`

---

# Chapter 3. Using Hosted Storage with YQL

In this Chapter:

- [“Introduction” \[58\]](#)
- [“Storing New Records” \[59\]](#)
- [“Using YQL to Read, Update, and Delete Records” \[60\]](#)
- [“Using Records within YQL” \[61\]](#)

## Introduction

YQL now provides two Open Data Tables, `yql.storage` and `yql.storage.admin`, that allow you to store and work with data using YQL itself. These default tables are available in the YQL console under the "yql" category. You interact with these Open Data Tables using the same keywords used with other YQL statements: namely SELECT, INSERT, UPDATE, and DELETE.

## About YQL Hosted Storage

Data that you store for use with YQL is hosted in Yahoo!'s Sherpa cloud storage infrastructure. It is a scalable, elastic, and geographically distributed storage system used by many Yahoo! services. Benefits of using Sherpa include:

- low latency that can handle a large number of concurrent requests.
- automated load balancing and failover to reduce operational complexity.
- high availability and fault tolerance.

For more information on Sherpa, refer to ["Moving to the Cloud" on the Yahoo! Developer Network \(YDN\) Blog<sup>1</sup>](#).

## Storage Limits and Requirements

The following storage limits and requirements apply to records stored using YQL:

- **Size Limit:** You can have up to 1000 records, with each record being up to 100KB.
- **Retention Limit:** Records not read, updated, or executed at least once every thirty days may be removed.
- **Record Format:** Records must be in a text-based format. Examples include JavaScript code, XML files, Open Data Tables, or YQL environment files.
- **Authentication for New Records:** All connections to `yql.storage.admin` must be authorized using two-legged OAuth. Alternatively, you can create new records using the YQL console. However, connections to `yql.storage` do not require authentication.

---

<sup>1</sup><http://developer.yahoo.net/blog/archives/2009/06/sherpa.html>

## Storing New Records

You create new records in three ways using the `yql.storage.admin` Open Data Table:

1. **value**: Create a record and insert text into it.
2. **url**: Create a record and insert content from an existing URL.
3. **name, url**: Create a record with an `execute` key that you name and insert content from an existing URL.



### Note

Any data you store within a record cannot exceed 100KB.

Upon creating a new record, YQL responds with three access keys, each of which serves a different function, as seen in the following response snippet:

```
...
  <results>
    <inserted>

<execute>store://35ad2c72-e353-41c4-9d21-4a7e5a1ec92</execute>
  <select>store://08fd2c74-d393-48c4-9ee1-3bf7e5a1ec92</select>

  <update>store://3cc85a99-6a89-4600-ade1-f7f83ecc4b83</update>

    </inserted>
  </results>
...
```

There are three access keys associated with each record in storage, each starting with `store://`:

- **execute**: This access key allows the value of the record to be used in a YQL statement containing `use` statements or referring to `env` files. You can share this access key with other developers, who can use the Open Data Table or environment file but will be unable to “read” the table definition or environment details.
- **select**: This access key allows the value of the record to be used in YQL `SELECT` statements.
- **update**: This access key allows the value of the record to be used in YQL `UPDATE` and `DELETE` statements.



### Note

Knowing the `execute`, `select`, or `update` access key is sufficient to perform those operations on the stored record. You should only share with others the access key that grants the desired permissions.

## Storing a New Record using Text

To create a new text record in storage for YQL, use the following statement format:

```
insert into yql.storage.admin (value) values ("example text content")
```



## Storing a New Record using Data from an URL

To copy the contents of an URL, such as an environment file or Open Data Tables, into a new record for YQL, use the following statement format:

```
insert into yql.storage.admin (url) values ("http://hostingdomain.com/mytable.xml")
```

## Storing a New Named Record using Data from an URL

To copy the contents of an URL, such as an environment file or Open Data Tables, into a new record with a custom `execute` access key, use the following statement format:

```
insert into yql.storage.admin (name,url) values ("newrecord","http://hostingdomain.com/mytable.xml")
```

When you create a record using the above format, the `execute` access key uses the name (`newrecord`) and top-level domain of the URL (`hostingdomain.com`) that you supply, as seen in the following response snippet:

```
...
  <results>
    <inserted>
      <execute>store://hostingdomain.com/newrecord</execute>
      <select>store://08fd2c74-d393-48c4-9ee1-3bf7e5a1ec92</select>

      <update>store://3cc85a99-6a89-4600-ade1-f7f83ecc4b83</update>

    </inserted>
  </results>
...
```



### Tip

Using the `SET` keyword in conjunction with a storage record, you can set and hide values, such as passwords, API keys, and other required values independently of YQL statements and API calls. For more information, refer to [Setting Key Values for Open Data Tables \[46\]](#).

## Using YQL to Read, Update, and Delete Records

Once you [create a record \[59\]](#) using the `yql.storage.admin` Open Data Table, you can use the `SELECT`, `UPDATE`, and `DELETE` keywords with `yql.storage` to access, modify, or delete records, respectively.

## Accessing Records using YQL

To read or access a record using YQL, use the following statement format that contains the `SELECT` keyword:

```
select * from yql.storage where name="store://08fd2c74-d393-48c4-9ee1-3bf7e5a1ec92"
```

Use the [select access key \[59\]](#) that YQL provided to you upon creating the record. The YQL response contains a `results` element similar to the following:

```
...
  <results>
    <value>example test content</value>
  </results>
...
```

## Deleting Records using YQL

Use the `DELETE` keyword in YQL to delete a record, similar to the following:

```
delete from yql.storage where name="store://3cc85a99-6a89-4600-ade1-
f7f83ecc4b83"
```

Use the [update access key \[59\]](#) that YQL provided to you upon creating the record. The YQL response contains a `results` element similar to the following:

```
...
  <results>
    <success>store://3cc85a99-6a89-4600-ade1-f7f83ecc4b83
deleted</success>
  </results>
...
```

## Updating Records using YQL

Use the `UPDATE` keyword in YQL to modify a record, similar to the following:

```
update yql.storage set value="new value" where name="store://3cc85a99-
6a89-4600-ade1-f7f83ecc4b83"
```

Use the [update access key \[59\]](#) that YQL provided to you upon creating the record. The YQL response contains a `results` element similar to the following:

```
...
  <results>
    <success>Updated
store://3cc85a99-6a89-4600-ade1-f7f83ecc4b83</success>
  </results>
...
```



### Note

If you create a [named execute record \[60\]](#), using a name and a url, you can only update the contents of this record to an URL on the same URL domain.

## Using Records within YQL

If your record contains an environment file, an Open Data Table, or JavaScript, YQL can "execute" or run it in a read-only manner.

## Using Hosted Environment Files

When a record contains a [YQL environment file \[44\]](#), use the corresponding [execute access key \[59\]](#) in YQL calls as you would any environment file. The following example uses URL encoding for the execute access key and appends it to the YQL console URL:

```
http://developer.yahoo.com/yql/console?env=store%3A%2F%2Fopendatatables
```

## Using Hosted YQL Open Data Tables

When a record contains an Open Data Table, use the corresponding [execute access key \[59\]](#) in YQL calls as you would any Open Data File. In the following example, the USE keyword invokes the access key, which is then used as mytable:

```
use "store://35ad2c72-e353-41c4-9d21-4a7e5a1ec92" as mytable; select *
from mytable;
```

## Including Hosted JavaScript

When a record contains JavaScript, use the [select access key \[59\]](#) in YQL to include it in an Open Data Table.



### Note

You do not use the [execute access key \[59\]](#) with JavaScript includes using [y.include \[21\]](#) because parsing JavaScript is done in the same manner as reading it.

To do a JavaScript include, first insert the JavaScript into a record in the following manner:

```
insert into yql.storage.admin (name, url) values ('testjs', 'http://jav-
arants.com/yql/test.js')
```

The response from YQL contains the following URLs:

```
<inserted>
  <execute>store://javarants.com/testjs</execute>
  <select>store://f7c24314-b0fd-488b-a67c-1d711ca3bc21</select>

  <update>store://077b02c0-b6e8-4923-90a4-9b8096ccdaec</update>

</inserted>
```

Use the select URL in the [select element \[5\]](#) of the Open Data Table in the following manner:

```
// contains:
// var success = true;
y.include('store://f7c24314-b0fd-488b-a67c-1d711ca3bc21');
response.object = <success>{success}</success>;
```