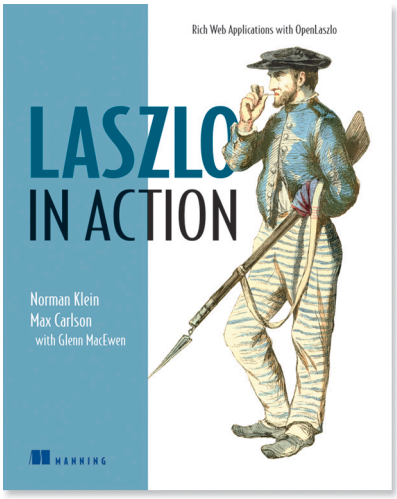
Interfacing OpenLaszlo Applications to Web Services

An Excerpt from



"Laszlo in Action"

By Norman Klein and Max Carlson with Glenn MacEwen
Early Access Release: January 2007
Print Release: November 2007 | 500 pages
ISBN: 1-932394-83-4
www.manning.com/laszloinaction

Interfacing OpenLaszlo Applications to Web Services

If you've been enviously eyeing the RIA capabilities of OpenLaszlo, but been daunted by its interface requirements, then you'll be glad to know that interfacing to OpenLaszlo is just like that old Talking Heads song "same as it ever was". In other words, OpenLaszlo uses the same HTTP standards that you're familiar with; allowing OpenLaszlo applications to co-exist with your HTML web applications to further leverage your server-side web services. All OpenLaszlo communications with the server are still based on sending and receiving HTTP requests and responses. OpenLaszlo operates just like Ajax by requiring HTTP responses to be composed of XML instead of HTML. The technical term for this is a ReST-based "XML-over-HTTP" service, but this is only a fancy way of describing the familiar operation of the web. So why does it use XML and not HTML? Since an RIA can provide a superior visual presentation, only the data is needed as the HTML presentation details are superfluous. OpenLaszlo's compliance to open standards allows it to interface to any HTTP web server; ranging from basic HTTP web servers such as Apache, Jetty or Microsoft's IIS to the different web frameworks such as Struts, Tapestry, Ruby on Rails, Microsoft .Net that work with servlet containers to provide enterprise class services. While RIA might be a revolutionary user interface technology, OpenLaszlo thankfully only requires a small evolutionary step for developers.

To demonstrate how simple it is to establish OpenLaszlo as an alternative front-end client, we'll create a OpenLaszlo application to communicates with an HTTP server. But before we can get started, we first need to ensure that our HTTP server can deliver XML output. Since RSS (Really Simple **S**yndication*)* newsfeeds provide a good example of a ReST-based "XML-over-HTTP" service, we'll use it for our initial example. RSS news feeds are a familiar feature on the Web as they aggregate syndicated web content such as news reports into a list of headlines. For example, to access the top stories from Yahoo, we'd type in the following URL:

```
http://rss.news.yahoo.com/rss/topstories
```

This will return an XML document, displayed in Listing 1, whose data composition conforms to the RSS 2.0 specification.

```
<rss version="2.0" xmlns:media="http://search.yahoo.com/mrss/">
   <channel>
      <title>Yahoo! News: Top Stories</title>
      <item>
          <title>Populations of 20 common birds declining (AP)</title>
          <link>http://us.rd.yahoo.com/dailynews/rss/topstories</link>
          ...
      </item>
      <item>
        ...
      </item>
</rss>
```

All RIAs, including OpenLaszlo, use XML to store their state within a data cache; as it allows them to execute independently like a desktop application. This XML needs to be complete and consistent, also known as well-formed, so its contents conform to the general rules of XML. Within a OpenLaszlo application, this data cache consists of a series of dataset objects. From OpenLaszlo's viewpoint, data is data and it is indifferent to its origins. So a dataset can contain local XML data compiled into an application or XML data returned from a web server. This data equivalence supports a development cycle allowing coding to be initiated with local test datasets and then later transitioned to server supplied datasets. This dramatically improves productivity, since it removes the coupling between client and server development.

Now that we have an HTTP server delivering RSS content, we'll next cover the attributes and methods of a dataset object, before creating a OpenLaszlo application to display this RSS output. Afterwards, we'll examine the limitations of this approach and demonstrate the benefits of using buffered datasets. Finally, we'll implement a *framework* that can be used across different datasets to easily generate and send HTTP requests that results in its response handler methods being automatically invoked. By the conclusion, we'll have a general approach for handling HTTP data that is easily extended to work with any request. The source code for these examples can be found at www.manning.com/laszloinaction in the Resources section.

Using datasets with HTTP

Datasets support every configuration feature within the HTTP standard, so we'll limit ourselves to the most important and heavily used features. A dataset is an unusual OpenLaszlo object since it is descended from multiple parents. It's derived from the LzNode object, to allow it to be used as a declarative tag, and from the LzDataElement object to gain access to its data manipulation methods. A dataset is flexible enough to supply a simple interface for accessing local data while also supporting all of the optional settings defined by the HTTP standard. A dataset's `src` attribute determines the location of the data and can have these values:

- none: Data is contained in the dataset.
- pathname: A local XML file is compiled into the application.
- URL: An absolute or relative URL points to HTTP-based data.

A dataset is smart enough to operate appropriately based on its `src` attribute setting. If the attribute is omitted or set to a local file, then we'll be using a local dataset and when it's set to a URL then we'll interfacing to HTTP-based data. This allows an application's development cycle to easily transition from using local test data to networking with a production server.

By default, OpenLaszlo sends HTTP GET requests, but a dataset's `setQueryType` method can be used to change this. Requests with a large number of query parameters should be sent with HTTP POST to ensure that data is sent as part of the request's body. Tables 1 and 2 list some of the most commonly used dataset attributes and methods. To see the complete listing of attributes and methods for a dataset, view the latest reference manual at the OpenLaszlo website:

http://www.openlaszlo.org/lps4/docs/reference

Table 1 Commonly used dataset attributes

Name	Data Type	Attribute	Defaults	Description
querystring	string	read-only		A string appended to a dataset request
request	boolean	setter		When true, the dataset makes a request when it begins the init stage.
secure	boolean	setter	false	Specifies whether or not the app-LPS connection is secure
secureport	number	setter	443	The port number to use to connect to the LPS for a secure connection
Src	string	setter		The source for requests made by this dataset
timeout	number	setter	30000	The timeout period in milliseconds for load requests
Type	string	setter		If set to "http", the dataset interprets its src attribute as a URL from which to load its content, rather than a static XML file to inline.

To ensure that an application is initially populated with data, the dataset's `request` attribute should be set to true. When application initialization completes, an initial HTTP request is sent to populate each dataset. Datasets can also be manually populated by calling their `doRequest` method.

Table 2 Commonly used dataset methods

Name	Description
abort()	Stops loading the dataset's current request
doRequest()	Issues a request immediately using the current values; if autorequest is true, this method is called automatically when values change.
getResponseHeader(name)*	Returns the value for the specified response header, or false if there is no header with that name; if name is omitted, all response headers are returned as an object of name/value pairs.
getSrc()	Returns the src attribute of the dataset
setHeader(key, val)*	Sets a header for the next request
setQueryParam(key, val)	Sets a named query parameter to the given value
setQueryParams(assoc array)	Sets multiple query parameters using the keys in the argument as keys and the values of those keys as values
setQueryString(string)	Sets the querystring parameter of the dataset to the given string
setQueryType(reqtype)	Sets the query type for the parent datasource to one of POST or GET by calling the method of the same name on this dataset's datasource
setRequest(boolean)	Sets whether or not the dataset makes its request on initialization
setSrc(src)	Sets the src attribute of the dataset's parent datasource

***Subject to platform capabilities (not available on Flash unless in proxy mode)**

We'll now use the attributes and methods, listed in Table 1 and 2, to demonstrate how easy it is to create a OpenLaszlo application to display the results from a RSS news feed within a window component.

Implementing an RSS news feed

Datapaths and datasets make accessing this RSS information simple. In Listing 1, to initially populate the display with a listing of the latest headlines, the `request` attribute for the `newsfeed` dataset is set to true. Since the data contained within these feeds is RSS-compliant, their data composition can be expressed with a single set of datapath expressions. We can now change the `src` attribute of our dataset to point to any other RSS 2.0-compliant news feed with the assurance that our datapath expressions will still work correctly. In our example, we'll toggle between the CNN and Yahoo news feeds by clicking the "Change" button. This example could easily be expanded to add additional selections.

Listing 1 Accessing RSS news feeds

```
<canvas>
   <dataset name="newsfeed" request="true"                                    #1
            src="http://rss.news.yahoo.com/rss/topstories"/>
   <button y="50" text="Change">
      <attribute name="cnt" value="1" type="number"/>
      <handler name="onclick">
         if (this.cnt % 2) {
            newsfeed.setSrc("http://rss.cnn.com/rss/cnn_topstories.rss");} #2
         else newsfeed.setSrc("http://rss.news.yahoo.com/rss/topstories");
         newsfeed.doRequest();                                              #3
         this.setAttribute("cnt", this.cnt+1);
      </handler>
   </button>
   <window title="RSS Reader" x="80" height="150" width="350">
      <view>
         <view datapath="newsfeed:/rss/channel/item">
            <text name="txt" fontsize="9" resize="true"
                  text="$path{'title/text()'}" fgcolor="blue"/>
         </view>
         <simplelayout axis="y"/>
      </view>
      <scrollbar/>
   </window>
</canvas>
```

(annotate)<#1 Generate initial display>
(annotate)<#2 Change news source>
(annotate)<#3 Send request to server>

The dataset's `request` attribute #1, as seen in Figure 1, generates the initial display of the Yahoo RSS news headlines. Pressing the Change button updates the `src` attribute #2 with the URL for CNN's RSS feed. After this update, the `doRequest` method #3 sends the request to the server to initiate a response.

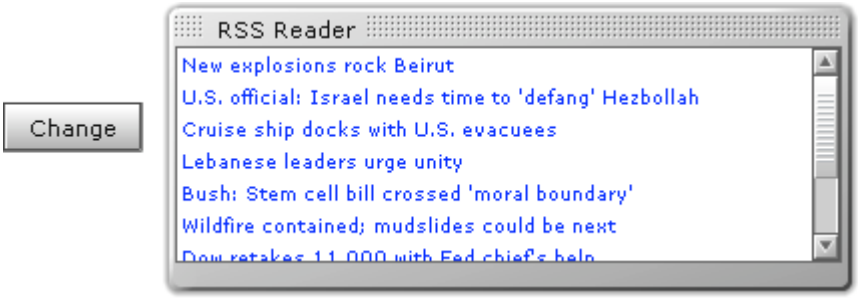


Figure 1 Titles from the Yahoo RSS news feed are displayed initially. A press on the Change button switches the RSS news feed to CNN.

Although a single dataset works adequately within this trivial example to demonstrate the basic dataset features, the shortcoming of this approach become apparent when we have to deal with real-world Internet problems such as HTTP servers timing out or being down. In these cases, our RSS

Reader would lose its current display listing and only display an HTTP server exception code such as "HTTP 404" or "HTTP 500". To provide users with a seamless viewing experience requires that buffered datasets be used to handle HTTP transfers.

Buffered datasets

Before an application discards its current display, it must have an acceptable next state. So, when an error or timeout blocks meaningful data, a supplemental error message should appear, but the current data state should be maintained. This allows a degraded but still usable application.

Maintaining this visual continuity requires multiple datasets; a *buffering dataset* to handle data transfers and a *destination dataset* for interfacing and binding to visual objects. Only when a data transmission successfully completes is the destination dataset updated with the buffered contents. This approach centralizes error processing and simplifies processing since the destination dataset is guaranteed to always receive valid data.

Here's a typical set of declarations for a set of buffering and destination datasets; the buffering dataset provides handlers for the three possible outcomes of a request-response sequence: data, error, or timeout. In this case, valid data is handled within a "handleData" method, errors within a "handleErrors" method and timeout situations within a "handleTimeout" method.

```
<dataset name="dsBuffer" type="http"
         ondata="this.handleData(this);"
         onerror="this.handleError(this);"
         ontimeout="this.handleTimeout(this);">
</dataset>                                              #1
<dataset name="dsProducts"/>                            #2
```

(annotate)<#1 Buffering dataset>
(annotate)<#2 Destination dataset>

The HTTP standard defines any response status within the 2xx range to be valid, resulting in the `ondata` handler being called. A response within the 4xx or 5xx range is an error, resulting in the `onerror` handler being called. The delay invoking `ontimeout` is configurable in the `timeout` attribute, defaulting to 30 seconds.

Although this provides a workable system for communicating with an HTTP server, it requires a buffering dataset for every destination dataset. With a large number of datasets, this is a significant overhead. A better solution is to pool the buffering datasets among the display datasets. This is the rationale behind the `LzHttpDatasetPool` service.

Pooling buffering datasets

The `LzHttpDatasetPool` service maintains a pool of buffering datasets for sharing among any number of destination datasets. Datasets are not pre-allocated to the pool; an additional dataset is created whenever required. Consequently, the number of pooled datasets increases to handle the highest level of traffic. Released datasets are put back into the pool for the next HTTP connection.

The `LzHttpDatasetPool` object has two methods; `get` retrieves a pooled dataset and `recycle` releases this dataset back into the pool. A call to `get` a buffering dataset looks like this:

```
var ds = LzHttpDatasetPool.get(this.dataDel, this.errorDel,
                               this.timeoutDel);
```

where:

dataDel	is a delegate for the `ondata` event
errorDel	is a delegate for the `onerror` event
timeoutDel	is a delegate for the `ontimeout` event

A delegate allows a method to be associated with each event. The `once` qualifier ensures that the "dsLoad", "dsError", and "dsTimeout" methods are set up only once:

```
<attribute name="dataDel"    value="$once{new LzDelegate(this, 'dsLoad')}"/>
<attribute name="errorDel"   value="$once{new LzDelegate(this, 'dsError')}"/>
<attribute name="timeoutDel" value="$once{new LzDelegate(this, 'dsTimeout')}"/>
```

The `LzHttpDatasetPool` service ensures that all buffered datasets are always clean, empty and ready for use to communicate with an HTTP server.

Building a data service

HTTP server communications is still built upon requests and responses. But we'll still want to keep the server implementation isolated within a class. This allows applications to be easily updated later to work with other networking technologies, such as SOAP or XML-RPC. Instead of directly invoking the methods of our dataset, we'll create a data service object to encapsulate the HTTP communication details for each dataset.

A data service's interface consists of matching sets of request and response methods corresponding to each server related service required by a dataset. The *request method* retrieves data from the HTTP server and the *response method* handles the server's response. A further level of encapsulation lies behind this data service interface to contain the common data transfer operations involving buffered datasets.

Let's suppose that we have a "dsProducts" dataset contained within a "productDataService" object whose request method is called "getProducts". The productDataService's "getProducts" method is used to populate the "dsProducts" dataset with new products and looks like this:

```
productDataService.getProducts("New");
```

The advantage of using data service objects is they can easily be integrated into an application. Suppose we controlled an application's operation through a state controller. Then populating a product listing for a particular screen can be controlled by adding a call to the productDataService's "getProducts" method:

Listing 2 Sample application state controller

```
<node id="gController">
   <method event="onappstate" args="state">
      var title = "";
      switch (state) {
         …
         case "Login to Main":
              this.setAttribute("currstate", "Main");
              productDataService.getProducts("new");
              break;
```

```
        … }
    </method>
    …
</node>
```

The dataset's `pathurl` attribute provides a base URL. We'll construct a request from the base URL and an associate array to hold the URL parameters. An `error` attribute is used to alert other view objects about network problems, this allows any visual object to easily provide a visual error indicator to users.

Listing 3 productDataService.lzx

```
<library>
    <dataset name="dsProducts"/>
    <node name="productDataService">
        <attribute name="error" type="string" value=""/>
        <attribute name="pathurl" type="string"
                   value="http://www.laszlomarket.com/store/"/>
        …
    </node>
    …
</library>
```

A `getProductParams` method packages the URL parameters into an associative array. Although OpenLaszlo provides an `LzParams` utility for handling HTTP parameters, it's a bit of overkill, so we'll just stick with a JavaScript associative array in Listing 4 to build the query string ourselves:

Listing 4 productDataService.lzx (cont.)

```
<method name="getProductParams">
    var params = {};
    params.action   = "list";
    params.category = "new";
    return params;
</method>
```

Listing 5 lists the data service object's request and response methods to retrieve all products from the HTTP server for the "dsProducts" dataset. Although any name can be used for the response, we'll adopt a convention of using the request name and appending "`Result`" to it.

Listing 5 productDataService.lzx (cont.)

```
<library>
    <include href="DataService.lzx"/>                                  #1
    <dataset name="dsProduct"/>
    <node name="productDataService">
        …
        <method name="getProducts">
            var requesturl = pathurl + "products.do";
            var params = getProductParams();
            gDataService.sendRequest(this, requesturl, params,
                                     "getProductsResult");          #2
```

```
      </method>
      <method name="getProductsResult" args="status, data">
         if (status == true)
            this.appendChild(data.getFirstChild());            #3
         else
            Debug.write("getProductsResult Failed: " + data);
      </method>
   </node>
</library>
```

(annotate)<#1 Include gDataService object>
(annotate)<#2 Establish request-response link>
(annotate)<#3 Update dataset>

All that remains is to encapsulate the common data transfer operations for using buffering datasets, and to connect the request and response methods.

Implementing data transfer operations

To support the clean interface of our data services, we've packaged the common underlying methods into a globally accessible node-based object called `gDataService`. Its functionality includes:

- obtaining and disposing of a buffering dataset
- transferring data from the buffering to the destination dataset
- asynchronously invoking the response method
- standardized error handling

To make it globally accessible within Listing 5, the "gDataService" object is defined as a top-level object within a library as shown within Listing 6. It uses the `LzHttpDatasetPool` service with this set of delegate attributes to attach methods for the data, error, and timeout events. The `sender` argument of `sendRequest` corresponds to the object, in this case to the "dsProducts" dataset that invokes the request. The buffering dataset stores the names of both the sending object and the response method to set up the automatic invocation of the response method.

Listing 6 gDataService.lzx : Encapsulate common HTTP functionality

```
<library>
   <node name="gDataService">
      <attribute name="loadDel"                                        |#1
                 value="$once{new LzDelegate(this, 'dsLoad')}"/>       |
      <attribute name="errorDel"                                       |
                 value="$once{new LzDelegate(this, 'dsError')}"/>      |
      <attribute name="timeoutDel"                                     |
                 value="$once{new LzDelegate(this, 'dsTimeout')}"/>    |

      <method name="sendRequest" args="sender, url, params, response">
         var ds = LzHttpDatasetPool.get(this.loadDel, this.errorDel,
                                        this.timeoutDel);
         ds.setAttribute("sender" , sender);                          |#2
         ds.setAttribute("response", response);                       |
         ds.setSrc(url);                                              |#3
         ds.setQueryType('POST');                                     |
```

```
            ds.setQueryParams(params);                                    |
            ds.doRequest();                                               |#4
        </method>
        …
    </node>
    …
</library>
```

(annotate)<#1 Set up data, error, and timeout delegates>
(annotate)<#2 Save sender name and response method>
(annotate)<#3 Update HTTP settings>
(annotate)<#4 Send HTTP request>

When a valid HTTP response arrives, the dataset receives an `ondata` event which is handled by the `dsLoad` method with the buffering dataset as an argument. Although the response is valid, it must still be checked for a status message indicating server-side processing errors. When the HTTP server returns an error, the returned XML response consists of a status element with a set `error` attribute and a `message` attribute containing an error description:

```
<response>
   status error="true" message="Can't find any products"/>
</response>
```

This approach, shown in listing 7, centralizes error processing and simplifies the response method since it is now guaranteed to only receive valid data.

Listing 7 gDataService.lzx (cont)

```
<method name="dsLoad" args="ds">
   var ebyt = ds.getFirstChild().getElementsByTagName("status");      #1
   if (ebyt.length){
      if (ebyt.getAttr("error") == true) {                            #2
         var msg = ebyt.getAttr("message");
         this.setAttribute("error", "Request Failure: " + msg);       #3
         LzHttpDatasetPool.recycle(ds);
         return; } }
   ds.callback[ds.response](ds.getFirstChild());                      #4
   LzHttpDatasetPool.recycle(ds);
   return;
</method>
```

(annotate)<#1 Find status code>
(annotate)<#2 Check its error attribute>
(annotate)<#3 Display error message>
(annotate)<#4 Invoke response handler>

Method `dsLoad` first checks the status code #1 by searching through the first child node for a node named `status`. If a status node #2 is found, its `error` attribute is checked and its `message` attribute #3 is displayed in an error window. Remember that `getAttr` retrieves XML attributes, while `setAttribute` sets dataset attributes. Line #4 performs the magic to invoke the response handler. This line of code requires an extra bit of explanation.

An object in JavaScript is represented by a name-value associative array. Since JavaScript treats functions as data—they be used anywhere a data value can be used—a value in this array can be a function. This is important here because the response string can be an identifier to access the response method. Since this is a method, it can naturally take an argument, which is an LzDataElement object containing the data returned by the server, minus its wrapping node.

```
ds.sender[ds.response](ds.getFirstChild());
```

When a valid HTTP response arrives, the productDataService's "getProductsDataResult" method is automatically invoked with an argument containing the returned XML data. This allows the appropriate response handler to be automatically called.

Finally, standardized methods for handling the error and timeout conditions are required. When one of these conditions occurs, the gDataService's error attribute is set with the failure reason and the buffered dataset is released back into the pool:

Listing 8 gDataService.lzx (cont)

```
<method name="dsError" args="ds">
   this.setAttribute('error', 'The request failed ' + ds.src);
   LzHttpDatasetPool.recycle(ds);
</method>
<method name="dsTimeout" args="ds">
   this.setAttribute('error', 'The request timed out. ' + ds.src);
   LzHttpDatasetPool.recycle(ds);
</method>
```

Any visual object can use the `error` attribute as a constraint to ensure that errors messages are automatically displayed. This provides a wide amount of latitude in how these messages are displayed.

Hopefully, you'll find that using this data service framework makes working in OpenLaszlo as straight-forward and easy as any other web environment. The dataset data service approach provides scalability to support additional server-side services; allowing each service to be easily supported by adding an additional set of request and response methods to a dataset's data service. Since we're able to re-use the existing pooled dataset facilities, these additional services only require a limited increase in resources. This allows an application to have an almost unlimited number of data service supported datasets. Now that we've successfully tackled the issue of interfacing to an HTTP server, this opens the door to use your existing web services in innovative new ways within an OpenLazlo application.