# Introduction to the Data Cookbook API

Brought to you by IData, Inc

# Table of Contents

# 1 How Data Cookbook Service APIs Work

Data Cookbook service APIs allow external systems to interact with the Data Cookbook application programmatically, similar to the way users interact with the system when they use it in a web browser. An external system must first authenticate itself with the Data Cookbook. Then, once it has been successfully authenticated, it can send requests that enable it to take advantage of any of the supported services listed below. With the services currently implemented, this allows a computer to search for terms based on keywords or look up specific terms based on term or term version ID.

These service requests are implemented using the same HTTP protocol that browsers use to talk with websites. Data Cookbook services listen for specially formed HTTP requests that contain information on who is making the request so we can authenticate and make sure that only authorized users have access to data exposed by services, and that they contain information needed to process the request (for example, the terms search is searching for the desired format of the output.). When each request is received, it is processed by the Data Cookbook and a response is sent back in a format that can easily be read by a computer (either xml, rss, json, or html, depending on the output format specified in the request). The requester can then use the results however it wants.

These services could support a widget on a portal page that would let a user search for terms in the Data Cookbook, then click to go into the Data Cookbook itself to see more details. They could also support a program that would sync terms between the Data Cookbook and another application, perhaps in a scenario where you want changes to terms that are in the Data Cookbook to be reflected in a second, external application that uses the same terms.

# 2 Supported Services

## 2.1 Service Login

The service_login service allows service clients to pass in a username and password to authenticate a user in the Data Cookbook. It returns an authentication token that can then be used for authentication in subsequent requests instead of the username and password (you can still pass the username and password with each service request if you prefer). This allows a widget implementer, for example, to make a login call from a server before rendering a widget so that the widget placed on the web page can authenticate based on a token. It will not have to have a user's Data Cookbook username and password embedded within the page. For more details, see the document titled "Data Cookbook Service Login API."

## 2.2 Term Search

The term_search service allows service clients to search for terms within a given institution's account. The API allows you to pass in a space-delimited list of keywords for which you want to search and returns a list of results whose name, functional definition, or technical definition contain any of the requested words. For more details, see the document titled "Data Cookbook Term Search API."

## 2.3 Term Lookup

The term_lookup service allows service clients to look up specific terms within a given institution's account. The API allows one to pass in either a term name, a term ID, or the ID of a particular version of a term and it returns the matching term, if one is found, else it returns nothing. For more details, see the document titled "Data Cookbook Term Lookup API".

# 3 How Service APIs Can Be Useful

Service APIs enable all kinds of different interactions between outside programs and the Data Cookbook. A server could use the term search API to include a list of suggested term matches to go alongside search matches on a given school's search site. A server with a program that allows Data Cookbook terms to be tied to its own entities could use the API to pull in a link to more information on terms whose IDs it retrieves from its database. And in a web browser, the APIs could be used to support AJAX calls that would let users look up terms from widgets inside portal or reporting web pages.

Service APIs allow your programs to interact with and retrieve information stored in the Data Cookbook, enabling your software to include the rich, collaborative data the Data Cookbook facilitates in other applications.

# 4 Implementation Guide

Data Cookbook services can be invoked in a couple of different ways, and the results of service invocations can then be formatted in a number of ways. This flexibility is intended to make consumption of these services relatively straightforward in a number of different scenarios, including as AJAX calls from a browser and as RPC invocations in server applications. This implementation guide is broken out into four sections:

- Service requests
- Service responses
- Sample implementation of a PHP service consumer
- Sample implementation of an AJAX service consumer

This documentation and these samples should give you a good idea of how to get started using Data Cookbook services. If you want more information on HTTP, we also have added a brief introduction to the HTTP protocol following the Implementation Guide section in this document.

## *4.1 Introduction to Data Cookbook Service Requests*

Data Cookbook services can be invoked one of three ways:

- Input parameters passed to the service as a GET query string, on the URL.
- Input parameters passed to the service in the body of an HTTP POST request, as a form post would submit them.
- XML passed to the service in the body of a post request.

If you choose to implement requests using form input parameters, a search request would be either an HTTP(S) GET request that has each of the non-optional parameters for a given service in its query string, and that could also contain any of the other parameters, or an HTTP(S) POST request that has each of the non-optional parameters for the service stored as form inputs in the body of the request, and that could also contain any of the other parameters. You should only place each parameter in the request once. If you accidentally place a parameter there twice and the two instances have two different values, we can't guarantee which will be used in processing your request.

Each service has its own API document that outlines the following:

- What parameters are able to be passed to the service.
- Which parameters are required and which are optional.
- Specific examples of how to invoke the service and what to expect in each of the supported response types.

The different services are standard to a point, but do have some differences in the data returned, so you should make sure to consult the service API guide for the particular service you are consuming.

If you choose to implement requests using XML, Data Cookbook uses a standard XML dialect for all of its service requests and responses. The basic format of request and response XML is outlined below in the section on Data Cookbook request XML.

### 4.1.1 Data Cookbook GET Query String Request

A GET query string is a set of name-value pairs of information that are appended to the end of the URL to which an HTTP request is sent. The query string starts with a question mark ("?") directly after the end of the URL. Each name-value pair is separated by an ampersand ("&"), and for each name-value pair, the name is separated from the value by an equal sign ("="). A sample query string is below. For more details, see the HTTP quick reference later in this document, or consult documentation on details of the HTTP protocol.

***Sample HTTP GET query string request:***

https://idata.datacookbook.com/institution/terms/search.xml?un=jonathanmorgan&pw=nopeeking!&requestType=term_search&search=hungry&outputFormat=rss

### 4.1.2 Data Cookbook POST Form Input Request

A POST form input request uses the same string of name-value pairs of information that are appended to the end of the URL to which an HTTP request is sent in a GET request. Only the query string is stored in the body of the request and no question mark is needed at the beginning of the string. In a form input string, each name-value pair is separated by an ampersand ("&") and for each name-value pair, the name is separated from the value by an equal sign ("="). A sample POST request of this type is below. For more details, see the HTTP quick reference later in this document, or consult documentation on details of the HTTP protocol.

**Sample HTTP POST form input request:**

```
POST /institution/terms/search HTTP/1.0
Host: idata.datacookbook.com
[blank line here]
un=jonathanmorgan&pw=nopeeking!&requestType=term_search&search=hungry&
outputFormat=rss
```

### 4.1.3 Data Cookbook Request XML

In a request that sends Data Cookbook request XML, the XML request is stored in the body of an HTTP(S) POST request instead of a query string. Data Cookbook request XML is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceTransaction xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ServiceRequest serviceName="service_login">
        <ParameterList>
            <Parameter>
                <Name>un</Name>
                <Value>jonathanmorgan</Value>
            </Parameter>
            <Parameter>
                <Name>pw</Name>
                <Value>nopeeking!</Value>
            </Parameter>
            <Parameter>
                <Name>requestType</Name>
                <Value>service_login</Value>
            </Parameter>
             <Parameter>
                <Name>outputFormat</Name>
                <Value>json</Value>
            </Parameter>
        </ParameterList>
    </ServiceRequest>
</ServiceTransaction>
```

The request starts with an XML declaration, followed by a “ServiceTransaction” element that wraps both requests and responses. When the transaction is a request, the ServiceTransaction element contains a ServiceRequest element with a “serviceName” attribute that contains the request type of the service you are invoking. Then, in the “ParameterList” element, the request contains a list of parameters, each in a “Parameter” element that contains a “Name” and “Value” element. The “Name” element contains the name of the parameter. The “Value” element contains the value for the parameter named in the “Name” element.

## Sample HTTP POST Data Cookbook XML request:

```
POST /institution/terms/search HTTP/1.0
Host: idata.datacookbook.com
[blank line here]
<?xml version="1.0" encoding="UTF-8"?>
<ServiceTransaction xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ServiceRequest serviceName="term_search">
        <ParameterList>
            <Parameter>
                <Name>un</Name>
                <Value>jonathanmorgan</Value>
            </Parameter>
            <Parameter>
                <Name>pw</Name>
                <Value>nopeeking!</Value>
            </Parameter>
            <Parameter>
                <Name>requestType</Name>
                <Value>term_search</Value>
            </Parameter>
            <Parameter>
                <Name>search</Name>
                <Value>hungry</Value>
            </Parameter>
            <Parameter>
                <Name>outputFormat</Name>
                <Value>rss</Value>
            </Parameter>
        </ParameterList>
    </ServiceRequest>
</ServiceTransaction>
```

## *4.2 Introduction to Data Cookbook Service Responses*

The response from a Data Cookbook service request can be returned in a number of formats: RSS, HTML, XML, and JSON.

If you plan on accessing services using AJAX, then JSON or HTML will make more sense, though either RSS or XML would be implementable, as well.

If you are planning on interacting with services programmatically on a server, then it makes more sense to use either RSS or XML, though some server-side languages let you parse JSON, too, and can also do a passable job treating HTML like XML because xhtml is returned.

All services implement all output formats, though some might include a message if the output format is not appropriate for the request. For example, if HTML is requested as the output type for an authentication request, a message is included in the HTML stating that the service is not intended to be displayed to the public.

All response types will return an indication of whether the request succeeded or failed. XML and JSON give a more granular status with a separate status code and status message. RSS will contain the status message in the description of the feed. HTML will return an empty <div> on error.

### 4.2.1 Data Cookbook Response XML

Data Cookbook response XML is the default response format for Data Cookbook services. It is returned in the body of the HTTP response to a service request.

The response starts with an XML declaration, followed by a “ServiceTransaction” element that wraps both requests and responses. When the transaction is a response, the “ServiceTransaction” element will always contain a “ServiceResponse” element with a “serviceName” attribute that contains the request type of the service you are invoking. There will then be a “ResponseStatus” element that contains a “ResponseCode” element (whose value will be 0 for success, -1 for general error, and potentially other numbers for errors specific to a given service) and a “ResponseMessage” element. The “ResponseMessage” element will contain a message describing the outcome of the request. If the request was successful, it could contain “Success!” or a more detailed success message. If there was an error, the “ResponseMessage” is intended to contain a detailed and specific error message. After the “ResponseStatus” element, each service will return different XML that contains the results of processing the request (the above contains the results of processing the term_search request used as an example in the request section above).

**Data Cookbook response XML is structured as follows:**

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceTransaction xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ServiceResponse serviceName="term_search">
        <ResponseStatus>
            <ResponseCode>0</ResponseCode>
            <ResponseMessage>Success!</ResponseMessage>
        </ResponseStatus>
        <TermList>
            <Term>
                <name>hungry student</name>
                <functional-definition>A hungry student is one who has
a need that he or she feels must be met.  This could be a hunger for
food.  It could be a hunger for knowledge.  More specific child terms
differentiate.</functional-definition>
                <perma-link-
url>http://idata.datacookbook.com/terms/12345</perma-link-url>
            </Term>
             <Term>
                <name>hungry student (food)</name>
                <functional-definition>A student who is hungry for
food is one who feels a need to eat food.  Might or might not actually
need food.</functional-definition>
                <perma-link-
url>http://idata.datacookbook.com/terms/12346</perma-link-url>
            </Term>
            <Term>
                <name>hungry student (knowledge)</name>
                <functional-definition>A student who is hungry for
knowledge is one who is driven to learn, to ingest knowledge.  Similar
to thirsting for knowledge (see term "thirsty student
(knowledge)".</functional-definition>
                <perma-link-
url>http://idata.datacookbook.com/terms/12347</perma-link-url>
            </Term>
        </TermList>
    </ServiceResponse>
</ServiceTransaction>
```

### 4.2.2 Data Cookbook Response JSON

JSON is a string-based method of creating structured data that is generally used to support a JavaScript AJAX implementation of API requests from within a browser. JSON uses JavaScript's generic object definition syntax to create a data structure that can be easily parsed by JavaScript and then interacted with in the way one would interact with any other JavaScript object. The ability to parse JSON is also written into other common scripting languages like PHP, Ruby, and Perl.

The JSON response is also stored in the body of the HTTP response to a Data Cookbook service request. It contains the same granularity and level of detail found in the XML response, and is structured the same way, though using a different syntax: result with status code and message, then service-specific information in JavaScript objects or arrays, depending on the service.

When you request JSON output, you have a few parameters you can also pass, to more specifically define how you want the JSON returned:

- **jsonFunction** (optional) – Name of function you want JSON to be passed to on return.
- **jsonVariable** (optional) – Name of variable you want JSON to be assigned to on return.

If you do not specify a JSON variable or function, you'll simply get the JSON response object as the sole contents of your response. This is the format JavaScript libraries like JQuery and Scriptaculous expect.

If you specify a jsonFunction, then the JavaScript that is returned will wrap the JSON response object in a call to the function you specify, so the response will be passed to that function for processing. This is typically used with JSONP, a cross-browser AJAX implementation described below.

If you specify a jsonVariable, then the JavaScript that is returned will assign the JSON response object returned to the variable whose name you passed in, so it will be accessible later in the page by accessing that variable.

One common way of implementing this is to use the JavaScript call-back or JSONP (JSON with padding) method of implementing cross-domain JSON AJAX requests. This is the most straightforward method of implementing cross-domain AJAX. In this method, part of our request is the name of the JavaScript function that that should be invoked on completion of the request and passed the JSON output, for processing. The calling page is responsible for implementing that method and actually processing the results.

To invoke the API, the calling page implements the call-back method, then includes a JavaScript <script> tag with the API call as the URL. The API performs the search and returns JavaScript formatted like the sample below that invokes the call-back function. In this sample, the function the consumer would have to implement is named “processServiceResponse()”. You specify the name of the callback function using the optional jsonFunction parameter. The sample JSON output below is an example of this kind of implementation, for the following request:

```
https://idata.datacookbook.com/institution/terms/search.xml?un=jonatha
nmorgan&pw=nopeeking!&requestType=term_search&search=hungry&outputForm
at=json&jsonFunction=processServiceResponse
```

## Sample Data Cookbook JSON service response:

```
processServiceResponse(
    {
        ServiceName : "term_search",
        ResponseStatus : {
            ResponseCode : 0,
            ResponseMessage : "Success!"
        },
        TermList : [
            {
                "version" : {
                    "name" : "hungry student",
                    "term_functional_definition" : "A hungry student
is one who has a need that he or she feels must be met.  This could be
a hunger for food.  It could be a hunger for knowledge.  More specific
child terms differentiate.",
                    "perma_link_url" :
"http://idata.datacookbook.com/institution/terms/12345"
                }
            },
            {
                "version" : {
                    "name" : "hungry student (food)",
                    "term_functional_definition" : "A student who is
hungry for food is one who feels a need to eat food.  Might or might
not actually need food.",
                    "perma_link_url" :
"http://idata.datacookbook.com/institution/terms/12346"
                }
            },
            {
                "version" : {
                    "name" : "hungry student (knowledge)",
                    "term_functional_definition"FunctionalDefinition :
"A student who is hungry for knowledge is one who is driven to learn,
to ingest knowledge.  Similar to thirsting for knowledge (see term
\"thirsty student (knowledge)\".",
                    "perma_link_url" :
"http://idata.datacookbook.com/institution/terms/12347"
                }
            }
        ]
    }
);
```

### 4.2.3 Data Cookbook Response RSS

In an RSS response, an RSS document is the body of the HTTP(S) response, with the item or items returned in the feed are the results of the service request. This could get ugly for complicated services, and so some might return XML in the description of the <item>. Our API implements RSS 2.0. For each item returned, the service will decide how to populate the standard RSS <item> child elements: <title>, <link>, <description>, <pubDate>, and <guid>.

**Sample RSS service transaction response:**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rss version="2.0">
   <channel>
      <title>Data Cookbook term_search results</title>
      <description>Here's what we found for your search on
"hungry".</description>
      <link>
        http://idata.datacookbook.com/institutions/terms/search.xml
      </link>
   <language>en-us</language>
   <copyright>Copyright 2010, IData, Inc.</copyright>
   <generator>IData Inc. Data Cookbook</generator>
   <managingEditor>bparish@idatainc.com (Brian Parish, President and
CEO, IData, Inc.)</managingEditor>
   <webMaster>kdezio@idatainc.com (Ken Dezio, CTO, IData,
Inc.)</webMaster>
   <docs>http://blogs.law.harvard.edu/tech/rss</docs>
   <pubDate>Fri, 05 Feb 2010 05:00:00 CST</pubDate>
   <lastBuildDate>Fri, 05 Feb 2010 13:30:08 EST</lastBuildDate>
   <ttl>30</ttl>
   <item>
     <title>hungry student</title>
    <link>http://idata.datacookbook.com/institution/terms/12345</link>
     <description><![CDATA[A hungry student is one who has a need that
he or she feels must be met.  This could be a hunger for food.  It
could be a hunger for knowledge.  More specific child terms
differentiate.]]></description>
     <pubDate>Fri, 05 Feb 2010 13:27:00 EST</pubDate>
     <guid
isPermaLink="true">http://idata.datacookbook.com/institution/terms/123
45</guid>
   </item>
   <item>
     <title>hungry student (food)</title>
     <link>http://idata.datacookbook.com/institution/terms/12346</link
>
     <description><![CDATA[A student who is hungry for food is one who
feels a need to eat food.  Might or might not actually need
food.]]></description>
```

```
     <pubDate>Fri, 05 Feb 2010 12:42:00 EST</pubDate>
     <guid isPermaLink="true">
        http://idata.datacookbook.com/institution/terms/12346
     </guid>
   </item>
   <item>
     <title>hungry student (knowledge)</title>
     <link>
       http://idata.datacookbook.com/institution/terms/12347
     </link>
     <description><![CDATA[A student who is hungry for knowledge is
one who is driven to learn, to ingest knowledge.  Similar to thirsting
for knowledge (see term "thirsty student
(knowledge)".]]></description>
      <pubDate>Fri, 05 Feb 2010 13:02:00 EST</pubDate>
      <guid isPermaLink="true">
        http://idata.datacookbook.com/institution/terms/12347
      </guid>
    </item>
  </channel>
</rss>
```

### 4.2.4 Data Cookbook Response HTML

In an HTML response, the results of your service request will be returned in a simple <div> structure in HTML, with classes and names assigned so you can target CSS to fit their appearance to your application, or use DOM to target the elements for more complex processing. The HTML will always be wrapped in an outer <div> with the class of "Data Cookbook services," so you can target CSS to only apply inside this element for custom display of Data Cookbook HTML.

***Sample HTML service transaction response:***

```
<div class="DATA_COOKBOOK_services">
    <div class="DC_TermList">
        <div class="DC_Term" name="hungry student" id="12345">
            <div class="DC_TermName">hungry student</div>
            <div class="DC_FunctionalDefinition">A hungry student is
one who has a need that he or she feels must be met.  This could be a
hunger for food.  It could be a hunger for knowledge.  More specific
child terms differentiate.</div>
            <div class="DC_URL"><a
href="http://idata.datacookbook.com/terms/12345">http://idata.datacook
book.com/terms/12345</a></div>
        </div>
        <div class="DC_term" name="hungry student (food)" id="12346">
            <div class="DC_termName">hungry student (food)</div>
            <div class="DC_functionalDefinition">A student who is
hungry for food is one who feels a need to eat food.  Might or might
not actually need food.</div>
            <div class="DC_URL"><a
href="http://idata.datacookbook.com/terms/12346">http://idata.datacook
book.com/terms/12346</a></div>
        </div>
        <div class="DC_term" name="hungry student (knowledge)"
id="12347">
            <div class="DC_termName">hungry student (knowledge)</div>
            <div class="DC_functionalDefinition">A student who is
hungry for knowledge is one who is driven to learn, to ingest
knowledge.  Similar to thirsting for knowledge (see term "thirsty
student (knowledge)".</div>
            <div class="DC_URL"><a
href="http://idata.datacookbook.com/terms/12347">http://idata.datacook
book.com/terms/12347</a></div>
        </div>
    </div>
</div>
```

## 4.3 Sample Implementation of a PHP Service Consumer

There are two basic ways you can implement a service request – as part of a server side program, or in JavaScript, as part of a web page that will be run inside each user's browser. Regardless of where you implement, to request a Data Cookbook service, you create and send an HTTP request to the URL that provides the service, and you'll receive an HTTP response with the results of processing your request.

### 4.3.1 Implementing a Service Request

To implement an HTTP service request on a server, you'll use a server-side language library in your language of choice that implements an HTTP Client to set up your HTTP request and then send it off to the Data Cookbook. In PHP, for example, you can use the curl library as follows.

***Sample PHP curl implementation of an HTTP POST XML service request:***

```
<?php

// try a curl_init call, to see if we have curl installed.
$curlHandle = curl_init();

// set up request - tell it we are sending POST
curl_setopt( $curlHandle, CURLOPT_POST, true ); // post request

// set up request headers
$headerArray = array();
$headerArray[ 'Content-Type' ] = 'text/xml'; // important to set this,
so recipient knows this is XML, not form data.
curl_setopt( $curlHandle, CURLOPT_HTTPHEADER, $headerArray );

// if you already have an authorization token, you can add it to the
request as a cookie
/*
curl_setopt( $curlHandle, CURLOPT_COOKIE,
"_itdb_session=BAh7BzoMdXNlcl9pZGlDOg9zZXNzaW9uX2lkIiU1ZTQ4ZTEyN2M3ZTg
3NjMwNjM5YzI5YzVkYzUyYjE2Yw%3D%3D--
8c5c23c155e4f6f75e342b59fa9413fd9d605309" );
*/

//=================================================================
==========
// search request
//=================================================================
==========

// set the URL of the service.
curl_setopt( $curlHandle, CURLOPT_URL,
'http://idata.datacookbook.local/institution/terms/search' );
```

```
// put the XML for the request in the body of the request.
curl_setopt( $curlHandle, CURLOPT_POSTFIELDS, '<?xml version="1.0"
encoding="UTF-8"?>
<ServiceTransaction xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ServiceRequest serviceName="term_search">
        <ParameterList>
            <Parameter>
                <Name>un</Name>
                <Value>dmcandrews</Value>
            </Parameter>
            <Parameter>
                <Name>pw</Name>
                <Value>1data1nc</Value>
            </Parameter>
            <Parameter>
                <Name>requestType</Name>
                <Value>term_search</Value>
            </Parameter>
            <Parameter>
                <Name>search</Name>
                <Value>student</Value>
            </Parameter>
            <Parameter>
                <Name>returnLimit</Name>
                <Value>5</Value>
            </Parameter>
            <Parameter>
                <Name>outputFormat</Name>
                <Value>xml</Value>
            </Parameter>
        </ParameterList>
    </ServiceRequest>
</ServiceTransaction>' ); // body of the post request.

// include the headers in the output, not just the body of the
response
curl_setopt( $curlHandle, CURLOPT_HEADER, true ); // include the
header in the output

// optional - tell CURL to not send cookies from previous sessions,
and to make a new connection.
curl_setopt( $curlHandle, CURLOPT_FRESH_CONNECT, true ); // for now,
don't store cookies...

// configure CURL to return the response as the return value of
curl_exec instead of a boolean success flag.
curl_setopt( $curlHandle, CURLOPT_RETURNTRANSFER, 1 );

// could also configure it to store results to a stream - in this
case, a file:
//$fp = fopen( "somefile.txt", "w" );
//curl_setopt( $curlHandle, CURLOPT_FILE, $fp );
```

```
// if HTTP password-protected, you can pass username and password
//curl_setopt( $curlHandle, CURLOPT_USERPWD,
"anonymous:your@email.com" );

// send the request.
$output = curl_exec( $curlHandle );

// close the connection
curl_close( $curlHandle );
?>
```

### 4.3.2 Processing a Service Response

On a server, once you programmatically submit your request, you then wait for the response and parse it according to the output format you requested. If you requested XML (or RSS or HTML), after you retrieve the response data from the body of the response, you can then process the contents of the body of the response like any other XML document. Parse it and then use DOM to interact with it or write a SAX parser to walk over it and process its contents as it goes. In the curl example above, the variable $output contains the full HTTP response, including the headers. Once you get that response, you can parse it and process it as you would any other HTTP response (for more details, see Section 5.7 HTTP Response).

## *4.4 Sample Implementation of an AJAX Service Consumer*

There are two basic ways you can implement a service request – as part of a server side program, or in JavaScript, as part of a web page that will be run inside each user's browser. Regardless of where you implement, to request a Data Cookbook service, you create and send an HTTP request to the URL that provides the service, and you'll receive an HTTP response with the results of processing your request.

### 4.4.1 Implementing a Service Request

If you were implementing a widget in a browser, you would use some form of AJAX, implemented in JavaScript, to connect to the service provider. You could use a library such as jQuery to make an XmlHttpRequest. You could also use JSONP to set up a function to handle the results of your service request, and then ask that the service request wrap its results in a call to the method you created. This example uses JSONP.

Steps for implementing a service request in JSONP:

1. Implement the function that accepts the results of the service request and processes them.
2. Implement the code to gather the request parameters, turn them into a service request URL, and then dynamically place a script tag in the body of the document that references the service URL.

Step 2 could be a form whose submit function pulls the parameters together and then generates the <script> tag. It could be done on the server as the page is rendering, such that the <script> tag is rendered by the server and included in the page that is returned to the browser. It could also be rendered in JavaScript in the browser, based on things the user does on the page, but not based on a form.

### 4.4.2 Processing a Service Response

In a browser, you'd probably either ask for JSON (a JavaScript data structure that is pre-packaged so the output of your service is ready for JavaScript to interact with it as soon as it is returned to the browser), HTML, or RSS. If you have JSON, you can just interact with the JSON object as you would any other generic JavaScript object. If you have HTML, you can either place it in the page or manipulate it using DOM before adding it to your page. If you have requested RSS, you could either use a JavaScript library to render the RSS, or you could parse it yourself using DOM and use the values to create your own HTML.

### 4.4.3 Example of an AJAX JSONP Implementation

The example shows how to create a form on a web page that accepts a term for which to search. On submittal, it creates a term search API request and outputs the results. The example below is divided into two parts: the JavaScript file that implements the example, and the HTML where the example sits on the page. In the JavaScript file, there are three sections – the callback function, an object that models a request and contains methods for putting a request together, and the event handler that the form calls when a user submits their request.

In the example below, the form in the HTML calls the DataCookbookService_processEvents() method when its submit button is pressed. This JavaScript function uses the DataCookbookService JavaScript class to store all the information needed for the request, then create and output the <script> tag that invokes the request. The request is set up so that it invokes the DataCookbookService_processResponse() method on completion. DataCookbookService_processResponse() accepts the JSON output of the service request, parses it, and outputs the terms in HTML on the web page, in the div with id="data_cookbook_search_results".

#### 4.4.3.1 AJAX JSONP Implementation Example Javascript Code

```
/*
 * file: DataCookbook-termSearch-API.js
 * purpose: Implements integration of Data Cookbook term search API
code with a web
 *    site.  Is essentially a widget.
 * preconditions: This javascript should be loaded before you try to
place term
 *    search HTML on the page, so it is available.
 */

/*
 * Function: DataCookbookService_processResponse( data_IN )
 * Accepts JSON object structured as follows:
{
    ServiceName : "term_search",
    ResponseStatus : {
        ResponseCode : 0,
        ResponseMessage : "Success!"
    },
    TermList : [
        {
            "version": {
                "perma_link_url":
"http://idata.datacookbook.com/institution/terms/154",
                "name": "Admitted student",
                "term_id": 154,
                "id": 184,
                "term_functional_definition": "Applicant who is
offered admission to a degree-granting program at your institution."
            }
        },
        {
            "version": {
                "perma_link_url":
"http://idata.datacookbook.com/institution/terms/159",
                "name": "Books and supplies",
                "term_id": 159,
                "id": 189,
                "term_functional_definition": "Average cost of books
and supplies. Do not include unusual costs for special groups of
students (e.g., engineering or art majors), unless they constitute the
majority of students at your institution."
            }
        }
    ]
}
 * Processes the information within, outputs an alert if there is an
error,
 *    either in the processing of the request or indicated by the
status code
 *    returned from the service.
 * postconditions: renders HTML and places it on the page, even if
there is an error.
 */
function DataCookbookService_processResponse( data_IN )
{
    // declare variables
    var DIV_ID_TERM_SEARCH_OUTPUT = 'data_cookbook_search_results';
    var HTML_NO_MATCHES_FOUND = '<div class="Events_ListItem
event"><span class=\"vevent\"><div class="eventTitle summary"><div
class="eventTitle summary"><p>No matches found for search
term.</p></div></span></div>';
    var requestType = '';
    var responseCode = '';
    var responseMessage = '';
    var termList = '';
    var termCount = -1;
    var termListIndex = -1;
    var currentTerm = null;
    var termName = '';
    var termFunctionalDefinition = '';
    var termUrl = '';
    var resultsElement = '';
    var headerElement = '';
    var headerTextElement = null;
    var listElement = '';
    var termListHtml = '';
    var bodyElement = '';

    //alert( "In callback DataCookbookService_processResponse, data: "
+ data_IN );

    // get response status
    responseCode = data_IN.ResponseStatus.ResponseCode;
    responseMessage = data_IN.ResponseStatus.ResponseMessage;

    // if success (code = 0), get requestType
    if ( responseCode == 0 )
    {
        // success.  Get request type to see what we need to do now.
        requestType = data_IN.ServiceName;
        //alert( "Request succeeded. ServiceName = " + requestType );

        // got a request type?
        if ( requestType != '' )
        {
            // if term search, build HTML output of returned search
            //    results.
            if ( ( requestType == "term_search" ) || ( requestType ==
'term_lookup' ) )
            {
                // set up the results area.
                // get the div into which we will place the HTML,
clear it
                //    out, then set it to invisible.
                resultsElement = document.getElementById(
DIV_ID_TERM_SEARCH_OUTPUT );
                resultsElement.style.display = "none";
                resultsElement.innerHTML = "";

                // create the results header  HTML string
                headerElement = document.createElement( "div" );
                headerElement.setAttribute( "class", "viewHeader" );

                // add results title
                headerTextElement = document.createElement( "h3" );
                headerTextElement.appendChild(
document.createTextNode("Search Results") );
                headerElement.appendChild( headerTextElement );

                // append to results
                resultsElement.appendChild( headerElement );

                // now, build and attach the element that will hold
the
                //    returned search terms
                bodyElement = document.createElement( "div" );
                bodyElement.setAttribute( "class", "termList" );
                // bodyElement.appendChild( document.createElement(
"br" ) );
                resultsElement.appendChild( bodyElement );

                // loop over term array, load and render each
                //    term in the list.
                termList = data_IN.TermList;

                // got a term list?
                if ( termList != null )
                {
                    // get count, see if we have terms.
                    termCount = termList.length;
                    if ( termCount > 0 )
                    {
                        // got at least one term.  Get the element we
will store the
                        //    results in, then output stuff that goes
before the
                        //    list of terms.
                        termListHtml += "<ul>\n";

                        // got at least three terms?
                        if ( termCount > 3 )
                        {
                            // yes!  Set height and make box scroll.
                            bodyElement.style.height = "300px";
                            //resultsElement.style.width = "100px";
                            bodyElement.style.overflow = "scroll";
                        }

                        //alert( "Request type has terms.  Before
loop. Term count: " + termCount );
                        for( termListIndex = 0; termListIndex <
termCount; termListIndex++ )
                        {
                            //alert( "Top of term loop. Term index: "
+ termListIndex );
                            // get next term to process.
                            currentTerm = termList[ termListIndex ][
'version' ];

                            // get cookie properties
                            //cookieName = currentCookie.Name +
"_test";
                            termName = currentTerm.name;
                            termFunctionalDefinition =
currentTerm.term_functional_definition;
                            termUrl = currentTerm.perma_link_url;

                            //alert( "In term loop. Values for term "
+ termListIndex + ": name=" + termName + ", URL=" + termUrl + ",
functional definition=" + termFunctionalDefinition );

                            // create list HTML for term. I got lazy
and did not do this part in DOM.
                            termListHtml += "<li>\n";
                            termListHtml += "<p><a class=\"eventLink
url\" href=\"" + termUrl + "\">" + termName + "</a></p>\n";
                            termListHtml += "<p>" +
termFunctionalDefinition + "</p>\n";
                            termListHtml += "</li>\n";

                            //alert( "Outputting term: " + termHtml );
                            // output the HTML

                        } //-- end loop over terms. --//

                        // close the <ul>
                        termListHtml += "</ul>\n";

                        // put the list of terms in the list element
                        listElement.innerHTML = termListHtml;

                    } //-- end check to see if we have any terms --//
                    else //-- no matches found. --//
                    {
                        listElement.innerHTML = HTML_NO_MATCHES_FOUND;
                    }

                }
                else
                {
                    // no terms returned.  Output a no results
message.
                    listElement.innerHTML = HTML_NO_MATCHES_FOUND;
                }

                 // done with the loop, so make visible
                resultsElement.style.display = "block";

            } //-- end conditional to find known services. --//
            else
            {
                alert( "Error processing Data Cookbook service
request: Service " + requestType + " unknown.  ResponseCode: " +
responseCode + "; ResponseMessage: " + responseMessage );
            }
        }
        else //-- no request type in response - which service? --//
        {
            // no request type returned.  Error.
            alert( "Error processing Data Cookbook service request: No
service type returned.  ResponseCode: " + responseCode + ";
ResponseMessage: " + responseMessage );
        }
    }
    else //-- error response code returned --//
    {
        // error.  Output alert, do nothing.
        alert( "Error processing Data Cookbook service request:  Error
code returned.  ResponseCode: " + responseCode + "; ResponseMessage: "
+ responseMessage );
    }
} //-- end function DataCookbookService_processResponse --//


/*
 * DataCookbookService class holds variables and methods for
integrating
 *    with Data Cookbook services.
 */
function DataCookbookService()
{
    // declare variables
    this.SERVICE_PROTOCOL = 'http';
        this.SERVICE_URL_PREFIX = '://';
        this.SERVICE_URL_HOST = 'idata.datacookbook.com';
        this.SERVICE_URL_PATH_TERM_SEARCH =
'/institution/terms/search';
        this.SERVICE_URL_PATH_TERM_LOOKUP =
'/institution/terms/lookup';
        this.SERVICE_URL_PATH = this.SERVICE_URL_PATH_TERM_SEARCH;
    this.SERVICE_DIV_ID = 'DataCookbookServiceInclude';
    this.SERVICE_CALLBACK = 'DataCookbookService.processResponse';

    // required parameter names for GET/POST parameters.
        this.PARAM_PASSWORD = 'pw';
    this.PARAM_REQUEST_TYPE = 'requestType';
        this.PARAM_SEARCH = 'search';
        this.PARAM_USERNAME = 'un';

        // optional parameters
        this.PARAM_JSON_FUNCTION = 'jsonFunction';
        this.PARAM_JSON_VARIABLE = 'jsonVariable';
        this.PARAM_OUTPUT_FORMAT = 'outputFormat';

        // parameters for lookup
        this.PARAM_LOOKUP = 'lookup';
        this.PARAM_MATCH_TYPE = 'matchType';

    // parameter values
    this.PARAM_VALUE_REQUEST_TYPE_TERM_SEARCH = 'term_search';
        this.PARAM_VALUE_REQUEST_TYPE_TERM_LOOKUP = 'term_lookup';
    this.PARAM_VALUE_OUTPUT_FORMAT_JSON = 'json';
        this.PARAM_VALUE_MATCH_TYPE_EXACT_TEXT = 'exact_text';

    // instance variables to hold parameters for request, from GCION
cookie.
    this.jsonFunction = '';
        this.jsonVariable = '';
    this.outputFormat = '';
    this.password = '';
        this.requestType = '';
        this.search = '';
    this.username = '';

        // variables for term_lookup request type.
        this.lookup = '';
        this.matchType = '';

    // bind functions
    this.processResponse = DataCookbookService_processResponse;

    // declare functions
    /**
     * Uses variables internal to this object to create the URL of the
JSONP
     *    javascript include service for our request. Returns URL.
     *
     * @return String - request URL for Data Cookbook service.
     */
    this.createRequestURL = function()
    {
        // return reference
        var URL_OUT = '';

        // declare variables
        var paramPrefix = '?';

        // first, create the URL.
        URL_OUT = this.SERVICE_PROTOCOL + this.SERVICE_URL_PREFIX +
this.SERVICE_URL_HOST + this.SERVICE_URL_PATH;

        // see what properties have values, append those with values.

        // request type - this.requestType
        if ( this.requestType != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_REQUEST_TYPE + "=" +
this.requestType;
            paramPrefix = '&';
        }

        // username value - this.username
        if ( this.username != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_USERNAME + "=" +
this.username;
            paramPrefix = '&';
        }

        // password value - this.password
        if ( this.password != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_PASSWORD + "=" +
this.password;
```

```
            paramPrefix = '&';
        }

        // search value - this.search
        if ( this.search != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_SEARCH + "=" +
this.search;
            paramPrefix = '&';
        }

        // output format value - this.outputFormat
        if ( this.outputFormat != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_OUTPUT_FORMAT + "=" +
this.outputFormat;
            paramPrefix = '&';
        }

        // JSON callback function name - this.jsonFunction
        if ( this.jsonFunction != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_JSON_FUNCTION + "=" +
this.jsonFunction;
            paramPrefix = '&';
        }

        // JSON assignment variable name - this.jsonVariable
        if ( this.jsonVariable != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_JSON_VARIABLE + "=" +
this.jsonVariable;
            paramPrefix = '&';
        }

        // match type - this.matchType
        if ( this.matchType != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_MATCH_TYPE + "=" +
this.matchType;
            paramPrefix = '&';
        }

        // lookup value - this.lookup
        if ( this.lookup != '' )
        {
            URL_OUT += paramPrefix + this.PARAM_LOOKUP + "=" +
this.lookup;
            paramPrefix = '&';
        }

                return URL_OUT;
    }; //-- end function createRequestURL() --//
    /**
     * Outputs contents of this object as a string.
     *
     * @return string - contents of this object.
     */
    this.outputContents = function()
    {
        // return reference
        var string_OUT = '';

        // declare variables
        var currentPropName = '';
        var outputArray = new Array();
        var itemCount = 0;
        var currentValue = '';
        var currentValueType = '';

        // loop over properties
        for( currentPropName in this )
        {
            // add current property to output array
            currentValue = this[ currentPropName ];

            // only output strings, numbers, and booleans.
            currentValueType = typeof( currentValue );
            if ( ( currentValueType == "string" ) || (
currentValueType == "number" ) || ( currentValueType == "boolean" ) )
            {
                outputArray[ itemCount ] = currentPropName + "=\"" +
this[ currentPropName ] + "\"";
                itemCount++;
            }
        }

        // join the items in array together with "; ".
        string_OUT = outputArray.join( ";\n " );

        // append name of this class to the front.
        string_OUT = "Contents of DataCookbookService instance: " +
string_OUT;

        return string_OUT;
    } //-- end function outputContents() --//


    /**
     * Uses variables internal to this object to create the URL of the
JSONP
     *    javascript include service for our request, then writes the
request
     *    into a script tag in the div specified by SERVICE_DIV_ID.
     * preconditions: there must be a <div> on the page somewhere
whose ID is
     *    the value in SERVICE_DIV_ID.
     * postconditions: The service JSONP include will call the
callback
     */
    this.submitRequest = function()
    {
        // declare variables
        var requestURL = '';
        var scriptElement = null;

        // get request URL
        requestURL = this.createRequestURL();

        // debug - output the URL
        //alert( "Service request URL = " + requestURL );

        // now, append the script tag to load this file.  To start,
we'll try
        //    just popping it on at the end of the page.
        scriptElement = document.createElement( "script" );
        scriptElement.setAttribute( "src", requestURL );
        scriptElement.setAttribute( "type", "text/javascript" );
        document.body.appendChild( scriptElement );
    }; //-- end function submitJSONPRequest() --//


} //-- end DataCookbookService constructor/class definition --//

//=====================================================================
==========
// class-level stuff
//=====================================================================
==========

// static functions
DataCookbookService.processResponse =
DataCookbookService_processResponse;

// static variables
DataCookbookService.PARAM_VALUE_REQUEST_TYPE_TERM_SEARCH =
'term_search';
DataCookbookService.PARAM_VALUE_REQUEST_TYPE_TERM_LOOKUP =
'term_lookup';
DataCookbookService.PARAM_VALUE_OUTPUT_FORMAT_JSON = 'json';
DATA_COOKBOOK_SERVICE_USERNAME = '';
DATA_COOKBOOK_SERVICE_PASSWORD = '';
DATA_COOKBOOK_SERVICE_HOST = '';
DATA_COOKBOOK_SERVICE_PROTOCOL = '';


/*
```

```
 * Accepts the ID of the form element whose inputs we will use to
create a service
 *    request, uses it to create service instance, populates the
instance from the
 *    form's inputs, and then invokes the service.
 */
function DataCookbookService_processEvents( formID_IN )
{
    // declare variables
    var formElement = null;
    var requestType = '';
    var searchString = '';
    var exactMatchFlag = '';
    var request = null;


    //alert( "In DataCookbookService_processEvents, event = " +
eventID_IN );

    // got a form ID?
    if ( formID_IN != '' )
    {
        // retrieve the form element
        formElement = document.getElementById( formID_IN );

        // retrieve request type
        requestType = formElement.DataCookbook_requestType.value;

        // see if the exact match checkbox is present.
        if ( typeof( formElement.DataCookbook_exact_text ) !=
'undefined' )
        {
            // the field is there - get its checked value.
            exactMatchFlag =
formElement.DataCookbook_exact_text.checked;

            // see if we are checked.
            if ( exactMatchFlag == true )
            {
                // it is checked.  Switch request type to TERM_LOOKUP.
                requestType =
DataCookbookService.PARAM_VALUE_REQUEST_TYPE_TERM_LOOKUP;
            }
        }

        // got a request type?
        if ( requestType ==
DataCookbookService.PARAM_VALUE_REQUEST_TYPE_TERM_SEARCH )
        {
            // search.  Do we have a search string?
            searchString = formElement.DataCookbook_search.value;
            if ( searchString != '' )
            {
                // create a DataCookbookService object.
                request = new DataCookbookService();

                // set up the request
                request.requestType = requestType;
                request.search = searchString;
                request.username = DATA_COOKBOOK_SERVICE_USERNAME;
                request.password = DATA_COOKBOOK_SERVICE_PASSWORD;
                request.SERVICE_PROTOCOL =
DATA_COOKBOOK_SERVICE_PROTOCOL;
                request.SERVICE_URL_HOST = DATA_COOKBOOK_SERVICE_HOST;
                request.SERVICE_URL_PATH =
request.SERVICE_URL_PATH_TERM_SEARCH;
                request.outputFormat =
DataCookbookService.PARAM_VALUE_OUTPUT_FORMAT_JSON;
                request.jsonFunction = request.SERVICE_CALLBACK;

                //alert( "Contents of request: " +
request.outputContents() );

                // run the request
                request.submitRequest();
            }
            else
            {
                alert( "No search string, so no search." );
            }
        }
        else if ( requestType ==
DataCookbookService.PARAM_VALUE_REQUEST_TYPE_TERM_LOOKUP )
        {
            // search.  Do we have a search string?
            searchString = formElement.DataCookbook_search.value;
            if ( searchString != '' )
            {
                // create a DataCookbookService object.
                request = new DataCookbookService();

                // set up the request
                request.requestType = requestType;
                request.lookup = searchString;
                request.username = DATA_COOKBOOK_SERVICE_USERNAME;
                request.password = DATA_COOKBOOK_SERVICE_PASSWORD;
                request.SERVICE_PROTOCOL =
DATA_COOKBOOK_SERVICE_PROTOCOL;
                request.SERVICE_URL_HOST = DATA_COOKBOOK_SERVICE_HOST;
                request.SERVICE_URL_PATH =
request.SERVICE_URL_PATH_TERM_LOOKUP;
                request.outputFormat =
DataCookbookService.PARAM_VALUE_OUTPUT_FORMAT_JSON;
                request.jsonFunction = request.SERVICE_CALLBACK;
                request.matchType =
request.PARAM_VALUE_MATCH_TYPE_EXACT_TEXT;
```

```
                //alert( "Contents of request: " +
request.outputContents() );

                // run the request
                request.submitRequest();
            }
            else
            {
                alert( "No search string, so no search." );
            }
        }
        else //-- no search string. --//
        {
            alert( "Unknown request type " + requestType + "." );
        }
    }
    else
    {
        alert( "Error processing Data Cookbook service request: No
form ID passed in." );
    }

    return false;
} //-- end function DataCookbookService_processEvents() --//
```

### 4.4.3.2 AJAX JSONP Implementation Example HTML Code

```
<!-- Data Cookbook Term Search: Begin -->
<div id="searchBox">
    <h1>DataCookbook Term Search</h1>
    <form name="DataCookbook_term_search"
id="DataCookbook_term_search" onsubmit="return
DataCookbookService_processEvents( 'DataCookbook_term_search' );">
        Search for: <input type="text" name="DataCookbook_search"
id="DataCookbook_search" /><br />
        <input type="checkbox" name="DataCookbook_exact_text"
id="DataCookbook_exact_text" /> Exact match?<br />
        <br />
        <input type="submit"
onclick="DataCookbookService_processEvents( 'DataCookbook_term_search'
)" name="DataCookbook_search_button" id="DataCookbook_search_button"
value="Search!" /> <input type="hidden"
name="DataCookbook_requestType" value="term_search" />
    </form>
</div>
<div id="data_cookbook_search_results"
name="data_cookbook_search_results" style="display:none;">
    <h3>Search Results</h3>
    <div class="termList">
    <ul>
        <li>
            <p><a class="eventLink url"
href="http://idata.datacookbook.com/institution/terms/150">Sample</a><
/p>
                <p>blah blah blah blah blah blah blah blah blah blah
blah blah blah blah blah blah blah blah blah..</p>
        </li>
    </ul>
    </div><!-- end results list div -->
    <p><a class="more" href="http://idata.datacookbook.com/">Go to the
Data Cookbook...</a></p>
</div>
<!-- End Data Cookbook term search -->
```

# 5 Introduction to HTTP

In order to understand how to implement a Data Cookbook service request, one first needs a basic understanding of the HTTP protocol. HTTP stands for HyperText Transfer Protocol. It is the underlying language that your web browser speaks when it interacts with web servers to retrieve web pages for you, and to enable your using web applications.

HTTP is a synchronous protocol. This means that a request will always have a corresponding response – your browser asks a web server for a web page, and then the server returns a response that includes a status (200 = success; 404 or 500 = error) and the results of the request. When you are using a web browser, the response for most pages is HTML – the web page you are trying to view).

HTTP can be used to implement asynchronous requests – requests where the action is not completed until later. Asynchronous requests using HTTP can be implemented several ways. One way is as a series of synchronous requests – a first request to start the asynchronous service, with a response that indicates that the response was received; then a series of "is it done yet?" requests whose response is "No" until the service is done processing the request.

A sample request (from http://www.jmarshall.com/easy/http/):

```
GET /path/file.html HTTP/1.0
From: someuser@jmarshall.com
User-Agent: HTTPTool/1.0
[blank line here]
```

A sample response (from http://www.jmarshall.com/easy/http/):

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>Happy New Millennium!</h1>
(more file contents)
.
.
.
</body>
</html>
```

## 5.1 Header Vs. Body Of Request Response

Requests and responses are both broken out into two parts – the header and the body. The header of a request or a response contains information to help whoever or whatever is receiving the request understand how to process what it has received.

The header of a request contains cookies that apply to the site being accessed. It contains the URL of the resource being requested and the type of method being used (get, post, etc). It can also contain as many header variables as the program creating the request wants to add in. If the server is expecting certain headers, it can relatively easily find and use the values in header variables. If not, then they are often ignored. Some standard header variables are expected in both request and response headers. For example, the content type header, which tells the recipient the type of the data contained in the body of the request or response, and a header variable to hold the number of bytes in the response are expected.

The header of a response can contain cookies that the server wants to set in the client. It contains the status code of the request. It can also contain a set of header variables just as the request can.

Requests and responses can also contain a request body. Normally, only "post" requests can contain a body while all responses are allowed to have a body. In requests, the request body is where form inputs are stored when a form that has type "post" is submitted. The request body can also be used to hold more complicated data, such as XML that specifies the details of a service request. It is more common for responses to have a body. When you request a web page, the HTML for the page is stored in the body of the response. When you request a web service, the XML or JSON that holds the results of your service request are stored in the body of the response.

## 5.2 Header Variables

Both requests and responses can contain header variables. Header variables are name value pairs structured like:

```
<header_variable_name>: <header_variable_value>
```

Header variables are placed in the header part of an HTTP request, one to a line. Two carriage-return line-feed pairs (an empty line) separate the header from the body of the request.

An example header variable:

```
Content-Type: text/html
```

## 5.3 Cookies

Cookies are set by servers using the Set-Cookie header variable in a response's header. For example:

```
Set-Cookie: foo=bar; path=/; expires Mon, 09-Dec-2002 13:46:00
GMT
```

They are sent to a server using the Cookie header variable in a request's header For example:

```
Cookie: foo=bar; cookie2=value2; ...
```

They can be used to help a server keep track of information on a given user or session across requests, since HTTP is a stateless protocol – HTTP doesn't contain built-in mechanism for tying subsequent related requests together.

## 5.4 Request Methods

A given request can have one of eight HTTP request methods:

- OPTIONS
- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE
- CONNECT

An HTTP request must have one of these request methods specified. The most commonly used are GET and POST, used for form submissions. In an HTTP request, the method is the first thing in the first line of the request. So, in the following example, the request method is "GET":

```
GET /path/file.html HTTP/1.0
```

## 5.5 Form Inputs

When you use HTTP to pass form requests, like when a login form logs you into a website, or a registration form collects information from you so you can be added to the users on a site, the information from the form is passed as part of an HTTP request in one of two ways.

If your form uses a "GET" request method, then the parameters are appended to the URL of the page that will process the form submission, in a query string made up of name-value pairs of form inputs. The query string starts with a question mark ("?") and then is followed by one or more form inputs where the name and its value are separated by an equal sign ("="). Each parameter is separated by an ampersand ("&"). An example:

```
http://idata.datacookbook.com/sessions/service_login?un=jonathanmorgan
&pw=nopeeking!
```

This request has two parameters – a username (name = "un", value = "jonathanmorgan") and a password (name = "pw", value = "nopeeking!").

If your form uses a "POST" request method, then the parameters are stored in the request body instead of on the end of the URL, in pretty much the same format, except there is no "?" at the beginning of the parameter list. So, a POST request of the same thing would look like:

```
POST /sessions/service_login HTTP/1.0
Host: idata.datacookbook.com
[blank line here]
un=jonathanmorgan&pw=nopeeking!
```

## 5.6 Passing XML in Body of Request

When you use the POST request method, you can also pass other things in the body of the request. In the case of Data Cookbook services, you can pass XML in a request. Example:

```
POST /sessions/service_login HTTP/1.0
Host: idata.datacookbook.com
[blank line here]
<?xml version="1.0" encoding="UTF-8"?>
<ServiceTransaction xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ServiceRequest serviceName="service_login">
        <ParameterList>
            <Parameter>
                <Name>un</Name>
                <Value>jonathanmorgan</Value>
            </Parameter>
            <Parameter>
                <Name>pw</Name>
                <Value>nopeeking!</Value>
            </Parameter>
            <Parameter>
                <Name>requestType</Name>
                <Value>service_login</Value>
            </Parameter>
             <Parameter>
                <Name>outputFormat</Name>
                <Value>json</Value>
            </Parameter>
        </ParameterList>
    </ServiceRequest>
</ServiceTransaction>
```

## 5.7 HTTP Response

An HTTP response is very similar to a request. The main differences are that it has a different first line (the status line) and it can always have a body.

It is always in the form:
```
<HTTP_version> <status_code> <status_message>
<header_list>

<response_body>
```

HTTP_version will be in the format “HTTP/X.X” and the status code will be a three-digit number. You can tell the type of status by the first digit:

- 1XX - informational
- 2XX - success
- 3XX - redirect
- 4XX - error in something the client did
- 5XX - server error

The status message is intended to be human readable, and it might vary from server to server for the same status code.

To parse out the headers and go straight to the body, look for two new-lines in a row. Cookies that are returned by the server are stored in a “Set-Cookie” header variable, with one cookie on the same line as the Set-Cookie header variable name, and one cookie per line after that if there are multiple cookies.

***Sample HTTP response:***

```
HTTP/1.0 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/xml
Content-Length: 1563
Set-Cookie:
_itdb_session=BAh7CToMdXNlcl9pZGlDIgpmbGFzaElDOidBY3Rpb25Db250cm9sbGVy
OjpGbGFzaDo6Rmxhc2hIYXNoewY6C25vdGljZSICjwF7ImZsYXNoIj0+e30sIDpfY3NyZl
90b2tlbj0+IjJ0bHhrOHZacTNGMnRhUlFRTmM3cDd4Tk15T1NCbjV3MnRPcGhodGJ2ODA9
IiwgOnNlc3Npb25faWQ9PiIzZjY5YjI3ZmUyNWY4MzRmNGQ0ZGRhOTI4NDliMGFkNSJ9Q2
9va2llPzogQkFoN0NDSUtabXhoYzJoSlF6b25RV04wYVc5dVEyOXVkSEp2Ykd4bGNqbzZS
bXhoYzJnNk9rWnNZWE5vU0dGemFIc0FCam9LUUhWelpXUjdBRG9RWDJOemNtWmZkRzlyWl
c0aU1USjBiSGhyT0haYWNUTkdNblJoVWxGUlRtTTNjRGQ0VGsxNVQxTkNialYzTW5SUGNH
aG9kR0oyT0RBOU9nOXpaWE56YVc5dVgybGtJaVV6WmpZNVlqSTNabVV5TldZNE16Um1OR1
EwWkdSaE9USTRORGxpTUdGa05RPT0tLWNiMDBjYTEwNWY0ZGI1OTIxNDFhM2RiMjU4NDhj
NzU1ODkxMjc1M2MGOgpAdXNlZHsGOwdUOhBfY3NyZl90b2tlbiIxMnRseGs4dlpxM0YydG
FSUVFOYzdwN3hOTXlPU0JuNXcydE9waGh0YnY4MD06D3Nlc3Npb25faWQiJTNmNjliMjdm
ZTI1ZjgzNGY0ZDRkZGE5Mjg0OWIwYWQ1--
```

```
11d1e4a6afab5ce50b1c181c5f6cb3bdb373f853; path=/;
domain=idata.datacookbook.local; secure=false
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Cache-Control: no-cache
Pragma: no-cache
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceTransaction xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ServiceResponse serviceName="term_search">
        <ResponseStatus>
            <ResponseCode>0</ResponseCode>
            <ResponseMessage>Success!</ResponseMessage>
        </ResponseStatus>
        <TermList>
            <Term>
                <name>hungry student</name>
                <functional-definition>A hungry student is one who has
a need that he or she feels must be met.  This could be a hunger for
food.  It could be a hunger for knowledge.  More specific child terms
differentiate.</functional-definition>
                <perma-link-
url>http://idata.datacookbook.com/terms/12345</perma-link-url>
            </Term>
             <Term>
                <name>hungry student (food)</name>
                <functional-definition>A student who is hungry for
food is one who feels a need to eat food.  Might or might not actually
need food.</functional-definition>
                <perma-link-
url>http://idata.datacookbook.com/terms/12346</perma-link-url>
            </Term>
            <Term>
                <name>hungry student (knowledge)</name>
                <functional-definition>A student who is hungry for
knowledge is one who is driven to learn, to ingest knowledge.  Similar
to thirsting for knowledge (see term "thirsty student
(knowledge)".</functional-definition>
                <perma-link-
url>http://idata.datacookbook.com/terms/12347</perma-link-url>
            </Term>
       </TermList>
    </ServiceResponse>
</ServiceTransaction>
```